

---

# PyTorch-Metrics Documentation

*Release 1.0.0*

Lightning-AI et al.

Jul 05, 2023



# USER GUIDE

<b>1</b>	<b>Install TorchMetrics</b>	<b>3</b>
1.1	Quick Start	3
1.2	All TorchMetrics	6
1.3	Structure Overview	9
1.4	Plotting	22
1.5	Implementing a Metric	31
1.6	TorchMetrics in PyTorch Lightning	39
1.7	Concatenation	42
1.8	Maximum	43
1.9	Mean	45
1.10	Minimum	47
1.11	Running Mean	49
1.12	Running Sum	50
1.13	Sum	51
1.14	Complex Scale-Invariant Signal-to-Noise Ratio (C-SI-SNR)	52
1.15	Perceptual Evaluation of Speech Quality (PESQ)	55
1.16	Permutation Invariant Training (PIT)	59
1.17	Scale-Invariant Signal-to-Distortion Ratio (SI-SDR)	59
1.18	Scale-Invariant Signal-to-Noise Ratio (SI-SNR)	60
1.19	Short-Time Objective Intelligibility (STOI)	60
1.20	Signal to Distortion Ratio (SDR)	63
1.21	Signal-to-Noise Ratio (SNR)	64
1.22	Speech-to-Reverberation Modulation Energy Ratio (SRMR)	65
1.23	Accuracy	68
1.24	AUROC	84
1.25	Average Precision	97
1.26	Calibration Error	111
1.27	Cohen Kappa	120
1.28	Confusion Matrix	128
1.29	Coverage Error	139
1.30	Dice	142
1.31	Exact Match	147
1.32	F-1 Score	157
1.33	F-Beta Score	172
1.34	Group Fairness	187
1.35	Hamming Distance	194
1.36	Hinge Loss	210
1.37	Jaccard Index	218
1.38	Label Ranking Average Precision	230
1.39	Label Ranking Loss	233

1.40	Matthews Correlation Coefficient	236
1.41	Precision	247
1.42	Precision At Fixed Recall	261
1.43	Precision Recall Curve	274
1.44	Recall	289
1.45	Recall At Fixed Precision	304
1.46	ROC	317
1.47	Specificity	332
1.48	Specificity At Sensitivity	347
1.49	Stat Scores	357
1.50	Complete Intersection Over Union (cIoU)	369
1.51	Distance Intersection Over Union (dIoU)	373
1.52	Generalized Intersection Over Union (gIoU)	377
1.53	Intersection Over Union (IoU)	381
1.54	Mean-Average-Precision (mAP)	385
1.55	Modified Panoptic Quality	390
1.56	Panoptic Quality	391
1.57	Error Relative Global Dim. Synthesis (ERGAS)	392
1.58	Frechet Inception Distance (FID)	394
1.59	Image Gradients	397
1.60	Inception Score	397
1.61	Kernel Inception Distance	400
1.62	Learned Perceptual Image Patch Similarity (LPIPS)	403
1.63	Memorization-Informed Frechet Inception Distance (MiFID)	406
1.64	Multi-Scale SSIM	409
1.65	Peak Signal-to-Noise Ratio (PSNR)	409
1.66	Peak Signal To Noise Ratio With Blocked Effect	410
1.67	Relative Average Spectral Error (RASE)	413
1.68	Root Mean Squared Error Using Sliding Window	413
1.69	Spectral Angle Mapper	414
1.70	Spectral Distortion Index	414
1.71	Structural Similarity Index Measure (SSIM)	415
1.72	Total Variation (TV)	415
1.73	Universal Image Quality Index	416
1.74	CLIP Score	416
1.75	Cramer's V	420
1.76	Fleiss Kappa	424
1.77	Pearson's Contingency Coefficient	427
1.78	Theil's U	431
1.79	Tschuprow's T	435
1.80	Cosine Similarity	439
1.81	Euclidean Distance	440
1.82	Linear Similarity	441
1.83	Manhattan Distance	442
1.84	Minkowski Distance	443
1.85	Concordance Corr. Coef.	444
1.86	Cosine Similarity	446
1.87	Explained Variance	449
1.88	Kendall Rank Corr. Coef.	452
1.89	KL Divergence	456
1.90	Log Cosh Error	459
1.91	Mean Absolute Error (MAE)	462
1.92	Mean Absolute Percentage Error (MAPE)	464
1.93	Mean Squared Error (MSE)	467

1.94	Mean Squared Log Error (MSLE)	469
1.95	Minkowski Distance	471
1.96	Pearson Corr. Coef.	474
1.97	R2 Score	476
1.98	Relative Squared Error (RSE)	480
1.99	Spearman Corr. Coef.	483
1.100	Symmetric Mean Absolute Percentage Error (SMAPE)	485
1.101	Tweedie Deviance Score	488
1.102	Weighted MAPE	491
1.103	Retrieval Fall-Out	493
1.104	Retrieval Hit Rate	494
1.105	Retrieval Mean Average Precision (MAP)	494
1.106	Retrieval Mean Reciprocal Rank (MRR)	495
1.107	Retrieval Normalized DCG	495
1.108	Retrieval Precision	496
1.109	Precision Recall Curve	496
1.110	Retrieval R-Precision	497
1.111	Retrieval Recall	497
1.112	BERT Score	498
1.113	BLEU Score	503
1.114	Char Error Rate	503
1.115	ChrF Score	504
1.116	Extended Edit Distance	504
1.117	InfoLM	505
1.118	Match Error Rate	509
1.119	Perplexity	509
1.120	ROUGE Score	512
1.121	Sacre BLEU Score	516
1.122	SQuAD	516
1.123	Translation Edit Rate (TER)	517
1.124	Word Error Rate	517
1.125	Word Info. Lost	518
1.126	Word Info. Preserved	518
1.127	Bootstrapper	519
1.128	Classwise Wrapper	521
1.129	Metric Tracker	524
1.130	Min / Max	528
1.131	Multi-output Wrapper	530
1.132	Multi-task Wrapper	533
1.133	Running	537
1.134	torchmetrics.Metric	539
1.135	torchmetrics.utilities.data	544
1.136	TorchMetrics Governance	545
1.137	Contributor Covenant Code of Conduct	547
1.138	Contributing	548
1.139	Changelog	551
<b>2</b>	<b>Indices and tables</b>	<b>575</b>
	<b>Index</b>	<b>577</b>



TorchMetrics is a collection of 100+ PyTorch metrics implementations and an easy-to-use API to create custom metrics. It offers:

- A standardized interface to increase reproducibility
- Reduces Boilerplate
- Distributed-training compatible
- Rigorously tested
- Automatic accumulation over batches
- Automatic synchronization between multiple devices

You can use TorchMetrics in any PyTorch model, or within [PyTorch Lightning](#) to enjoy the following additional benefits:

- Your data will always be placed on the same device as your metrics
- You can log *Metric* objects directly in Lightning to reduce even more boilerplate





## INSTALL TORCHMETRICS

For pip users

```
pip install torchmetrics
```

Or directly from conda

```
conda install -c conda-forge torchmetrics
```

### 1.1 Quick Start

TorchMetrics is a collection of 100+ PyTorch metrics implementations and an easy-to-use API to create custom metrics. It offers:

- A standardized interface to increase reproducibility
- Reduces Boilerplate
- Distributed-training compatible
- Rigorously tested
- Automatic accumulation over batches
- Automatic synchronization between multiple devices

You can use TorchMetrics in any PyTorch model, or within [PyTorch Lightning](#) to enjoy additional features:

- This means that your data will always be placed on the same device as your metrics.
- Native support for logging metrics in Lightning to reduce even more boilerplate.

### 1.1.1 Install

You can install TorchMetrics using pip or conda:

```
# Python Package Index (PyPI)
pip install torchmetrics
# Conda
conda install -c conda-forge torchmetrics
```

Eventually if there is a missing PyTorch wheel for your OS or Python version you can simply compile PyTorch from source:

```
# Optional if you do not need compile GPU support
export USE_CUDA=0 # just to keep it simple
# you can install the latest state from master
pip install git+https://github.com/pytorch/pytorch.git
# OR set a particular PyTorch release
pip install git+https://github.com/pytorch/pytorch.git@<release-tag>
# and finalize with installing TorchMetrics
pip install torchmetrics
```

### 1.1.2 Using TorchMetrics

#### Functional metrics

Similar to `torch.nn`, most metrics have both a class-based and a functional version. The functional versions implement the basic operations required for computing each metric. They are simple python functions that as input take `torch.tensors` and return the corresponding metric as a `torch.tensor`. The code-snippet below shows a simple example for calculating the accuracy using the functional interface:

```
import torch
# import our library
import torchmetrics

# simulate a classification problem
preds = torch.randn(10, 5).softmax(dim=-1)
target = torch.randint(5, (10,))

acc = torchmetrics.functional.accuracy(preds, target, task="multiclass", num_classes=5)
```

#### Module metrics

Nearly all functional metrics have a corresponding class-based metric that calls it a functional counterpart underneath. The class-based metrics are characterized by having one or more internal metrics states (similar to the parameters of the PyTorch module) that allow them to offer additional functionalities:

- Accumulation of multiple batches
- Automatic synchronization between multiple devices
- Metric arithmetic

The code below shows how to use the class-based interface:

```

import torch
# import our library
import torchmetrics

# initialize metric
metric = torchmetrics.classification.Accuracy(task="multiclass", num_classes=5)

n_batches = 10
for i in range(n_batches):
    # simulate a classification problem
    preds = torch.randn(10, 5).softmax(dim=-1)
    target = torch.randint(5, (10,))
    # metric on current batch
    acc = metric(preds, target)
    print(f"Accuracy on batch {i}: {acc}")

# metric on all batches using custom accumulation
acc = metric.compute()
print(f"Accuracy on all data: {acc}")

# Resetting internal state such that metric ready for new data
metric.reset()

```

### 1.1.3 Implementing your own metric

Implementing your own metric is as easy as subclassing a `torch.nn.Module`. Simply, subclass `Metric` and do the following:

1. Implement `__init__` where you call `self.add_state` for every internal state that is needed for the metrics computations
2. Implement update method, where all logic that is necessary for updating metric states go
3. Implement compute method, where the final metric computations happens

For practical examples and more info about implementing a metric, please see this [page](#).

## Development Environment

TorchMetrics provides a [Devcontainer](#) configuration for [Visual Studio Code](#) to use a [Docker container](#) as a pre-configured development environment. This avoids struggles setting up a development environment and makes them reproducible and consistent. Please follow the [installation instructions](#) and make yourself familiar with the [container tutorials](#) if you want to use them. In order to use GPUs, you can enable them within the `.devcontainer/devcontainer.json` file.

## 1.2 All TorchMetrics





## 1.3 Structure Overview

TorchMetrics is a Metrics API created for easy metric development and usage in PyTorch and PyTorch Lightning. It is rigorously tested for all edge cases and includes a growing list of common metric implementations.

The metrics API provides `update()`, `compute()`, `reset()` functions to the user. The metric *base class* inherits `torch.nn.Module` which allows us to call `metric(...)` directly. The `forward()` method of the base `Metric` class serves the dual purpose of calling `update()` on its input and simultaneously returning the value of the metric over the provided input.

These metrics work with DDP in PyTorch and PyTorch Lightning by default. When `.compute()` is called in distributed mode, the internal state of each metric is synced and reduced across each process, so that the logic present in `.compute()` is applied to state information from all processes.

This metrics API is independent of PyTorch Lightning. Metrics can directly be used in PyTorch as shown in the example:

```
from torchmetrics.classification import BinaryAccuracy

train_accuracy = BinaryAccuracy()
valid_accuracy = BinaryAccuracy()

for epoch in range(epochs):
    for x, y in train_data:
        y_hat = model(x)

        # training step accuracy
        batch_acc = train_accuracy(y_hat, y)
        print(f"Accuracy of batch{i} is {batch_acc}")

    for x, y in valid_data:
        y_hat = model(x)
        valid_accuracy.update(y_hat, y)

    # total accuracy over all training batches
    total_train_accuracy = train_accuracy.compute()

    # total accuracy over all validation batches
    total_valid_accuracy = valid_accuracy.compute()

    print(f"Training acc for epoch {epoch}: {total_train_accuracy}")
    print(f"Validation acc for epoch {epoch}: {total_valid_accuracy}")

    # Reset metric states after each epoch
    train_accuracy.reset()
    valid_accuracy.reset()
```

---

**Note:** Metrics contain internal states that keep track of the data seen so far. Do not mix metric states across training, validation and testing. It is highly recommended to re-initialize the metric per mode as shown in the examples above.

---

---

**Note:** Metric states are **not** added to the models `state_dict` by default. To change this, after initializing the metric, the method `.persistent(mode)` can be used to enable (`mode=True`) or disable (`mode=False`) this behaviour.

---

---

**Note:** Due to specialized logic around metric states, we in general do **not** recommend that metrics are initialized inside other metrics (nested metrics), as this can lead to weird behaviour. Instead consider subclassing a metric or use `torchmetrics.MetricCollection`.

---



### 1.3.1 Metrics and devices

Metrics are simple subclasses of `Module` and their metric states behave similar to buffers and parameters of modules. This means that metrics states should be moved to the same device as the input of the metric:

```
from torchmetrics.classification import BinaryAccuracy

target = torch.tensor([1, 1, 0, 0], device=torch.device("cuda", 0))
preds = torch.tensor([0, 1, 0, 0], device=torch.device("cuda", 0))

# Metric states are always initialized on cpu, and needs to be moved to
# the correct device
confmat = BinaryAccuracy().to(torch.device("cuda", 0))
out = confmat(preds, target)
print(out.device) # cuda:0
```

However, when **properly defined** inside a `Module` or `LightningModule` the metric will be automatically moved to the same device as the module when using `.to(device)`. Being **properly defined** means that the metric is correctly identified as a child module of the model (check `.children()` attribute of the model). Therefore, metrics cannot be placed in native python list and dict, as they will not be correctly identified as child modules. Instead of list use `ModuleList` and instead of dict use `ModuleDict`. Furthermore, when working with multiple metrics the native `MetricCollection` module can also be used to wrap multiple metrics.

```
from torchmetrics import MetricCollection
from torchmetrics.classification import BinaryAccuracy

class MyModule(torch.nn.Module):
    def __init__(self):
        ...
        # valid ways metrics will be identified as child modules
        self.metric1 = BinaryAccuracy()
        self.metric2 = nn.ModuleList(BinaryAccuracy())
        self.metric3 = nn.ModuleDict({'accuracy': BinaryAccuracy()})
        self.metric4 = MetricCollection([BinaryAccuracy()]) # torchmetrics build-in
        ↪collection class

    def forward(self, batch):
        data, target = batch
        preds = self(data)
        ...
        val1 = self.metric1(preds, target)
        val2 = self.metric2[0](preds, target)
        val3 = self.metric3['accuracy'](preds, target)
        val4 = self.metric4(preds, target)
```

You can always check which device the metric is located on using the `.device` property.

## Metrics in Distributed Data Parallel (DDP) mode

When using metrics in [Distributed Data Parallel \(DDP\)](#) mode, one should be aware that DDP will add additional samples to your dataset if the size of your dataset is not equally divisible by `batch_size * num_processors`. The added samples will always be replicates of datapoints already in your dataset. This is done to secure an equal load for all processes. However, this has the consequence that the calculated metric value will be slightly biased towards those replicated samples, leading to a wrong result.

During training and/or validation this may not be important, however it is highly recommended when evaluating the test dataset to only run on a single gpu or use a [join](#) context in conjunction with DDP to prevent this behaviour.

### 1.3.2 Metrics and 16-bit precision

Most metrics in our collection can be used with 16-bit precision (`torch.half`) tensors. However, we have found the following limitations:

- In general pytorch had better support for 16-bit precision much earlier on GPU than CPU. Therefore, we recommend that anyone that want to use metrics with half precision on CPU, upgrade to atleast pytorch v1.6 where support for operations such as addition, subtraction, multiplication ect. was added.
- Some metrics does not work at all in half precision on CPU. We have explicitly stated this in their docstring, but they are also listed below:
  - *Peak Signal-to-Noise Ratio (PSNR)*
  - *Structural Similarity Index Measure (SSIM)*
  - *KL Divergence*

You can always check the precision/dtype of the metric by checking the `.dtype` property.

### 1.3.3 Metric Arithmetics

Metrics support most of python built-in operators for arithmetic, logic and bitwise operations.

For example for a metric that should return the sum of two different metrics, implementing a new metric is an overhead that is not necessary. It can now be done with:

```
first_metric = MyFirstMetric()
second_metric = MySecondMetric()

new_metric = first_metric + second_metric
```

`new_metric.update(*args, **kwargs)` now calls `update` of `first_metric` and `second_metric`. It forwards all positional arguments but forwards only the keyword arguments that are available in respective metric's update declaration. Similarly `new_metric.compute()` now calls `compute` of `first_metric` and `second_metric` and adds the results up. It is important to note that all implemented operations always returns a new metric object. This means that the line `first_metric == second_metric` will not return a bool indicating if `first_metric` and `second_metric` is the same metric, but will return a new metric that checks if the `first_metric.compute() == second_metric.compute()`.

This pattern is implemented for the following operators (with `a` being metrics and `b` being metrics, tensors, integer or floats):

- Addition (`a + b`)
- Bitwise AND (`a & b`)

- Equality (`a == b`)
- Floordivision (`a // b`)
- Greater Equal (`a >= b`)
- Greater (`a > b`)
- Less Equal (`a <= b`)
- Less (`a < b`)
- Matrix Multiplication (`a @ b`)
- Modulo (`a % b`)
- Multiplication (`a * b`)
- Inequality (`a != b`)
- Bitwise OR (`a | b`)
- Power (`a ** b`)
- Subtraction (`a - b`)
- True Division (`a / b`)
- Bitwise XOR (`a ^ b`)
- Absolute Value (`abs(a)`)
- Inversion (`~a`)
- Negative Value (`neg(a)`)
- Positive Value (`pos(a)`)
- Indexing (`a[0]`)

---

**Note:** Some of these operations are only fully supported from Pytorch v1.4 and onwards, explicitly we found: `add`, `mul`, `rmatmul`, `rsub`, `rmod`

---

### 1.3.4 MetricCollection

In many cases it is beneficial to evaluate the model output by multiple metrics. In this case the `MetricCollection` class may come in handy. It accepts a sequence of metrics and wraps these into a single callable metric class, with the same interface as any other metric.

Example:

```
from torchmetrics import MetricCollection
from torchmetrics.classification import MulticlassAccuracy, MulticlassPrecision,
↳ MulticlassRecall
target = torch.tensor([0, 2, 0, 2, 0, 1, 0, 2])
preds = torch.tensor([2, 1, 2, 0, 1, 2, 2, 2])
metric_collection = MetricCollection([
    MulticlassAccuracy(num_classes=3, average="micro"),
    MulticlassPrecision(num_classes=3, average="macro"),
    MulticlassRecall(num_classes=3, average="macro")
])
```

(continues on next page)

(continued from previous page)

```
])
print(metric_collection(preds, target))
```

```
{'MulticlassAccuracy': tensor(0.1250),
 'MulticlassPrecision': tensor(0.0667),
 'MulticlassRecall': tensor(0.1111)}
```

Similarly it can also reduce the amount of code required to log multiple metrics inside your LightningModule. In most cases we just have to replace `self.log` with `self.log_dict`.

```
from torchmetrics import MetricCollection
from torchmetrics.classification import MulticlassAccuracy, MulticlassPrecision,
↪MulticlassRecall

class MyModule(LightningModule):
    def __init__(self, num_classes: int):
        super().__init__()
        metrics = MetricCollection([
            MulticlassAccuracy(num_classes), MulticlassPrecision(num_classes),
↪MulticlassRecall(num_classes)
        ])
        self.train_metrics = metrics.clone(prefix='train_')
        self.valid_metrics = metrics.clone(prefix='val_')

    def training_step(self, batch, batch_idx):
        logits = self(x)
        # ...
        output = self.train_metrics(logits, y)
        # use log_dict instead of log
        # metrics are logged with keys: train_Accuracy, train_Precision and train_Recall
        self.log_dict(output)

    def validation_step(self, batch, batch_idx):
        logits = self(x)
        # ...
        self.valid_metrics.update(logits, y)

    def validation_epoch_end(self, outputs):
        # use log_dict instead of log
        # metrics are logged with keys: val_Accuracy, val_Precision and val_Recall
        output = self.valid_metrics.compute()
        self.log_dict(output)
        # remember to reset metrics at the end of the epoch
        self.valid_metrics.reset()
```

**Note:** *MetricCollection* as default assumes that all the metrics in the collection have the same call signature. If this is not the case, input that should be given to different metrics can given as keyword arguments to the collection.

An additional advantage of using the *MetricCollection* object is that it will automatically try to reduce the computations needed by finding groups of metrics that share the same underlying metric state. If such a group of metrics is found only one of them is actually updated and the updated state will be broadcasted to the rest of the metrics within the

group. In the example above, this will lead to a 2x-3x lower computational cost compared to disabling this feature in the case of the validation metrics where only `update` is called (this feature does not work in combination with `forward`). However, this speedup comes with a fixed cost upfront, where the state-groups have to be determined after the first update. In case the groups are known beforehand, these can also be set manually to avoid this extra cost of the dynamic search. See the `compute_groups` argument in the class docs below for more information on this topic.

```
class torchmetrics.MetricCollection(metrics, *additional_metrics, prefix=None, postfix=None,
                                     compute_groups=True)
```

MetricCollection class can be used to chain metrics that have the same call pattern into one single class.

#### Parameters

- **metrics** (`Union[Metric, Sequence[Metric], Dict[str, Metric]]`) – One of the following
  - list or tuple (sequence): if metrics are passed in as a list or tuple, will use the metrics class name as key for output dict. Therefore, two metrics of the same class cannot be chained this way.
  - arguments: similar to passing in as a list, metrics passed in as arguments will use their metric class name as key for the output dict.
  - dict: if metrics are passed in as a dict, will use each key in the dict as key for output dict. Use this format if you want to chain together multiple of the same metric with different parameters. Note that the keys in the output dict will be sorted alphabetically.
- **prefix** (`Optional[str]`) – a string to append in front of the keys of the output dict
- **postfix** (`Optional[str]`) – a string to append after the keys of the output dict
- **compute\_groups** (`Union[bool, List[List[str]]]`) – By default the MetricCollection will try to reduce the computations needed for the metrics in the collection by checking if they belong to the same **compute group**. All metrics in a compute group share the same metric state and are therefore only different in their compute step e.g. accuracy, precision and recall can all be computed from the true positives/negatives and false positives/negatives. By default, this argument is `True` which enables this feature. Set this argument to `False` for disabling this behaviour. Can also be set to a list of lists of metrics for setting the compute groups yourself.

---

**Note:** The compute groups feature can significantly speedup the calculation of metrics under the right conditions. First, the feature is only available when calling the `update` method and not when calling `forward` method due to the internal logic of `forward` preventing this. Secondly, since we compute groups share metric states by reference, calling `.items()`, `.values()` etc. on the metric collection will break this reference and a copy of states are instead returned in this case (reference will be reestablished on the next call to `update`).

---



---

**Note:** Metric collections can be nested at initialization (see last example) but the output of the collection will still be a single flatten dictionary combining the prefix and postfix arguments from the nested collection.

---

#### Raises

- **ValueError** – If one of the elements of `metrics` is not an instance of `pl.metrics.Metric`.
- **ValueError** – If two elements in `metrics` have the same name.
- **ValueError** – If `metrics` is not a list, tuple or a dict.

- **ValueError** – If `metrics` is dict and `additional_metrics` are passed in.
- **ValueError** – If `prefix` is set and it is not a string.
- **ValueError** – If `postfix` is set and it is not a string.

**Example::**

In the most basic case, the metrics can be passed in as a list or tuple. The keys of the output dict will be the same as the class name of the metric:

```
>>> from torch import tensor
>>> from pprint import pprint
>>> from torchmetrics import MetricCollection
>>> from torchmetrics.regression import MeanSquaredError
>>> from torchmetrics.classification import MulticlassAccuracy,
↳ MulticlassPrecision, MulticlassRecall
>>> target = tensor([0, 2, 0, 2, 0, 1, 0, 2])
>>> preds = tensor([2, 1, 2, 0, 1, 2, 2, 2])
>>> metrics = MetricCollection([MulticlassAccuracy(num_classes=3, average='micro
↳ '),
...                               MulticlassPrecision(num_classes=3, average=
↳ 'macro'),
...                               MulticlassRecall(num_classes=3, average='macro
↳ ')])
>>> metrics(preds, target)
{'MulticlassAccuracy': tensor(0.1250),
 'MulticlassPrecision': tensor(0.0667),
 'MulticlassRecall': tensor(0.1111)}
```

**Example::**

Alternatively, metrics can be passed in as arguments. The keys of the output dict will be the same as the class name of the metric:

```
>>> metrics = MetricCollection(MulticlassAccuracy(num_classes=3, average='micro
↳ '),
...                               MulticlassPrecision(num_classes=3, average='macro
↳ '),
...                               MulticlassRecall(num_classes=3, average='macro'
↳ ))
>>> metrics(preds, target)
{'MulticlassAccuracy': tensor(0.1250),
 'MulticlassPrecision': tensor(0.0667),
 'MulticlassRecall': tensor(0.1111)}
```

**Example::**

If multiple of the same metric class (with different parameters) should be chained together, metrics can be passed in as a dict and the output dict will have the same keys as the input dict:

```
>>> metrics = MetricCollection({'micro_recall': MulticlassRecall(num_classes=3,
↳ average='micro'),
...                               'macro_recall': MulticlassRecall(num_classes=3,
↳ average='macro')})
>>> same_metric = metrics.clone()
>>> pprint(metrics(preds, target))
{'macro_recall': tensor(0.1111), 'micro_recall': tensor(0.1250)}
```

(continues on next page)

(continued from previous page)

```
>>> pprint(same_metric(preds, target))
{'macro_recall': tensor(0.1111), 'micro_recall': tensor(0.1250)}
```

**Example::**

Metric collections can also be nested up to a single time. The output of the collection will still be a single dict with the prefix and postfix arguments from the nested collection:

```
>>> metrics = MetricCollection([
...     MetricCollection([
...         MulticlassAccuracy(num_classes=3, average='macro'),
...         MulticlassPrecision(num_classes=3, average='macro')
...     ], postfix='_macro'),
...     MetricCollection([
...         MulticlassAccuracy(num_classes=3, average='micro'),
...         MulticlassPrecision(num_classes=3, average='micro')
...     ], postfix='_micro'),
... ], prefix='valmetrics/')
>>> pprint(metrics(preds, target))
{'valmetrics/MulticlassAccuracy_macro': tensor(0.1111),
 'valmetrics/MulticlassAccuracy_micro': tensor(0.1250),
 'valmetrics/MulticlassPrecision_macro': tensor(0.0667),
 'valmetrics/MulticlassPrecision_micro': tensor(0.1250)}
```

**Example::**

The `compute_groups` argument allow you to specify which metrics should share metric state. By default, this will automatically be derived but can also be set manually.

```
>>> metrics = MetricCollection(
...     MulticlassRecall(num_classes=3, average='macro'),
...     MulticlassPrecision(num_classes=3, average='macro'),
...     MeanSquaredError(),
...     compute_groups=[['MulticlassRecall', 'MulticlassPrecision'], [
...         ↪ 'MeanSquaredError']]
... )
>>> metrics.update(preds, target)
>>> pprint(metrics.compute())
{'MeanSquaredError': tensor(2.3750), 'MulticlassPrecision': tensor(0.0667),
 ↪ 'MulticlassRecall': tensor(0.1111)}
>>> pprint(metrics.compute_groups)
{0: ['MulticlassRecall', 'MulticlassPrecision'], 1: ['MeanSquaredError']}
```

**add\_metrics**(*metrics*, \**additional\_metrics*)

Add new metrics to Metric Collection.

**Return type**

`None`

**clone**(*prefix=None*, *postfix=None*)

Make a copy of the metric collection.

**Parameters**

- **prefix** (`Optional[str]`) – a string to append in front of the metric keys
- **postfix** (`Optional[str]`) – a string to append after the keys of the output dict.

**Return type**`MetricCollection`**items**(*keep\_base=False, copy\_state=True*)

Return an iterable of the ModuleDict key/value pairs.

**Parameters**

- **keep\_base** (`bool`) – Whether to add prefix/postfix on the collection.
- **copy\_state** (`bool`) – If metric states should be copied between metrics in the same compute group or just passed by reference

**Return type**`Iterable[Tuple[str, Metric]]`**keys**(*keep\_base=False*)

Return an iterable of the ModuleDict key.

**Parameters**

**keep\_base** (`bool`) – Whether to add prefix/postfix on the items collection.

**Return type**`Iterable[Hashable]`**persistent**(*mode=True*)

Change if metric states should be saved to its state\_dict after initialization.

**Return type**`None`**plot**(*val=None, ax=None, together=False*)

Plot a single or multiple values from the metric.

The plot method has two modes of operation. If argument *together* is set to *False* (default), the *.plot* method of each metric will be called individually and the result will be list of figures. If *together* is set to *True*, the values of all metrics will instead be plotted in the same figure.

**Parameters**

- **val** (`Union[Dict, Sequence[Dict], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Union[Axes, Sequence[Axes], None]`) – Either a single instance of matplotlib axis object or an sequence of matplotlib axis objects. If provided, will add the plots to the provided axis objects. If not provided, will create a new. If argument *together* is set to *True*, a single object is expected. If *together* is set to *False*, the number of axis objects needs to be the same length as the number of metrics in the collection.
- **together** (`bool`) – If *True*, will plot all metrics in the same axis. If *False*, will plot each metric in a separate

**Return type**`Sequence[Tuple[Figure, Union[Axes, ndarray]]]`**Returns**

Either instal tupel of Figure and Axes object or an sequence of tuples with Figure and Axes object for each metric in the collection.

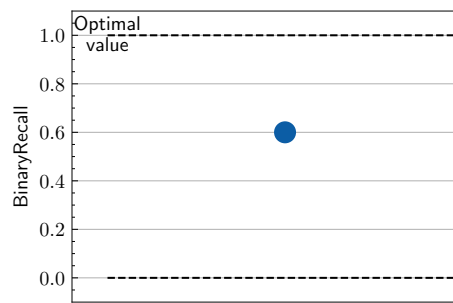
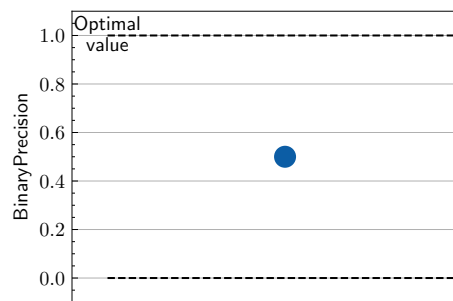
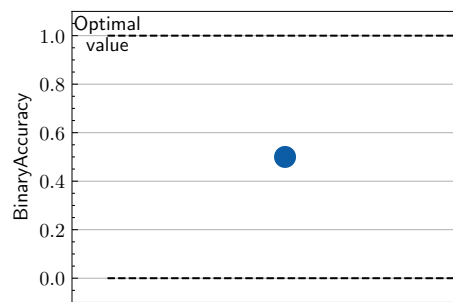
**Raises**

- **ModuleNotFoundError** – If *matplotlib* is not installed



- **ValueError** – If *together* is not an bool
- **ValueError** – If *ax* is not an instance of matplotlib axis object or a sequence of matplotlib axis objects

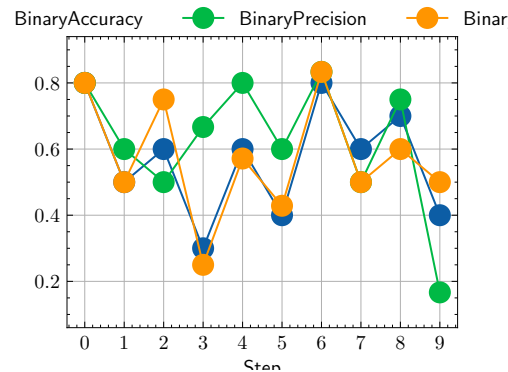
```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics import MetricCollection
>>> from torchmetrics.classification import BinaryAccuracy, BinaryPrecision, \
↳ BinaryRecall
>>> metrics = MetricCollection([BinaryAccuracy(), BinaryPrecision(), \
↳ BinaryRecall()])
>>> metrics.update(torch.rand(10), torch.randint(2, (10,)))
>>> fig_ax_ = metrics.plot()
```



```

>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics import MetricCollection
>>> from torchmetrics.classification import BinaryAccuracy, BinaryPrecision, BinaryRecall
>>> metrics = MetricCollection([BinaryAccuracy(), BinaryPrecision(), BinaryRecall()])
>>> values = []
>>> for _ in range(10):
...     values.append(metrics(torch.rand(10), torch.randint(2, (10,))))
>>> fig_, ax_ = metrics.plot(values, together=True)

```



**reset()**

Call reset for each metric sequentially.

**Return type**

`None`

**set\_dtype(*dst\_type*)**

Transfer all metric state to specific dtype. Special version of standard *type* method.

**Parameters**

**dst\_type** (*type* or *string*) – the desired type.

**Return type**

`MetricCollection`

**values(*copy\_state=True*)**

Return an iterable of the ModuleDict values.

**Parameters**

**copy\_state** (*bool*) – If metric states should be copied between metrics in the same compute group or just passed by reference

**Return type**

`Iterable[Metric]`

**property compute\_groups: Dict[int, List[str]]**

Return a dict with the current compute groups in the collection.

### 1.3.5 Module vs Functional Metrics

The functional metrics follow the simple paradigm input in, output out. This means they don't provide any advanced mechanisms for syncing across DDP nodes or aggregation over batches. They simply compute the metric value based on the given inputs.

Also, the integration within other parts of PyTorch Lightning will never be as tight as with the Module-based interface. If you look for just computing the values, the functional metrics are the way to go. However, if you are looking for the best integration and user experience, please consider also using the Module interface.

### 1.3.6 Metrics and differentiability

Metrics support backpropagation, if all computations involved in the metric calculation are differentiable. All modular metric classes have the property `is_differentiable` that determines if a metric is differentiable or not.

However, note that the cached state is detached from the computational graph and cannot be back-propagated. Not doing this would mean storing the computational graph for each update call, which can lead to out-of-memory errors. In practise this means that:

```
MyMetric.is_differentiable  # returns True if metric is differentiable
metric = MyMetric()
val = metric(pred, target)  # this value can be back-propagated
val = metric.compute()      # this value cannot be back-propagated
```

A functional metric is differentiable if its corresponding modular metric is differentiable.

### 1.3.7 Metrics and hyperparameter optimization

If you want to directly optimize a metric it needs to support backpropagation (see section above). However, if you are just interested in using a metric for hyperparameter tuning and are not sure if the metric should be maximized or minimized, all modular metric classes have the `higher_is_better` property that can be used to determine this:

```
# returns True because accuracy is optimal when it is maximized
torchmetrics.classification.Accuracy.higher_is_better

# returns False because the mean squared error is optimal when it is minimized
torchmetrics.MeanSquaredError.higher_is_better
```

### 1.3.8 Advanced metric settings

The following is a list of additional arguments that can be given to any metric class (in the `**kwargs` argument) that will alter how metric states are stored and synced.

If you are running metrics on GPU and are encountering that you are running out of GPU VRAM then the following argument can help:

- `compute_on_cpu`: will automatically move the metric states to cpu after calling `update`, making sure that GPU memory is not filling up. The consequence will be that the `compute` method will be called on CPU instead of GPU. Only applies to metric states that are lists.
- `compute_with_cache`: This argument indicates if the result after calling the `compute` method should be cached. By default this is `True` meaning that repeated calls to `compute` (with no change to the metric state inbetween) does not recompute the metric but just returns the cache. By setting it to `False` the metric will be recomputed every time `compute` is called, but it can also help clean up a bit of memory.

If you are running in a distributed environment, TorchMetrics will automatically take care of the distributed synchronization for you. However, the following three keyword arguments can be given to any metric class for further control over the distributed aggregation:

- `sync_on_compute`: This argument is an `bool` that indicates if the metrics should automatically sync between devices whenever the `compute` method is called. By default this is `True`, but by setting this to `False` you can manually control when the synchronization happens.
- `dist_sync_on_step`: This argument is `bool` that indicates if the metric should synchronize between different devices every time `forward` is called. Setting this to `True` is in general not recommended as synchronization is an expensive operation to do after each batch.
- `process_group`: By default we synchronize across the *world* i.e. all processes being computed on. You can provide an `torch._C._distributed_c10d.ProcessGroup` in this argument to specify exactly what devices should be synchronized over.
- `dist_sync_fn`: By default we use `torch.distributed.all_gather()` to perform the synchronization between devices. Provide another callable function for this argument to perform custom distributed synchronization.

## 1.4 Plotting

---

**Note:** The visualization/plotting interface of Torchmetrics requires `matplotlib` to be installed. Install with either `pip install matplotlib` or `pip install 'torchmetrics[visual]'`. If the latter option is chosen the [Scienceplot](#) package is also installed and all plots in Torchmetrics will default to using that style.

---

Torchmetrics comes with build-in support for quick visualization of your metrics, by simply using the `.plot` method that all modular metrics implement. This method provides a consistent interface for basic plotting of all metrics.

```
metric = AnyMetricYouLike()
for _ in range(num_updates):
    metric.update(preds[i], target[i])
fig, ax = metric.plot()
```

`.plot` will always return two objects: `fig` is an instance of `Figure` which contains figure level attributes and `ax` is an instance of `Axes` that contains all the elements of the plot. These two objects allow to change attributes of the plot after it is created. For example, if you want to make the fontsize of the x-axis a bit bigger and give the figure a nice title and finally save it on the above example, it could be do like this:

```
ax.set_fontsize(fs=20)
fig.set_title("This is a nice plot")
fig.save_fig("my_awesome_plot.png")
```

If you want to include a Torchmetrics plot in a bigger figure that has subfigures and subaxes, all `.plot` methods support an optional `ax` argument where you can pass in the subaxes you want the plot to be inserted into:

```
# combine plotting of two metrics into one figure
fig, ax = plt.subplots(nrows=1, ncols=2)
metric1 = Metric1()
metric2 = Metric2()
for _ in range(num_updates):
    metric1.update(preds[i], target[i])
    metric2.update(preds[i], target[i])
```

(continues on next page)

(continued from previous page)

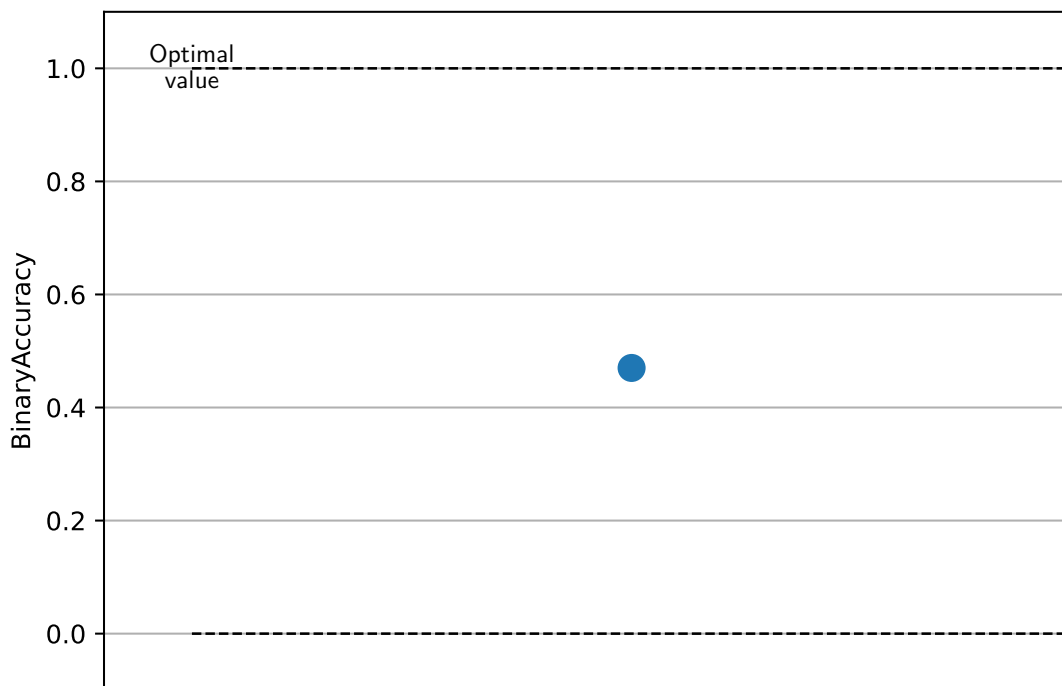
```
metric1.plot(ax=ax[0])
metric2.plot(ax=ax[1])
```

### 1.4.1 Plotting a single step

At the most basic level the `.plot` method can be used to plot the value from a single step. This can be done in two ways: \* Either `.plot` method is called with no input, and internally `metric.compute()` is called and that value is plotted \* `.plot` is called on a single returned value by the metric, for example from `metric.forward()`

In both cases it will generate a plot like this (Accuracy as an example):

```
metric = torchmetrics.Accuracy(task="binary")
for _ in range(num_updates):
    metric.update(torch.rand(10,), torch.randint(2, (10,)))
fig, ax = metric.plot()
```



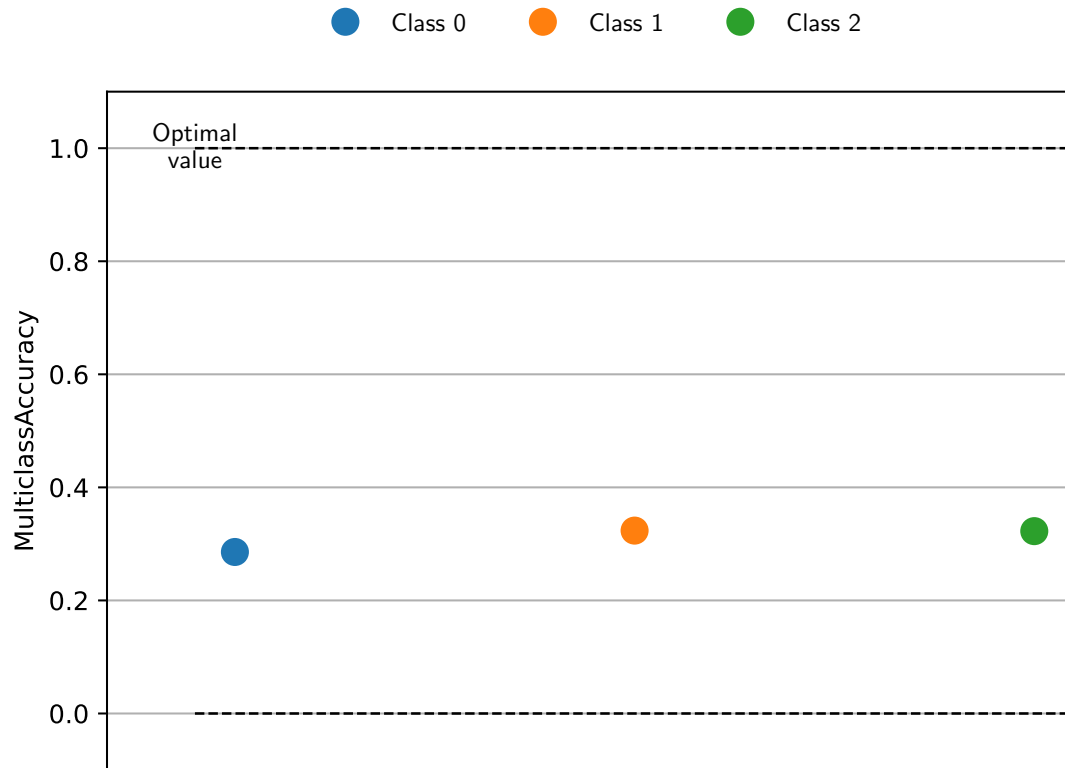
A single point plot is not that informative in itself, but if available we will try to include additional information such as the lower and upper bounds the particular metric can take and if the metric should be minimized or maximized to be optimal. This is true for all metrics that return a scalar tensor. Some metrics return multiple values (such as an tensor with multiple elements or a dict of scalar tensors), and in that case calling `.plot` will return a figure similar to this:

```
metric = torchmetrics.Accuracy(task="multiclass", num_classes=3, average=None)
for _ in range(num_updates):
```

(continues on next page)

(continued from previous page)

```
metric.update(torch.randint(3, (10,)), torch.randint(3, (10,)))
fig, ax = metric.plot()
```

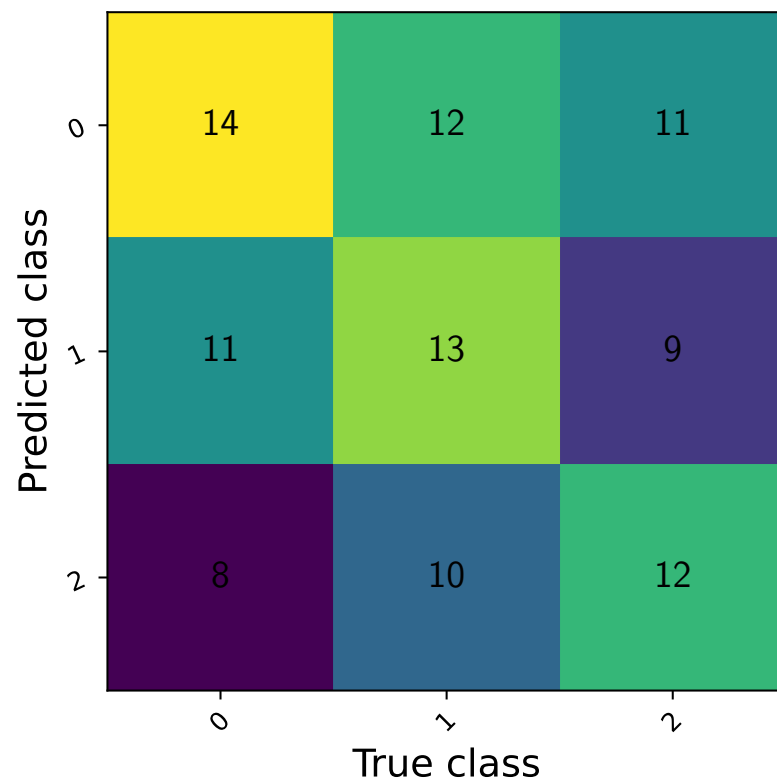


Here, each element is assumed to be an independent metric and plotted as its own point for comparing. The above is true for all metrics that return a scalar tensor, but if the metric returns a tensor with multiple elements then the `.plot` method will return a specialized plot for that particular metric. Take for example the `ConfusionMatrix` metric:

```
metric = torchmetrics.ConfusionMatrix(task="multiclass", num_classes=3)
for _ in range(num_updates):
    metric.update(torch.randint(3, (10,)), torch.randint(3, (10,)))
fig, ax = metric.plot()
```

If you prefer to use the functional interface of Torchmetrics, you can also plot the values returned by the functional. However, you would still need to initialize the corresponding metric class to get the information about the metric:

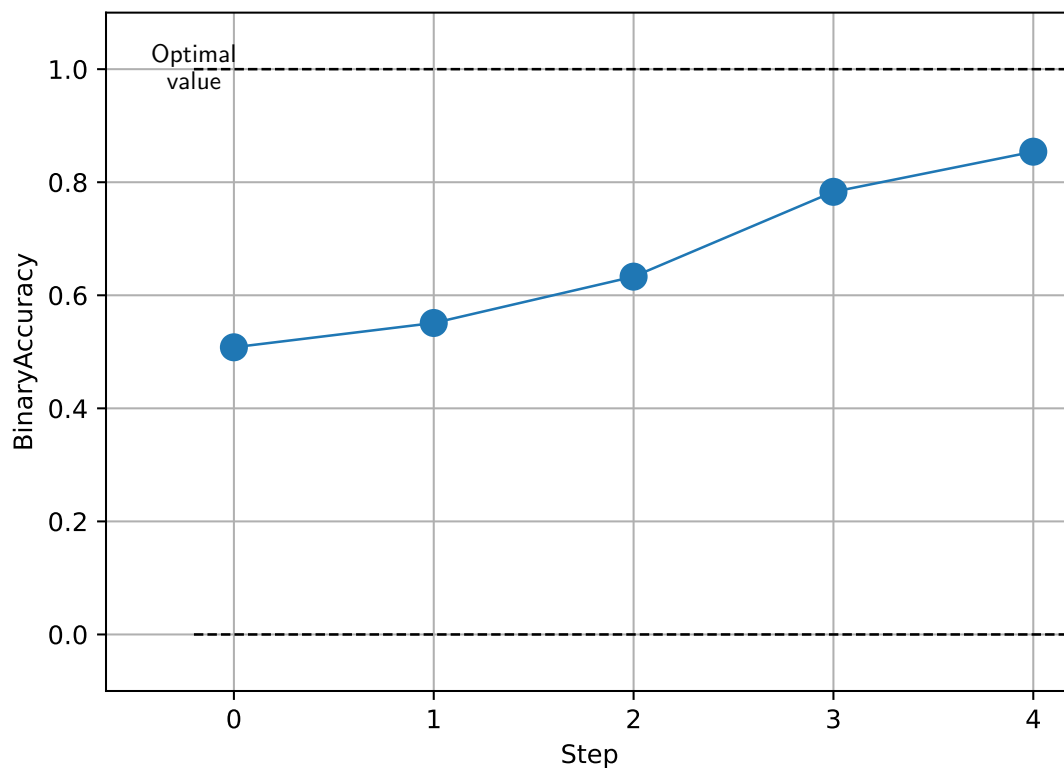
```
plot_class = torchmetrics.Accuracy(task="multiclass", num_classes=3)
value = torchmetrics.functional.accuracy(
    torch.randint(3, (10,)), torch.randint(3, (10,)), num_classes=3
)
fig, ax = plot_class.plot(value)
```



### 1.4.2 Plotting multi steps

In the above examples we have only plotted a single step/single value, but it is also possible to plot multiple steps from the same metric. This is often the case when training a machine learning model, where you are tracking one or multiple metrics that you want to plot as they are changing over time. This can be done by providing a sequence of outputs from any metric, computed using `metric.forward` or `metric.compute`. For example, if we want to plot the accuracy of a model over time, we could do it like this:

```
metric = torchmetrics.Accuracy(task="binary")
values = [ ]
for step in range(num_steps):
    for _ in range(num_updates):
        metric.update(preds(step), target(step))
    values.append(metric.compute()) # save value
    metric.reset()
fig, ax = metric.plot(values)
```



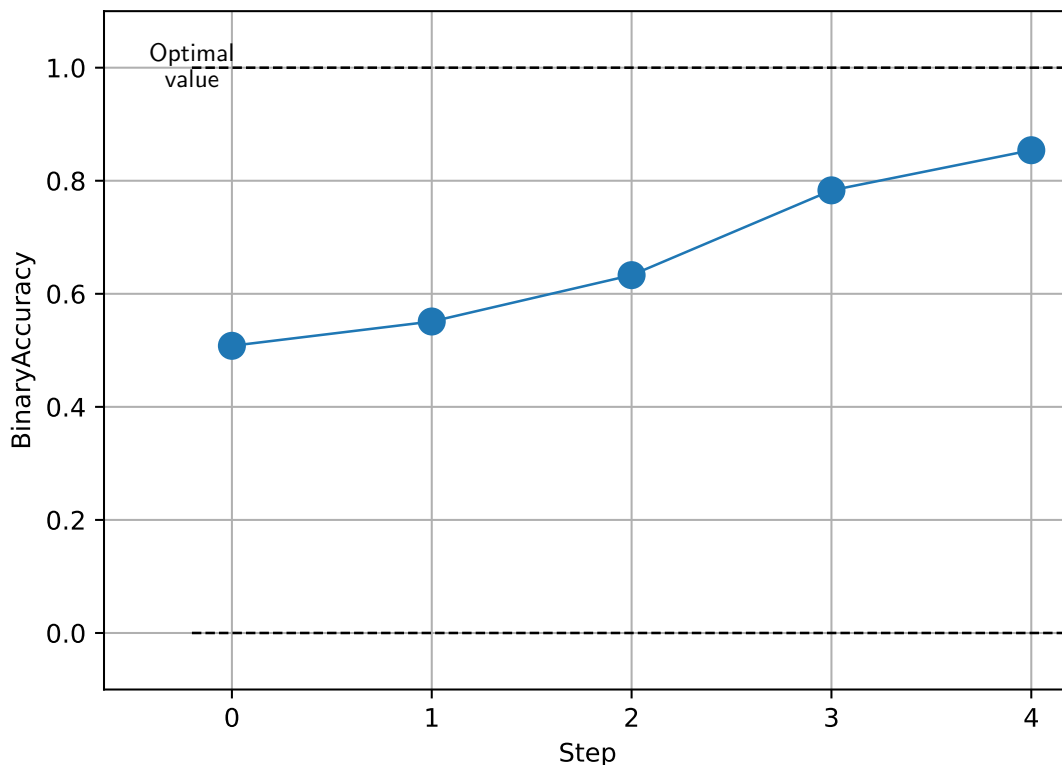
Do note that metrics that do not return simple scalar tensors, such as *ConfusionMatrix*, *ROC* that have specialized visualization does not support plotting multiple steps, out of the box and the user needs to manually plot the values for each step.



### 1.4.3 Plotting a collection of metrics

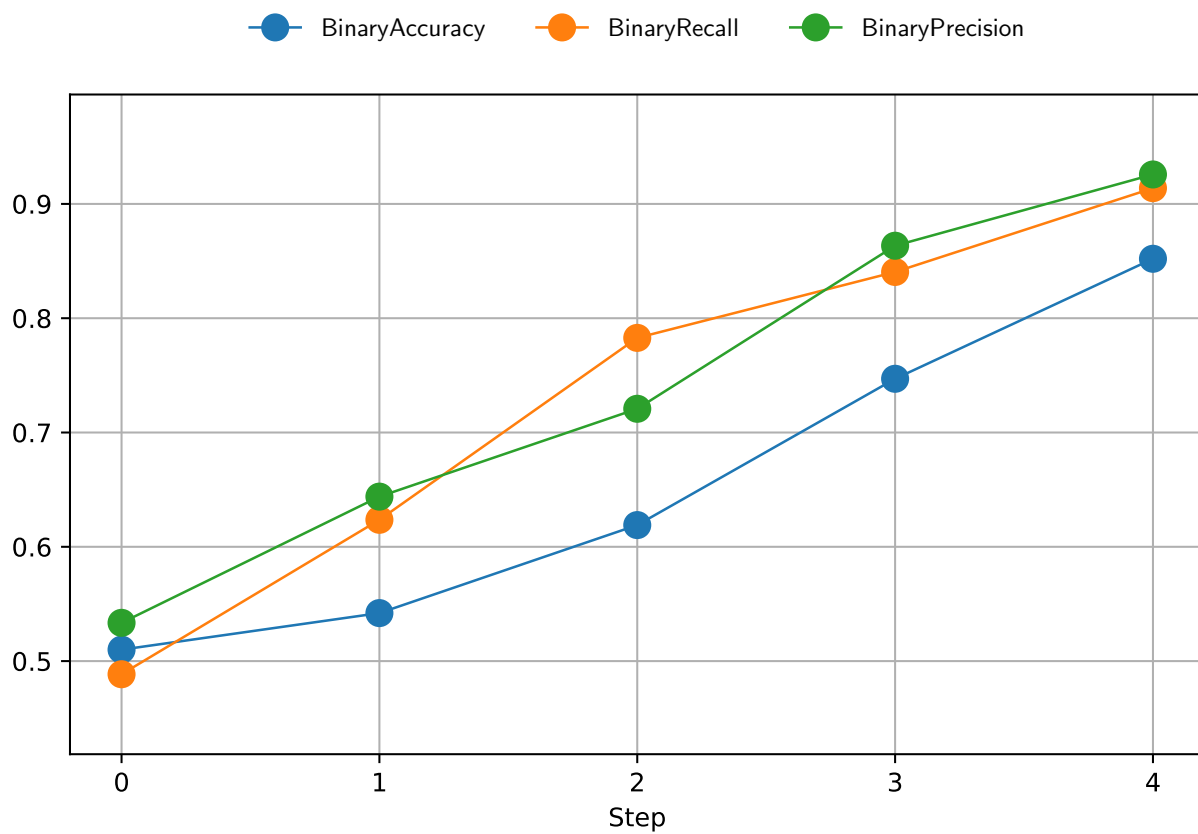
`MetricCollection` also supports `.plot` method and by default it works by just returning a collection of plots for all its members. Thus, instead of returning a single (fig, ax) pair, calling `.plot` method of `MetricCollection` will return a sequence of such pairs, one for each member in the collection. In the following example we are forming a collection of binary classification metrics and redirecting the output of `.plot` to different subplots:

```
collection = torchmetrics.MetricCollection(
    torchmetrics.Accuracy(task="binary"),
    torchmetrics.Recall(task="binary"),
    torchmetrics.Precision(task="binary"),
)
fig, ax = plt.subplots(nrows=1, ncols=3)
values = [ ]
for step in range(num_steps):
    for _ in range(num_updates):
        collection.update(preds(step), target(step))
    values.append(collection.compute())
    collection.reset()
collection.plot(val=values, ax=ax)
```



However, the `plot` method of `MetricCollection` also supports an additional argument called `together` that will automatically try to plot all the metrics in the collection together in the same plot (with appropriate labels). This is only possible if all the metrics in the collection return a scalar tensor.

```
collection = torchmetrics.MetricCollection(  
    torchmetrics.Accuracy(task="binary"),  
    torchmetrics.Recall(task="binary"),  
    torchmetrics.Precision(task="binary"),  
)  
values = [ ]  
fig, ax = plt.subplots(figsize=(6.8, 4.8))  
for step in range(num_steps):  
    for _ in range(num_updates):  
        collection.update(preds(step), target(step))  
        values.append(collection.compute())  
        collection.reset()  
collection.plot(val=values, together=True)
```



### 1.4.4 Advance example

In the following we are going to show how to use the `.plot` method to create a more advanced plot. We are going to combine the functionality of several metrics using `MetricCollection` and plot them together. In addition we are going to rely on `MetricTracker` to keep track of the metrics over multiple steps.

```
# Define collection that is a mix of metrics that return a scalar tensors and not
confmat = torchmetrics.ConfusionMatrix(task="binary")
roc = torchmetrics.ROC(task="binary")
collection = torchmetrics.MetricCollection(
    torchmetrics.Accuracy(task="binary"),
    torchmetrics.Recall(task="binary"),
    torchmetrics.Precision(task="binary"),
    confmat,
    roc,
)

# Define tracker over the collection to easy keep track of the metrics over multiple
↳ steps
tracker = torchmetrics.wrappers.MetricTracker(collection)

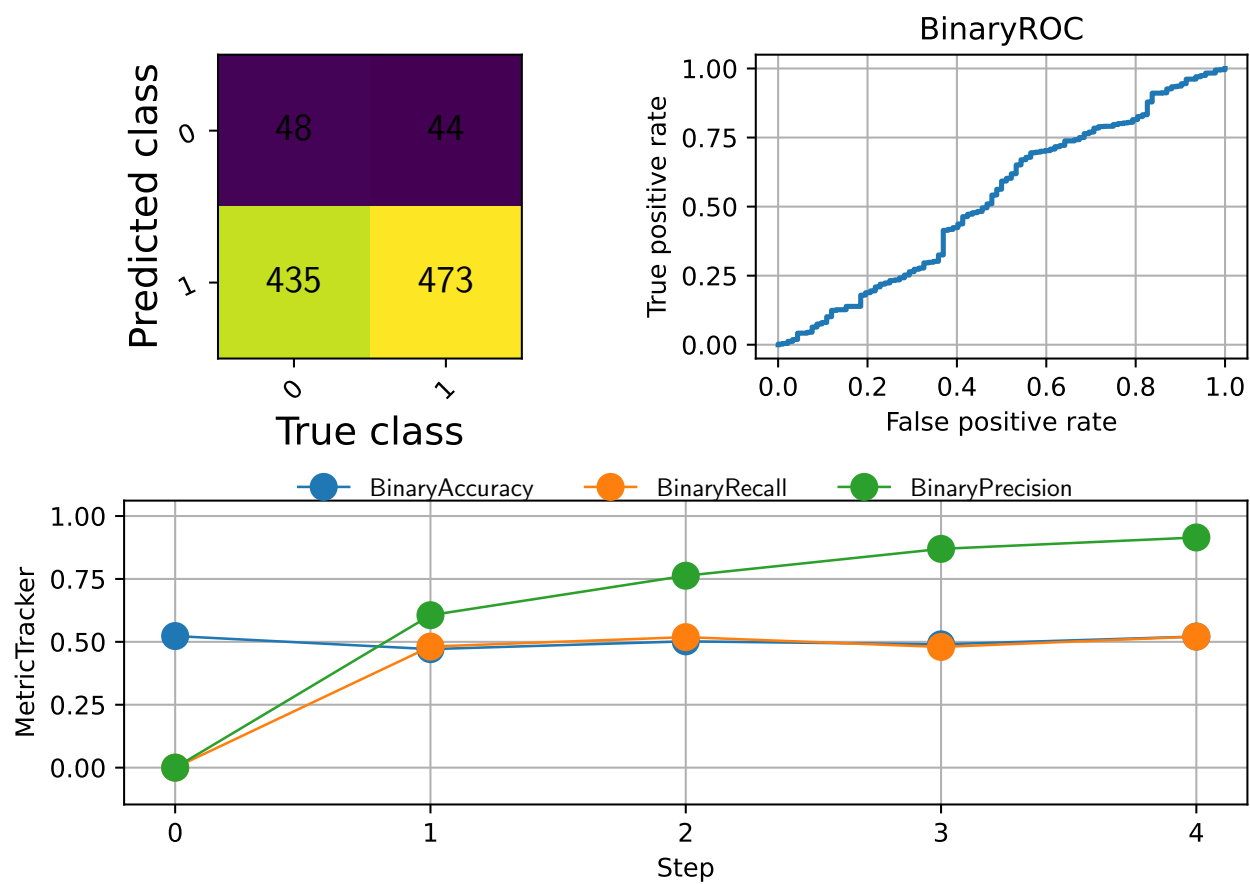
# Run "training" loop
for step in range(num_steps):
    tracker.increment()
    for _ in range(N):
        tracker.update(preds(step), target(step))

# Extract all metrics from all steps
all_results = tracker.compute_all()

# Constuct a single figure with appropriate layout for all metrics
fig = plt.figure(layout="constrained")
ax1 = plt.subplot(2, 2, 1)
ax2 = plt.subplot(2, 2, 2)
ax3 = plt.subplot(2, 2, (3, 4))

# ConfusionMatrix and ROC we just plot the last step, notice how we call the plot method
↳ of those metrics
confmat.plot(val=all_results[-1]['BinaryConfusionMatrix'], ax=ax1)
roc.plot(all_results[-1]["BinaryROC"], ax=ax2)

# For the remainig we plot the full history, but we need to extract the scalar values
↳ from the results
scalar_results = [
    {k: v for k, v in ar.items() if isinstance(v, torch.Tensor) and v.numel() == 1} for
    ↳ ar in all_results
    ar in all_results
]
tracker.plot(val=scalar_results, ax=ax3)
```



## 1.5 Implementing a Metric

To implement your own custom metric, subclass the base `Metric` class and implement the following methods:

- `__init__()`: Each state variable should be called using `self.add_state(...)`.
- `update()`: Any code needed to update the state given any inputs to the metric.
- `compute()`: Computes a final value from the state of the metric.

We provide the remaining interface, such as `reset()` that will make sure to correctly reset all metric states that have been added using `add_state`. You should therefore not implement `reset()` yourself. Additionally, adding metric states with `add_state` will make sure that states are correctly synchronized in distributed settings (DDP). To see how metric states are synchronized across distributed processes, refer to `add_state()` docs from the base `Metric` class.

Example implementation:

```
from torchmetrics import Metric

class MyAccuracy(Metric):
    def __init__(self):
        super().__init__()
        self.add_state("correct", default=torch.tensor(0), dist_reduce_fx="sum")
        self.add_state("total", default=torch.tensor(0), dist_reduce_fx="sum")

    def update(self, preds: Tensor, target: Tensor):
        preds, target = self._input_format(preds, target)
        assert preds.shape == target.shape

        self.correct += torch.sum(preds == target)
        self.total += target.numel()

    def compute(self):
        return self.correct.float() / self.total
```

Additionally you may want to set the class properties: `is_differentiable`, `higher_is_better` and `full_state_update`. Note that none of them are strictly required for the metric to work.

```
from torchmetrics import Metric

class MyMetric(Metric):
    # Set to True if the metric is differentiable else set to False
    is_differentiable: Optional[bool] = None

    # Set to True if the metric reaches it optimal value when the metric is maximized.
    # Set to False if it when the metric is minimized.
    higher_is_better: Optional[bool] = True

    # Set to True if the metric during 'update' requires access to the global metric
    # state for its calculations. If not, setting this to False indicates that all
    # batch states are independent and we will optimize the runtime of 'forward'
    full_state_update: bool = True
```

Finally, from `torchmetrics v1.0.0` onwards, we also support plotting of metrics through the `.plot` method. By default this method will raise `NotImplementedError` but can be implemented by the user to provide a custom plot for the metric.

For any metrics that returns a simple scalar tensor, or a dict of scalar tensors the internal `._plot` method can be used, that provides the common plotting functionality for most metrics in torchmetrics.

```
from torchmetrics import Metric
from torchmetrics.utilities.plot import _AX_TYPE, _PLOT_OUT_TYPE

class MyMetric(Metric):
    ...

    def plot(
        self, val: Optional[Union[Tensor, Sequence[Tensor]]] = None, ax: Optional[_AX_
↪TYPE] = None
    ) -> _PLOT_OUT_TYPE:
        return self._plot(val, ax)
```

If the metric returns a more complex output, a custom implementation of the `plot` method is required. For more details on the plotting API, see the [this page](#).

### 1.5.1 Internal implementation details

This section briefly describes how metrics work internally. We encourage looking at the source code for more info. Internally, TorchMetrics wraps the user defined `update()` and `compute()` method. We do this to automatically synchronize and reduce metric states across multiple devices. More precisely, calling `update()` does the following internally:

1. Clears computed cache.
2. Calls user-defined `update()`.

Similarly, calling `compute()` does the following internally:

1. Syncs metric states between processes.
2. Reduce gathered metric states.
3. Calls the user defined `compute()` method on the gathered metric states.
4. Cache computed result.

From a user's standpoint this has one important side-effect: computed results are cached. This means that no matter how many times `compute` is called after one and another, it will continue to return the same result. The cache is first emptied on the next call to `update`.

`forward` serves the dual purpose of both returning the metric on the current data and updating the internal metric state for accumulating over multiple batches. The `forward()` method achieves this by combining calls to `update`, `compute` and `reset`. Depending on the class property `full_state_update`, `forward` can behave in two ways:

1. If `full_state_update` is `True` it indicates that the metric during `update` requires access to the full metric state and we therefore need to do two calls to `update` to secure that the metric is calculated correctly
  1. Calls `update()` to update the global metric state (for accumulation over multiple batches)
  2. Caches the global state.
  3. Calls `reset()` to clear global metric state.
  4. Calls `update()` to update local metric state.
  5. Calls `compute()` to calculate metric for current batch.
  6. Restores the global state.

2. If `full_state_update` is `False` (default) the metric state of one batch is completely independent of the state of other batches, which means that we only need to call `update` once.
  1. Caches the global state.
  2. Calls `reset` the metric to its default state
  3. Calls `update` to update the state with local batch statistics
  4. Calls `compute` to calculate the metric for the current batch
  5. Reduce the global state and batch state into a single state that becomes the new global state

If implementing your own metric, we recommend trying out the metric with `full_state_update` class property set to both `True` and `False`. If the results are equal, then setting it to `False` will usually give the best performance.

---

```
class torchmetrics.Metric(**kwargs)
```

Base class for all metrics present in the Metrics API.

This class is inherited by all metrics and implements the following functionality: 1. Handles the transfer of metric states to correct device 2. Handles the synchronization of metric states across processes

The three core methods of the base class are `* add_state()` `* forward()` `* reset()`

which should almost never be overwritten by child classes. Instead, the following methods should be overwritten `* update()` `* compute()`

#### Parameters

**kwargs** (*Any*) – additional keyword arguments, see [Advanced metric settings](#) for more info.

- `compute_on_cpu`: If metric state should be stored on CPU during computations. Only works for list states.
- `dist_sync_on_step`: If metric state should synchronize on `forward()`. Default is `False`
- `process_group`: The process group on which the synchronization is called. Default is the world.
- `dist_sync_fn`: Function that performs the allgather option on the metric state. Default is a custom implementation that calls `torch.distributed.all_gather` internally.
- `distributed_available_fn`: Function that checks if the distributed backend is available. Defaults to a check of `torch.distributed.is_available()` and `torch.distributed.is_initialized()`.
- `sync_on_compute`: If metric state should synchronize when `compute` is called. Default is `True`
- `compute_with_cache`: If results from `compute` should be cached. Default is `False`

```
add_state(name, default, dist_reduce_fx=None, persistent=False)
```

Add metric state variable. Only used by subclasses.

Metric state variables are either `:class:`~torch.Tensor` or an empty list, which can be appended to by the metric. Each state variable must have a unique name associated with it. State variables are accessible as attributes of the metric i.e, if name is "my\_state" then its value can be accessed from an instance `metric` as `metric.my_state`. Metric states behave like buffers and parameters of `Module` as they are also updated when `.to()` is called. Unlike parameters and buffers, metric states are not by default saved in the `modules.state_dict`.

#### Parameters

- **name** (*str*) – The name of the state variable. The variable will then be accessible at `self.name`.
- **default** (*Union[list, Tensor]*) – Default value of the state; can either be a `Tensor` or an empty list. The state will be reset to this value when `self.reset()` is called.
- **dist\_reduce\_fx** (*Optional*) – Function to reduce state across multiple processes in distributed mode. If value is "sum", "mean", "cat", "min" or "max" we will use `torch.sum`, `torch.mean`, `torch.cat`, `torch.min` and `torch.max` respectively, each with argument `dim=0`. Note that the "cat" reduction only makes sense if the state is a list, and not a tensor. The user can also pass a custom function in this parameter.
- **persistent** (*Optional*) – whether the state will be saved as part of the modules `state_dict`. Default is `False`.

**Return type**`None`

---

**Note:** Setting `dist_reduce_fx` to `None` will return the metric state synchronized across different processes. However, there won't be any reduction function applied to the synchronized metric state.

The metric states would be synced as follows

- If the metric state is `Tensor`, the synced value will be a stacked `Tensor` across the process dimension if the metric state was a `Tensor`. The original `Tensor` metric state retains dimension and hence the synchronized output will be of shape `(num_process, ...)`.
- If the metric state is a `list`, the synced value will be a `list` containing the combined elements from all processes.

---

**Note:** When passing a custom function to `dist_reduce_fx`, expect the synchronized metric state to follow the format discussed in the above note.

---

**Raises**

- **ValueError** – If default is not a tensor or an empty list.
- **ValueError** – If `dist_reduce_fx` is not callable or one of "mean", "sum", "cat", "min", "max" or `None`.

**clone()**

Make a copy of the metric.

**Return type**`Metric`**abstract compute()**

Override this method to compute the final metric value.

This method will automatically synchronize state variables when running in distributed backend.

**Return type**`Any`**double()**

Override default and prevent dtype casting.

Please use `Metric.set_dtype()` instead.



**Return type***Metric***float()**

Override default and prevent dtype casting.

Please use *Metric.set\_dtype()* instead.

**Return type***Metric***forward(\*args, \*\*kwargs)**

Aggregate and evaluate batch input directly.

Serves the dual purpose of both computing the metric on the current batch of inputs but also add the batch statistics to the overall accumulating metric state. Input arguments are the exact same as corresponding update method. The returned output is the exact same as the output of `compute`.

**Parameters**

- **args** (*Any*) – Any arguments as required by the metric update method.
- **kwargs** (*Any*) – Any keyword arguments as required by the metric update method.

**Return type***Any***Returns**

The output of the `compute` method evaluated on the current batch.

**Raises**

**TorchMetricsUserError** – If the metric is already synced and `forward` is called again.

**half()**

Override default and prevent dtype casting.

Please use *Metric.set\_dtype()* instead.

**Return type***Metric***persistent(mode=False)**

Change post-init if metric states should be saved to its `state_dict`.

**Return type***None***plot(\*\_, \*\*\_\_)**

Override this method plot the metric value.

**Return type***Any***reset()**

Reset metric state variables to their default value.

**Return type***None***set\_dtype(dst\_type)**

Transfer all metric state to specific dtype. Special version of standard *type* method.

**Parameters**

**dst\_type** (*type* or *string*) – the desired type.

**Return type**

*Metric*

**state\_dict**(*destination=None, prefix="", keep\_vars=False*)

Get the current state of metric as an dictionary.

**Parameters**

- **destination** (*Optional*[*Dict*[*str*, *Any*]]) – Optional dictionary, that if provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an *OrderedDict* will be created and returned.
- **prefix** (*str*) – optional string, a prefix added to parameter and buffer names to compose the keys in *state\_dict*.
- **keep\_vars** (*bool*) – by default the If set to ``True``, detaching will not be performed.

**Return type**

*Dict*[*str*, *Any*]

**sync**(*dist\_sync\_fn=None, process\_group=None, should\_sync=True, distributed\_available=None*)

Sync function for manually controlling when metrics states should be synced across processes.

**Parameters**

- **dist\_sync\_fn** (*Optional*[*Callable*]) – Function to be used to perform states synchronization
- **process\_group** (*Optional*[*Any*]) – Specify the process group on which synchronization is called. default: *None* (which selects the entire world)
- **should\_sync** (*bool*) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.
- **distributed\_available** (*Optional*[*Callable*]) – Function to determine if we are running inside a distributed setting

**Raises**

**TorchMetricsUserError** – If the metric is already synced and *sync* is called again.

**Return type**

*None*

**sync\_context**(*dist\_sync\_fn=None, process\_group=None, should\_sync=True, should\_unsync=True, distributed\_available=None*)

Context manager to synchronize states.

This context manager is used in distributed setting and makes sure that the local cache states are restored after yielding the synchronized state.

**Parameters**

- **dist\_sync\_fn** (*Optional*[*Callable*]) – Function to be used to perform states synchronization
- **process\_group** (*Optional*[*Any*]) – Specify the process group on which synchronization is called. default: *None* (which selects the entire world)
- **should\_sync** (*bool*) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.

- **should\_unsync** (`bool`) – Whether to restore the cache state so that the metrics can continue to be accumulated.
- **distributed\_available** (`Optional[Callable]`) – Function to determine if we are running inside a distributed setting

**Return type**

`Generator`

**type**(*dst\_type*)

Override default and prevent dtype casting.

Please use `Metric.set_dtype()` instead.

**Return type**

`Metric`

**unsync**(*should\_unsync=True*)

Unsync function for manually controlling when metrics states should be reverted back to their local states.

**Parameters**

**should\_unsync** (`bool`) – Whether to perform unsync

**Return type**

`None`

**abstract update**(*\*, \*\*\_\_*)

Override this method to update the state variables of your metric class.

**Return type**

`None`

**property device:** `device`

Return the device of the metric.

**property update\_called:** `bool`

Returns `True` if `update` or `forward` has been called initialization or last `reset`.

**property update\_count:** `int`

Get the number of times `update` and/or `forward` has been called since initialization or last `reset`.

## 1.5.2 Contributing your metric to TorchMetrics

Wanting to contribute the metric you have implemented? Great, we are always open to adding more metrics to `torchmetrics` as long as they serve a general purpose. However, to keep all our metrics consistent we request that the implementation and tests gets formatted in the following way:

1. Start by reading our [contribution guidelines](#).
2. First implement the functional backend. This takes cares of all the logic that goes into the metric. The code should be put into a single file placed under `torchmetrics/functional/"domain"/"new_metric".py` where `domain` is the type of metric (classification, regression, nlp etc) and `new_metric` is the name of the metric. In this file, there should be the following three functions:
  1. `_new_metric_update(...)`: everything that has to do with type/shape checking and all logic required before distributed syncing need to go here.
  2. `_new_metric_compute(...)`: all remaining logic.

3. `new_metric(...)`: essentially wraps the `_update` and `_compute` private functions into one public function that makes up the functional interface for the metric.

---

**Note:** The `functional accuracy` metric is a great example of this division of logic.

---

3. In a corresponding file placed in `torchmetrics/"domain"/"new_metric".py` create the module interface:
  1. Create a new module metric by subclassing `torchmetrics.Metric`.
  2. In the `__init__` of the module call `self.add_state` for as many metric states are needed for the metric to proper accumulate metric statistics.
  3. The module interface should essentially call the private `_new_metric_update(...)` in its `update` method and similarly the `_new_metric_compute(...)` function in its `compute`. No logic should really be implemented in the module interface. We do this to not have duplicate code to maintain.

---

**Note:** The module `Accuracy` metric that corresponds to the above functional example shows these steps.

---

4. Remember to add binding to the different relevant `__init__` files.
5. Testing is key to keeping `torchmetrics` trustworthy. This is why we have a very rigid testing protocol. This means that we in most cases require the metric to be tested against some other common framework (`sklearn`, `scipy` etc).
  1. Create a testing file in `unittests/"domain"/test_"new_metric".py`. Only one file is needed as it is intended to test both the functional and module interface.
  2. In that file, start by defining a number of test inputs that your metric should be evaluated on.
  3. Create a testclass `class NewMetric(MetricTester)` that inherits from `tests.helpers.testers.MetricTester`. This testclass should essentially implement the `test_"new_metric"_class` and `test_"new_metric"_fn` methods that respectively tests the module interface and the functional interface.
  4. The testclass should be parameterized (using `@pytest.mark.parametrize`) by the different test inputs defined initially. Additionally, the `test_"new_metric"_class` method should also be parameterized with an `ddp` parameter such that it gets tested in a distributed setting. If your metric has additional parameters, then make sure to also parameterize these such that different combinations of inputs and parameters gets tested.
  5. (optional) If your metric raises any exception, please add tests that showcase this.

---

**Note:** The `test file for accuracy` metric shows how to implement such tests.

---

If you only can figure out part of the steps, do not fear to send a PR. We will much rather receive working metrics that are not formatted exactly like our codebase, than not receiving any. Formatting can always be applied. We will gladly guide and/or help implement the remaining :]

## 1.6 TorchMetrics in PyTorch Lightning

TorchMetrics was originally created as part of [PyTorch Lightning](#), a powerful deep learning research framework designed for scaling models without boilerplate.

---

**Note:** TorchMetrics always offers compatibility with the last 2 major PyTorch Lightning versions, but we recommend to always keep both frameworks up-to-date for the best experience.

---

While TorchMetrics was built to be used with native PyTorch, using TorchMetrics with Lightning offers additional benefits:

- Modular metrics are automatically placed on the correct device when properly defined inside a `LightningModule`. This means that your data will always be placed on the same device as your metrics. No need to call `.to(device)` anymore!
- Native support for logging metrics in Lightning using `self.log` inside your `LightningModule`.
- The `.reset()` method of the metric will automatically be called at the end of an epoch.

The example below shows how to use a metric in your `LightningModule`:

```
class MyModel(LightningModule):

    def __init__(self, num_classes):
        ...
        self.accuracy = torchmetrics.classification.Accuracy(task="multiclass", num_
        ↪ classes=num_classes)

    def training_step(self, batch, batch_idx):
        x, y = batch
        preds = self(x)
        ...
        # log step metric
        self.accuracy(preds, y)
        self.log('train_acc_step', self.accuracy)
        ...

    def on_train_epoch_end(self):
        # log epoch metric
        self.log('train_acc_epoch', self.accuracy)
```

Metric logging in Lightning happens through the `self.log` or `self.log_dict` method. Both methods only support the logging of *scalar-tensors*. While the vast majority of metrics in torchmetrics returns a scalar tensor, some metrics such as `ConfusionMatrix`, `ROC`, `MeanAveragePrecision`, `ROUGEScore` return outputs that are non-scalar tensors (often dicts or list of tensors) and should therefore be dealt with separately. For info about the return type and shape please look at the documentation for the `compute` method for each metric you want to log.

### 1.6.1 Logging TorchMetrics

Logging metrics can be done in two ways: either logging the metric object directly or the computed metric values. When `Metric` objects, which return a scalar tensor are logged directly in Lightning using the `LightningModule` `self.log` method, Lightning will log the metric based on `on_step` and `on_epoch` flags present in `self.log(...)`. If `on_epoch` is `True`, the logger automatically logs the end of epoch metric value by calling `.compute()`.

---

**Note:** `sync_dist`, `sync_dist_group` and `reduce_fx` flags from `self.log(...)` don't affect the metric logging in any manner. The metric class contains its own distributed synchronization logic.

This however is only true for metrics that inherit the base class `Metric`, and thus the functional metric API provides no support for in-built distributed synchronization or reduction functions.

---

```
class MyModule(LightningModule):

    def __init__(self, num_classes):
        ...
        self.train_acc = torchmetrics.classification.Accuracy(task="multiclass", num_
↪classes=num_classes)
        self.valid_acc = torchmetrics.classification.Accuracy(task="multiclass", num_
↪classes=num_classes)

    def training_step(self, batch, batch_idx):
        x, y = batch
        preds = self(x)
        ...
        self.train_acc(preds, y)
        self.log('train_acc', self.train_acc, on_step=True, on_epoch=False)

    def validation_step(self, batch, batch_idx):
        logits = self(x)
        ...
        self.valid_acc(logits, y)
        self.log('valid_acc', self.valid_acc, on_step=True, on_epoch=True)
```

As an alternative to logging the metric object and letting Lightning take care of when to reset the metric etc. you can also manually log the output of the metrics.

```
class MyModule(LightningModule):

    def __init__(self):
        ...
        self.train_acc = torchmetrics.classification.Accuracy(task="multiclass", num_
↪classes=num_classes)
        self.valid_acc = torchmetrics.classification.Accuracy(task="multiclass", num_
↪classes=num_classes)

    def training_step(self, batch, batch_idx):
        x, y = batch
        preds = self(x)
        ...
        batch_value = self.train_acc(preds, y)
        self.log('train_acc_step', batch_value)
```

(continues on next page)

(continued from previous page)

```

def on_train_epoch_end(self):
    self.train_acc.reset()

def validation_step(self, batch, batch_idx):
    logits = self(x)
    ...
    self.valid_acc.update(logits, y)

def on_validation_epoch_end(self, outputs):
    self.log('valid_acc_epoch', self.valid_acc.compute())
    self.valid_acc.reset()

```

Note that logging metrics this way will require you to manually reset the metrics at the end of the epoch yourself. In general, we recommend logging the metric object to make sure that metrics are correctly computed and reset. Additionally, we highly recommend that the two ways of logging are not mixed as it can lead to wrong results.

**Note:** When using any Modular metric, calling `self.metric(...)` or `self.metric.forward(...)` serves the dual purpose of calling `self.metric.update()` on its input and simultaneously returning the metric value over the provided input. So if you are logging a metric *only* on epoch-level (as in the example above), it is recommended to call `self.metric.update()` directly to avoid the extra computation.

```

class MyModule(LightningModule):

    def __init__(self, num_classes):
        ...
        self.valid_acc = torchmetrics.classification.Accuracy(task="multiclass", num_
↪classes=num_classes)

    def validation_step(self, batch, batch_idx):
        logits = self(x)
        ...
        self.valid_acc.update(logits, y)
        self.log('valid_acc', self.valid_acc, on_step=True, on_epoch=True)

```

## 1.6.2 Common Pitfalls

The following contains a list of pitfalls to be aware of:

- Modular metrics contain internal states that should belong to only one DataLoader. In case you are using multiple DataLoaders, it is recommended to initialize a separate modular metric instances for each DataLoader and use them separately. The same holds for using separate metrics for training, validation and testing.

```

class MyModule(LightningModule):

    def __init__(self, num_classes):
        ...
        self.val_acc = nn.ModuleList(
            [torchmetrics.classification.Accuracy(task="multiclass", num_classes=num_
↪classes) for _ in range(2)]

```

(continues on next page)

(continued from previous page)

```

    )

    def val_dataloader(self):
        return [DataLoader(...), DataLoader(...)]

    def validation_step(self, batch, batch_idx, dataloader_idx):
        x, y = batch
        preds = self(x)
        ...
        self.val_acc[dataloader_idx](preds, y)
        self.log('val_acc', self.val_acc[dataloader_idx])

```

- Mixing the two logging methods by calling `self.log("val", self.metric)` in `{training}/{val}/{test}_step` method and then calling `self.log("val", self.metric.compute())` in the corresponding `{training}/{val}/{test}_epoch_end` method. Because the object is logged in the first case, Lightning will reset the metric before calling the second line leading to errors or nonsense results.
- Calling `self.log("val", self.metric(preds, target))` with the intention of logging the metric object. Because `self.metric(preds, target)` corresponds to calling the forward method, this will return a tensor and not the metric object. Such logging will be wrong in this case. Instead it is important to separate into separate lines:

```

def training_step(self, batch, batch_idx):
    x, y = batch
    preds = self(x)
    ...
    # log step metric
    self.accuracy(preds, y) # compute metrics
    self.log('train_acc_step', self.accuracy) # log metric object

```

- Using `~torchmetrics.MetricTracker` wrapper with Lightning is a special case, because the wrapper in itself is not a metric i.e. it does not inherit from the base `~torchmetrics.Metric` class but instead from `~torch.nn.ModuleList`. Thus, to log the output of this metric one needs to manually log the returned values (not the object) using `self.log` and for epoch level logging this should be done in the appropriate `on_***_epoch_end` method.

## 1.7 Concatenation

### 1.7.1 Module Interface

**class** `torchmetrics.CatMetric`(*nan\_strategy*='warn', *\*\*kwargs*)

Concatenate a stream of values.

As input to `forward` and `update` the metric accepts the following input

- **value** (`float` or `Tensor`): a single float or an tensor of float values with arbitrary shape `(...)`.

As output of `forward` and `compute` the metric returns the following output

- **agg** (`Tensor`): scalar float tensor with concatenated values over all input received

#### Parameters

- **nan\_strategy** (`Union[str, float]`) – options: - 'error': if any *nan* values are encountered will give a `RuntimeError` - 'warn': if any *nan* values are encountered will give a warning and



continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value

- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

**ValueError** – If `nan_strategy` is not one of `error`, `warn`, `ignore` or a float

#### Example

```
>>> from torch import tensor
>>> from torchmetrics.aggregation import CatMetric
>>> metric = CatMetric()
>>> metric.update(1)
>>> metric.update(tensor([2, 3]))
>>> metric.compute()
tensor([1., 2., 3.]
```

## 1.8 Maximum

### 1.8.1 Module Interface

**class** `torchmetrics.MaxMetric`(`nan_strategy='warn', **kwargs`)

Aggregate a stream of value into their maximum value.

As input to `forward` and `update` the metric accepts the following input

- **value** (*float* or *Tensor*): a single float or an tensor of float values with arbitrary shape `(...)`.

As output of `forward` and `compute` the metric returns the following output

- **agg** (*Tensor*): scalar float tensor with aggregated maximum value over all inputs received

#### Parameters

- **nan\_strategy** (*Union[str, float]*) – options: - 'error': if any *nan* values are encountered will give a `RuntimeError` - 'warn': if any *nan* values are encountered will give a warning and continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

**ValueError** – If `nan_strategy` is not one of `error`, `warn`, `ignore` or a float

## Example

```
>>> from torch import tensor
>>> from torchmetrics.aggregation import MaxMetric
>>> metric = MaxMetric()
>>> metric.update(1)
>>> metric.update(tensor([2, 3]))
>>> metric.compute()
tensor(3.)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

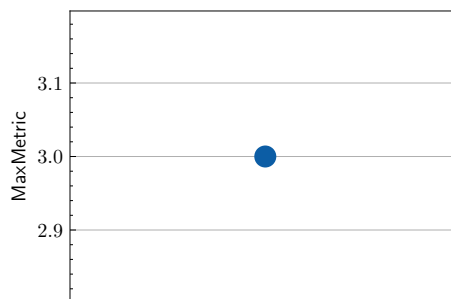
### Returns

Figure and Axes object

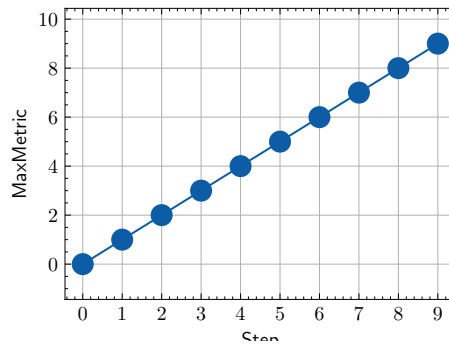
### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> from torchmetrics.aggregation import MaxMetric
>>> metric = MaxMetric()
>>> metric.update([1, 2, 3])
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torchmetrics.aggregation import MaxMetric
>>> metric = MaxMetric()
>>> values = [ ]
>>> for i in range(10):
...     values.append(metric(i))
>>> fig_, ax_ = metric.plot(values)
```



## 1.9 Mean

### 1.9.1 Module Interface

**class** `torchmetrics.MeanMetric`(*nan\_strategy*='warn', *\*\*kwargs*)

Aggregate a stream of value into their mean value.

As input to `forward` and `update` the metric accepts the following input

- **value** (`float` or `Tensor`): a single float or an tensor of float values with arbitrary shape  $(\dots)$ .
- **weight** (`float` or `Tensor`): a single float or an tensor of float value with arbitrary shape  $(\dots)$ . Needs to be broadcastable with the shape of `value` tensor.

As output of `forward` and `compute` the metric returns the following output

- **agg** (`Tensor`): scalar float tensor with aggregated (weighted) mean over all inputs received

#### Parameters

**nan\_strategy** (`Union[str, float]`) –

options:

- `'error'`: if any *nan* values are encountered will give a `RuntimeError`
- `'warn'`: if any *nan* values are encountered will give a warning and continue
- `'ignore'`: all *nan* values are silently removed
- a float: if a float is provided will impute any *nan* values with this value

*kwargs*: Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

**ValueError** – If *nan\_strategy* is not one of `error`, `warn`, `ignore` or a float

## Example

```
>>> from torchmetrics.aggregation import MeanMetric
>>> metric = MeanMetric()
>>> metric.update(1)
>>> metric.update(torch.tensor([2, 3]))
>>> metric.compute()
tensor(2.)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

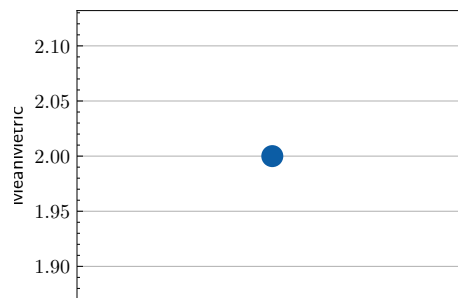
### Returns

Figure and Axes object

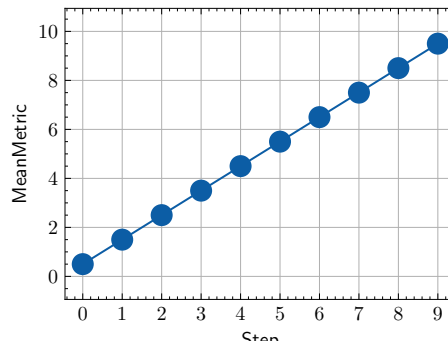
### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> from torchmetrics.aggregation import MeanMetric
>>> metric = MeanMetric()
>>> metric.update([1, 2, 3])
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torchmetrics.aggregation import MeanMetric
>>> metric = MeanMetric()
>>> values = [ ]
>>> for i in range(10):
...     values.append(metric([i, i+1]))
>>> fig_, ax_ = metric.plot(values)
```



## 1.10 Minimum

### 1.10.1 Module Interface

**class** torchmetrics.**MinMetric**(*nan\_strategy='warn', \*\*kwargs*)

Aggregate a stream of value into their minimum value.

As input to *forward* and *update* the metric accepts the following input

- **value** (**float** or **Tensor**): a single float or an tensor of float values with arbitrary shape ( $\dots$ ).

As output of *forward* and *compute* the metric returns the following output

- **agg** (**Tensor**): scalar float tensor with aggregated minimum value over all inputs received

#### Parameters

- **nan\_strategy** (**Union[str, float]**) – options: - 'error': if any *nan* values are encountered will give a `RuntimeError` - 'warn': if any *nan* values are encountered will give a warning and continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impute any *nan* values with this value
- **kwargs** (**Any**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

**ValueError** – If *nan\_strategy* is not one of `error`, `warn`, `ignore` or a float

#### Example

```
>>> from torch import tensor
>>> from torchmetrics.aggregation import MinMetric
>>> metric = MinMetric()
>>> metric.update(1)
>>> metric.update(tensor([2, 3]))
>>> metric.compute()
tensor(1.)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

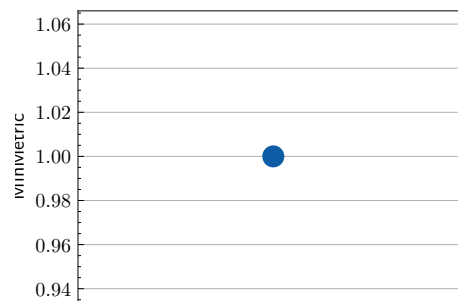
- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

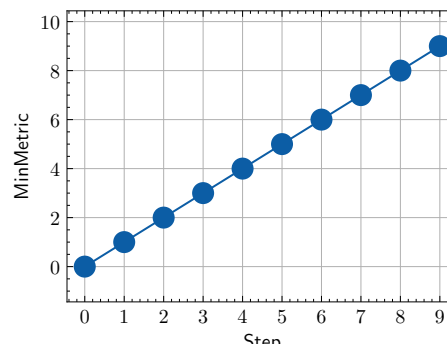
Figure and Axes object

**Raises****`ModuleNotFoundError`** – If `matplotlib` is not installed

```
>>> # Example plotting a single value
>>> from torchmetrics.aggregation import MinMetric
>>> metric = MinMetric()
>>> metric.update([1, 2, 3])
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torchmetrics.aggregation import MinMetric
>>> metric = MinMetric()
>>> values = [ ]
>>> for i in range(10):
...     values.append(metric(i))
>>> fig_, ax_ = metric.plot(values)
```



## 1.11 Running Mean

### 1.11.1 Module Interface

**class** `torchmetrics.RunningMean`(*window=5*, *nan\_strategy='warn'*, *\*\*kwargs*)

Aggregate a stream of value into their mean over a running window.

Using this metric compared to *MeanMetric* allows for calculating metrics over a running window of values, instead of the whole history of values. This is beneficial when you want to get a better estimate of the metric during training and don't want to wait for the whole training to finish to get epoch level estimates.

As input to *forward* and *update* the metric accepts the following input

- **value** (`float` or `Tensor`): a single float or an tensor of float values with arbitrary shape `(...)`.

As output of *forward* and *compute* the metric returns the following output

- **agg** (`Tensor`): scalar float tensor with aggregated sum over all inputs received

#### Parameters

- **window** (`int`) – The size of the running window.
- **nan\_strategy** (`Union[str, float]`) – options: - `'error'`: if any *nan* values are encountered will give a `RuntimeError` - `'warn'`: if any *nan* values are encountered will give a warning and continue - `'ignore'`: all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

**ValueError** – If *nan\_strategy* is not one of `error`, `warn`, `ignore` or a float

#### Example

```
>>> from torch import tensor
>>> from torchmetrics.aggregation import RunningMean
>>> metric = RunningMean(window=3)
>>> for i in range(6):
...     current_val = metric(tensor([i]))
...     running_val = metric.compute()
...     total_val = tensor(sum(list(range(i+1)))) / (i+1) # total mean over all
↳ samples
...     print(f"{current_val=}, {running_val=}, {total_val=}")
current_val=tensor(0.), running_val=tensor(0.), total_val=tensor(0.)
current_val=tensor(1.), running_val=tensor(0.5000), total_val=tensor(0.5000)
current_val=tensor(2.), running_val=tensor(1.), total_val=tensor(1.)
current_val=tensor(3.), running_val=tensor(2.), total_val=tensor(1.5000)
current_val=tensor(4.), running_val=tensor(3.), total_val=tensor(2.)
current_val=tensor(5.), running_val=tensor(4.), total_val=tensor(2.5000)
```

## 1.12 Running Sum

### 1.12.1 Module Interface

**class** torchmetrics.**RunningSum**(window=5, nan\_strategy='warn', \*\*kwargs)

Aggregate a stream of value into their sum over a running window.

Using this metric compared to *MeanMetric* allows for calculating metrics over a running window of values, instead of the whole history of values. This is beneficial when you want to get a better estimate of the metric during training and don't want to wait for the whole training to finish to get epoch level estimates.

As input to *forward* and *update* the metric accepts the following input

- **value** (**float** or **Tensor**): a single float or an tensor of float values with arbitrary shape ( $\dots$ ).

As output of *forward* and *compute* the metric returns the following output

- **agg** (**Tensor**): scalar float tensor with aggregated sum over all inputs received

#### Parameters

- **window** (**int**) – The size of the running window.
- **nan\_strategy** (**Union**[**str**, **float**]) – options: - 'error': if any *nan* values are encountered will give a `RuntimeError` - 'warn': if any *nan* values are encountered will give a warning and continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value
- **kwargs** (**Any**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

**ValueError** – If `nan_strategy` is not one of `error`, `warn`, `ignore` or a float

#### Example

```
>>> from torch import tensor
>>> from torchmetrics.aggregation import RunningSum
>>> metric = RunningSum(window=3)
>>> for i in range(6):
...     current_val = metric(tensor([i]))
...     running_val = metric.compute()
...     total_val = tensor(sum(list(range(i+1)))) # total sum over all samples
...     print(f"{current_val=}, {running_val=}, {total_val=}")
current_val=tensor(0.), running_val=tensor(0.), total_val=tensor(0)
current_val=tensor(1.), running_val=tensor(1.), total_val=tensor(1)
current_val=tensor(2.), running_val=tensor(3.), total_val=tensor(3)
current_val=tensor(3.), running_val=tensor(6.), total_val=tensor(6)
current_val=tensor(4.), running_val=tensor(9.), total_val=tensor(10)
current_val=tensor(5.), running_val=tensor(12.), total_val=tensor(15)
```



## 1.13 Sum

### 1.13.1 Module Interface

**class** `torchmetrics.SumMetric`(*nan\_strategy*='warn', *\*\*kwargs*)

Aggregate a stream of value into their sum.

As input to `forward` and `update` the metric accepts the following input

- **value** (`float` or `Tensor`): a single float or an tensor of float values with arbitrary shape (`...`).

As output of `forward` and `compute` the metric returns the following output

- **agg** (`Tensor`): scalar float tensor with aggregated sum over all inputs received

#### Parameters

- **nan\_strategy** (`Union[str, float]`) – options: - 'error': if any *nan* values are encountered will give a `RuntimeError` - 'warn': if any *nan* values are encountered will give a warning and continue - 'ignore': all *nan* values are silently removed - a float: if a float is provided will impude any *nan* values with this value
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

**ValueError** – If *nan\_strategy* is not one of `error`, `warn`, `ignore` or a float

#### Example

```
>>> from torch import tensor
>>> from torchmetrics.aggregation import SumMetric
>>> metric = SumMetric()
>>> metric.update(1)
>>> metric.update(tensor([2, 3]))
>>> metric.compute()
tensor(6.)
```

**plot**(*val*=None, *ax*=None)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

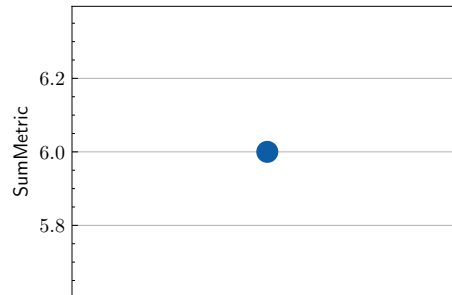
#### Returns

Figure and Axes object

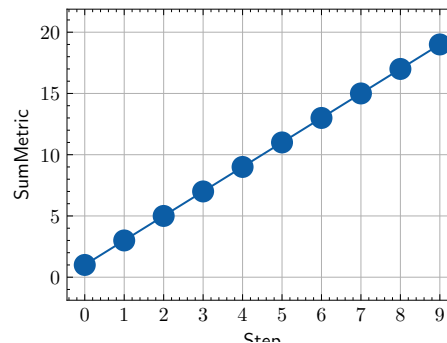
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> from torchmetrics.aggregation import SumMetric
>>> metric = SumMetric()
>>> metric.update([1, 2, 3])
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torch import rand, randint
>>> from torchmetrics.aggregation import SumMetric
>>> metric = SumMetric()
>>> values = [ ]
>>> for i in range(10):
...     values.append(metric([i, i+1]))
>>> fig_, ax_ = metric.plot(values)
```



## 1.14 Complex Scale-Invariant Signal-to-Noise Ratio (C-SI-SNR)

### 1.14.1 Module Interface

**class** torchmetrics.audio.**ComplexScaleInvariantSignalNoiseRatio**(zero\_mean=False, \*\*kwargs)

Calculate Complex scale-invariant signal-to-noise ratio (C-SI-SNR) metric for evaluating quality of audio.

As input to *forward* and *update* the metric accepts the following input

- `preds` (`Tensor`): real/complex float tensor with shape  $(\dots, \text{frequency}, \text{time}, 2) / (\dots, \text{frequency}, \text{time})$
- `target` (`Tensor`): real/complex float tensor with shape  $(\dots, \text{frequency}, \text{time}, 2) / (\dots, \text{frequency}, \text{time})$

As output of *forward* and *compute* the metric returns the following output

- `c_si_snr` (`Tensor`): float scalar tensor with average C-SI-SNR value over samples

#### Parameters

- `zero_mean` (`bool`) – if to zero mean target and preds or not
- `kwargs` (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- `ValueError` – If `zero_mean` is not an `bool`
- `TypeError` – If `preds` is not the shape  $(\dots, \text{frequency}, \text{time}, 2)$  (after being converted to real if it is complex). If `preds` and `target` does not have the same shape.

### Example

```
>>> import torch
>>> from torch import tensor
>>> from torchmetrics.audio import ComplexScaleInvariantSignalNoiseRatio
>>> g = torch.manual_seed(1)
>>> preds = torch.randn((1,257,100,2))
>>> target = torch.randn((1,257,100,2))
>>> c_si_snr = ComplexScaleInvariantSignalNoiseRatio()
>>> c_si_snr(preds, target)
tensor(-63.4849)
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- `val` (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- `ax` (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure and Axes object

#### Raises

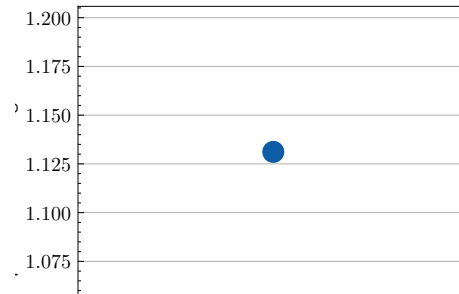
`ModuleNotFoundError` – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.audio import ComplexScaleInvariantSignalNoiseRatio
>>> metric = ComplexScaleInvariantSignalNoiseRatio()
```

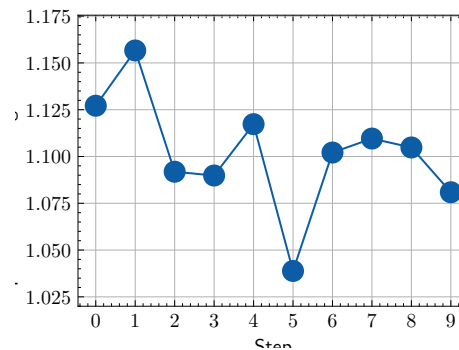
(continues on next page)

(continued from previous page)

```
>>> metric.update(torch.rand(1,257,100,2), torch.rand(1,257,100,2))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.audio import ComplexScaleInvariantSignalNoiseRatio
>>> metric = ComplexScaleInvariantSignalNoiseRatio()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.rand(1,257,100,2), torch.rand(1,257,100,2)))
>>> fig_, ax_ = metric.plot(values)
```



## 1.14.2 Functional Interface

`torchmetrics.functional.audio.complex_scale_invariant_signal_noise_ratio(preds, target, zero_mean=False)`

Complex scale-invariant signal-to-noise ratio (C-SI-SNR).

### Parameters

- **preds** (`Tensor`) – real/complex float tensor with shape `(..., frequency, time, 2)/(..., frequency, time)`
- **target** (`Tensor`) – real/complex float tensor with shape `(..., frequency, time, 2)/(..., frequency, time)`

- **zero\_mean** (`bool`) – When set to `True`, the mean of all signals is subtracted prior to computation of the metrics

**Return type**`Tensor`**Returns**Float tensor with shape `(...,)` of C-SI-SNR values per sample**Raises**

**RuntimeError** – If `preds` is not the shape `(..., frequency, time, 2)` (after being converted to real if it is complex). If `preds` and `target` does not have the same shape.

**Example**

```
>>> import torch
>>> from torchmetrics.functional.audio import complex_scale_invariant_signal_noise_
    ↪ ratio
>>> g = torch.manual_seed(1)
>>> preds = torch.randn((1, 257, 100, 2))
>>> target = torch.randn((1, 257, 100, 2))
>>> complex_scale_invariant_signal_noise_ratio(preds, target)
tensor([-63.4849])
```

## 1.15 Perceptual Evaluation of Speech Quality (PESQ)

### 1.15.1 Module Interface

**class** `torchmetrics.audio.pesq.PerceptualEvaluationSpeechQuality`(*fs, mode, n\_processes=1,*  
*\*\*kwargs*)

Calculate [Perceptual Evaluation of Speech Quality](#) (PESQ).

It's a recognized industry standard for audio quality that takes into considerations characteristics such as: audio sharpness, call volume, background noise, clipping, audio interference ect. PESQ returns a score between -0.5 and 4.5 with the higher scores indicating a better quality.

This metric is a wrapper for the [pesq package](#). Note that input will be moved to `cpu` to perform the metric calculation.

As input to `forward` and `update` the metric accepts the following input

- `preds` (`Tensor`): float tensor with shape `(..., time)`
- `target` (`Tensor`): float tensor with shape `(..., time)`

As output of `forward` and `compute` the metric returns the following output

- `pesq` (`Tensor`): float tensor with shape `(...,)` of PESQ value per sample

---

**Note:** using this metrics requires you to have `pesq` install. Either install as `pip install torchmetrics[audio]` or `pip install pesq`. `pesq` will compile with your currently installed version of `numpy`, meaning that if you upgrade `numpy` at some point in the future you will most likely have to reinstall `pesq`.

---

**Parameters**

- **fs** (`int`) – sampling frequency, should be 16000 or 8000 (Hz)
- **mode** (`str`) – 'wb' (wide-band) or 'nb' (narrow-band)
- **keep\_same\_device** – whether to move the pesq value to the device of preds
- **n\_processes** (`int`) – integer specifying the number of processes to run in parallel for the metric calculation. Only applies to batches of data and if `multiprocessing` package is installed.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises**

- **ModuleNotFoundError** – If pesq package is not installed
- **ValueError** – If fs is not either 8000 or 16000
- **ValueError** – If mode is not either "wb" or "nb"

**Example**

```
>>> import torch
>>> from torchmetrics.audio import PerceptualEvaluationSpeechQuality
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> nb_pesq = PerceptualEvaluationSpeechQuality(8000, 'nb')
>>> nb_pesq(preds, target)
tensor(2.2076)
>>> wb_pesq = PerceptualEvaluationSpeechQuality(16000, 'wb')
>>> wb_pesq(preds, target)
tensor(1.7359)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

**Parameters**

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

**Returns**

Figure and Axes object

**Raises**

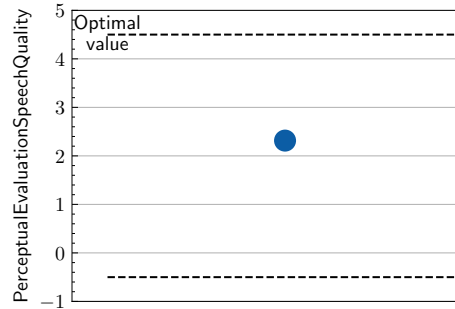
**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.audio import PerceptualEvaluationSpeechQuality
>>> metric = PerceptualEvaluationSpeechQuality(8000, 'nb')
```

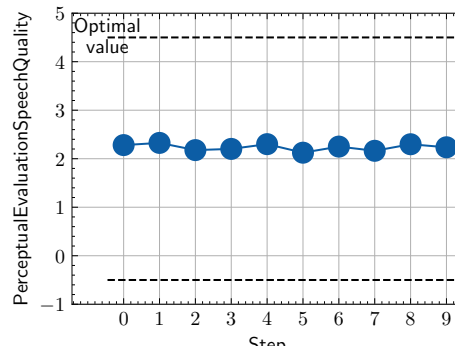
(continues on next page)

(continued from previous page)

```
>>> metric.update(torch.rand(8000), torch.rand(8000))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.audio import PerceptualEvaluationSpeechQuality
>>> metric = PerceptualEvaluationSpeechQuality(8000, 'nb')
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.rand(8000), torch.rand(8000)))
>>> fig_, ax_ = metric.plot(values)
```



### 1.15.2 Functional Interface

`torchmetrics.functional.audio.pesq.perceptual_evaluation_speech_quality(preds, target, fs, mode, keep_same_device=False, n_processes=1)`

Calculate [Perceptual Evaluation of Speech Quality](#) (PESQ).

It's a recognized industry standard for audio quality that takes into considerations characteristics such as: audio sharpness, call volume, background noise, clipping, audio interference ect. PESQ returns a score between -0.5 and 4.5 with the higher scores indicating a better quality.

This metric is a wrapper for the [pesq package](#). Note that input will be moved to *cpu* to perform the metric calculation.

**Note:** using this metrics requires you to have `pesq` install. Either install as `pip install torchmetrics[audio]` or `pip install pesq`. Note that `pesq` will compile with your currently installed version of `numpy`, meaning that if you upgrade `numpy` at some point in the future you will most likely have to reinstall `pesq`.

---

#### Parameters

- **preds** (`Tensor`) – float tensor with shape `(..., time)`
- **target** (`Tensor`) – float tensor with shape `(..., time)`
- **fs** (`int`) – sampling frequency, should be 16000 or 8000 (Hz)
- **mode** (`str`) – 'wb' (wide-band) or 'nb' (narrow-band)
- **keep\_same\_device** (`bool`) – whether to move the `pesq` value to the device of `preds`
- **n\_processes** (`int`) – integer specifying the number of processes to run in parallel for the metric calculation. Only applies to batches of data and if `multiprocessing` package is installed.

#### Return type

`Tensor`

#### Returns

Float tensor with shape `(...,)` of PESQ values per sample

#### Raises

- **ModuleNotFoundError** – If `pesq` package is not installed
- **ValueError** – If `fs` is not either 8000 or 16000
- **ValueError** – If `mode` is not either "wb" or "nb"
- **RuntimeError** – If `preds` and `target` do not have the same shape

#### Example

```
>>> from torch import randn
>>> from torchmetrics.functional.audio.pesq import perceptual_evaluation_speech_
↳ quality
>>> g = torch.manual_seed(1)
>>> preds = randn(8000)
>>> target = randn(8000)
>>> perceptual_evaluation_speech_quality(preds, target, 8000, 'nb')
tensor(2.2076)
>>> perceptual_evaluation_speech_quality(preds, target, 16000, 'wb')
tensor(1.7359)
```



## 1.16 Permutation Invariant Training (PIT)

### 1.16.1 Module Interface

`torchmetrics.PermutationInvariantTraining`  
alias of `_PermutationInvariantTraining`

### 1.16.2 Functional Interface

`torchmetrics.functional.permutation_invariant_training(preds, target, metric_func, mode='speaker-wise', eval_func='max', **kwargs)`

Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([[-0.0579,  0.3560, -0.9604], [-0.1719,  0.3205,  0.2951]])
>>> target = tensor([[ 1.0958, -0.1648,  0.5228], [-0.4100,  1.1942, -0.5103]])
>>> best_metric, best_perm = _permutation_invariant_training(
...     preds, target, _scale_invariant_signal_distortion_ratio)
>>> best_metric
tensor([-5.1091])
>>> best_perm
tensor([[0, 1]])
>>> pit_permutate(preds, best_perm)
:rtype: :py:data:~typing.Tuple\[:py:class:~torch.Tensor`, :py:class:~torch.
↪Tensor`]
```

```
tensor([[[[-0.0579, 0.3560, -0.9604],
          [-0.1719, 0.3205, 0.2951]]]])
```

## 1.17 Scale-Invariant Signal-to-Distortion Ratio (SI-SDR)

### 1.17.1 Module Interface

`torchmetrics.ScaleInvariantSignalDistortionRatio`  
alias of `_ScaleInvariantSignalDistortionRatio`

### 1.17.2 Functional Interface

`torchmetrics.functional.scale_invariant_signal_distortion_ratio(preds, target, zero_mean=False)`

Wrapper for deprecated import.

```
>>> from torch import tensor
>>> target = tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = tensor([2.5, 0.0, 2.0, 8.0])
:rtype: :py:class:~torch.Tensor`
```

```
>>> _scale_invariant_signal_distortion_ratio(preds, target)
tensor(18.4030)
```

## 1.18 Scale-Invariant Signal-to-Noise Ratio (SI-SNR)

### 1.18.1 Module Interface

`torchmetrics.ScaleInvariantSignalNoiseRatio`  
alias of `_ScaleInvariantSignalNoiseRatio`

### 1.18.2 Functional Interface

`torchmetrics.functional.scale_invariant_signal_noise_ratio(preds, target)`  
Wrapper for deprecated import.

```
>>> from torch import tensor
>>> target = tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = tensor([2.5, 0.0, 2.0, 8.0])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _scale_invariant_signal_noise_ratio(preds, target)
tensor(15.0918)
```

## 1.19 Short-Time Objective Intelligibility (STOI)

### 1.19.1 Module Interface

**class** `torchmetrics.audio.stoi.ShortTimeObjectiveIntelligibility(fs, extended=False, **kwargs)`  
Calculate STOI (Short-Time Objective Intelligibility) metric for evaluating speech signals.

Intelligibility measure which is highly correlated with the intelligibility of degraded speech signals, e.g., due to additive noise, single-/multi-channel noise reduction, binary masking and vocoded speech as in CI simulations. The STOI-measure is intrusive, i.e., a function of the clean and degraded speech signals. STOI may be a good alternative to the speech intelligibility index (SII) or the speech transmission index (STI), when you are interested in the effect of nonlinear processing to noisy speech, e.g., noise reduction, binary masking algorithms, on speech intelligibility. Description taken from [Cees Taal's website](#) and for further details see [STOI ref1](#) and [STOI ref2](#).

This metric is a wrapper for the [pystoi package](#). As the implementation backend implementation only supports calculations on CPU, all input will automatically be moved to CPU to perform the metric calculation before being moved back to the original device.

As input to *forward* and *update* the metric accepts the following input

- `preds` ([Tensor](#)): float tensor with shape `(..., time)`
- `target` ([Tensor](#)): float tensor with shape `(..., time)`

As output of *forward* and *compute* the metric returns the following output

- `stoi` ([Tensor](#)): float scalar tensor

---

**Note:** using this metrics requires you to have pystoi install. Either install as `pip install torchmetrics[audio]` or `pip install pystoi`.

---

### Parameters

- **fs** (`int`) – sampling frequency (Hz)
- **extended** (`bool`) – whether to use the extended STOI described in [STOI ref3](#).
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Raises

**ModuleNotFoundError** – If pystoi package is not installed

### Example

```
>>> import torch
>>> from torchmetrics.audio import ShortTimeObjectiveIntelligibility
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> stoi = ShortTimeObjectiveIntelligibility(8000, False)
>>> stoi(preds, target)
tensor(-0.0100)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

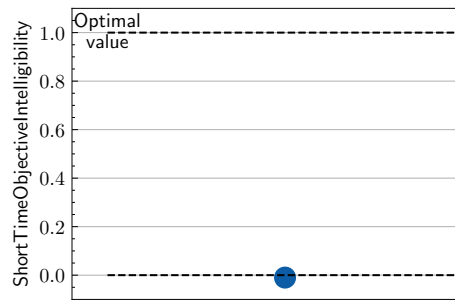
### Returns

Figure and Axes object

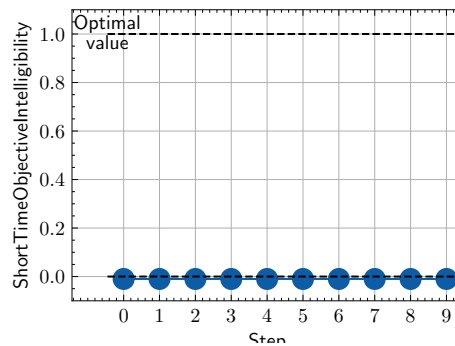
### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.audio import ShortTimeObjectiveIntelligibility
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> metric = ShortTimeObjectiveIntelligibility(8000, False)
>>> metric.update(preds, target)
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.audio import ShortTimeObjectiveIntelligibility
>>> metric = ShortTimeObjectiveIntelligibility(8000, False)
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(preds, target))
>>> fig_, ax_ = metric.plot(values)
```



## 1.19.2 Functional Interface

`torchmetrics.functional.audio.stoi.short_time_objective_intelligibility(preds, target, fs, extended=False, keep_same_device=False)`

Calculate STOI (Short-Time Objective Intelligibility) metric for evaluating speech signals.

Intelligibility measure which is highly correlated with the intelligibility of degraded speech signals, e.g., due to additive noise, single-/multi-channel noise reduction, binary masking and vocoded speech as in CI simulations. The STOI-measure is intrusive, i.e., a function of the clean and degraded speech signals. STOI may be a good alternative to the speech intelligibility index (SII) or the speech transmission index (STI), when you are interested in the effect of nonlinear processing to noisy speech, e.g., noise reduction, binary masking algorithms, on speech intelligibility. Description taken from [Cees Taal's website](#) and for further details see [STOI ref1](#) and [STOI ref2](#).

This metric is a wrapper for the `pystoi` package. As the implementation backend implementation only supports calculations on CPU, all input will automatically be moved to CPU to perform the metric calculation before being moved back to the original device.

---

**Note:** using this metrics requires you to have `pystoi` install. Either install as `pip install torchmetrics[audio]` or `pip install pystoi`

---

#### Parameters

- **preds** (`Tensor`) – float tensor with shape  $(\dots, \text{time})$
- **target** (`Tensor`) – float tensor with shape  $(\dots, \text{time})$
- **fs** (`int`) – sampling frequency (Hz)
- **extended** (`bool`) – whether to use the extended STOI described in [STOI ref3](#).
- **keep\_same\_device** (`bool`) – whether to move the stoi value to the device of preds

#### Return type

`Tensor`

#### Returns

stoi value of shape  $[\dots]$

#### Raises

- **ModuleNotFoundError** – If `pystoi` package is not installed
- **RuntimeError** – If preds and target does not have the same shape

#### Example

```
>>> import torch
>>> from torchmetrics.functional.audio.stoi import short_time_objective_
↪intelligibility
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> short_time_objective_intelligibility(preds, target, 8000).float()
tensor(-0.0100)
```

## 1.20 Signal to Distortion Ratio (SDR)

### 1.20.1 Module Interface

`torchmetrics.SignalDistortionRatio`  
alias of `_SignalDistortionRatio`

## 1.20.2 Functional Interface

`torchmetrics.functional.signal_distortion_ratio(preds, target, use_cg_iter=None, filter_length=512, zero_mean=False, load_diag=None)`

Wrapper for deprecated import.

```
>>> import torch
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> target = torch.randn(8000)
>>> _signal_distortion_ratio(preds, target)
tensor(-12.0589)
>>> # use with permutation_invariant_training
>>> preds = torch.randn(4, 2, 8000) # [batch, spk, time]
>>> target = torch.randn(4, 2, 8000)
>>> best_metric, best_perm = _permutation_invariant_training(preds, target, _signal_
↪ distortion_ratio)
>>> best_metric
tensor([-11.6375, -11.4358, -11.7148, -11.6325])
>>> best_perm
tensor([[1, 0],
        [0, 1],
:rtype: :py:class:`~torch.Tensor`
```

[1, 0], [0, 1]])

## 1.21 Signal-to-Noise Ratio (SNR)

### 1.21.1 Module Interface

`torchmetrics.SignalNoiseRatio`  
alias of `_SignalNoiseRatio`

### 1.21.2 Functional Interface

`torchmetrics.functional.signal_noise_ratio(preds, target, zero_mean=False)`

Wrapper for deprecated import.

```
>>> from torch import tensor
>>> target = tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = tensor([2.5, 0.0, 2.0, 8.0])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _signal_noise_ratio(preds, target)
tensor(16.1805)
```

## 1.22 Speech-to-Reverberation Modulation Energy Ratio (SRMR)

### 1.22.1 Module Interface

```
class torchmetrics.audio.srmr.SpeechReverberationModulationEnergyRatio(fs,
                                                                    n_cochlear_filters=23,
                                                                    low_freq=125,
                                                                    min_cf=4,
                                                                    max_cf=None,
                                                                    norm=False,
                                                                    fast=False, **kwargs)
```

Calculate [Speech-to-Reverberation Modulation Energy Ratio \(SRMR\)](#).

SRMR is a non-intrusive metric for speech quality and intelligibility based on a modulation spectral representation of the speech signal. This code is translated from [SRMRToolbox](#) and [SRMRpy](#).

As input to `forward` and `update` the metric accepts the following input

- `preds` ([Tensor](#)): float tensor with shape `(..., time)`

As output of `forward` and `compute` the metric returns the following output

- `srmr` ([Tensor](#)): float scalar tensor

---

**Note:** using this metrics requires you to have `gammatone` and `torchaudio` installed. Either install as `pip install torchmetrics[audio]` or `pip install torchaudio` and `pip install git+https://github.com/detly/gammatone`.

---



---

**Note:** This implementation is experimental, and might not be consistent with the matlab implementation [SRMRToolbox](#), especially the fast implementation. The slow versions, a) `fast=False`, `norm=False`, `max_cf=128`, b) `fast=False`, `norm=True`, `max_cf=30`, have a relatively small inconsistency.

---

#### Parameters

- `fs` ([int](#)) – the sampling rate
- `n_cochlear_filters` ([int](#)) – Number of filters in the acoustic filterbank
- `low_freq` ([float](#)) – determines the frequency cutoff for the corresponding gammatone filterbank.
- `min_cf` ([float](#)) – Center frequency in Hz of the first modulation filter.
- `max_cf` ([Optional\[float\]](#)) – Center frequency in Hz of the last modulation filter. If `None` is given, then 30 Hz will be used for `norm==False`, otherwise 128 Hz will be used.
- `norm` ([bool](#)) – Use modulation spectrum energy normalization
- `fast` ([bool](#)) – Use the faster version based on the gammatonegram. Note: this argument is inherited from [SRMRpy](#). As the translated code is based to pytorch, setting `fast=True` may slow down the speed for calculating this metric on GPU.

#### Raises

[ModuleNotFoundError](#) – If `gammatone` or `torchaudio` package is not installed

## Example

```
>>> import torch
>>> from torchmetrics.audio import SpeechReverberationModulationEnergyRatio
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> srmr = SpeechReverberationModulationEnergyRatio(8000)
>>> srmr(preds)
tensor(0.3354)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

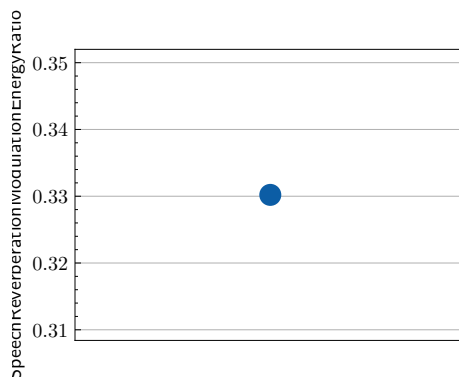
### Returns

Figure and Axes object

### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.audio import SpeechReverberationModulationEnergyRatio
>>> metric = SpeechReverberationModulationEnergyRatio(8000)
>>> metric.update(torch.rand(8000))
>>> fig_, ax_ = metric.plot()
```



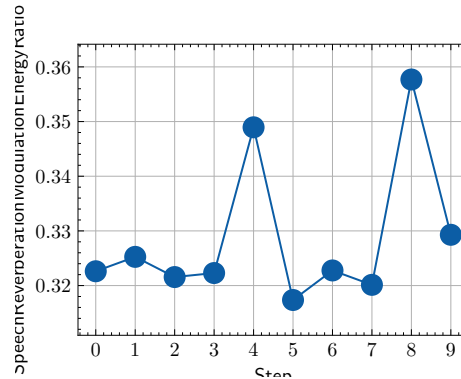
```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.audio import SpeechReverberationModulationEnergyRatio
>>> metric = SpeechReverberationModulationEnergyRatio(8000)
>>> values = [ ]
```

(continues on next page)



(continued from previous page)

```
>>> for _ in range(10):
...     values.append(metric(torch.rand(8000)))
>>> fig_, ax_ = metric.plot(values)
```



### 1.22.2 Functional Interface

`torchmetrics.functional.audio.srmr.speech_reverberation_modulation_energy_ratio(preds, fs,`  
`n_cochlear_filters=23,`  
`low_freq=125,`  
`min_cf=4,`  
`max_cf=None,`  
`norm=False,`  
`fast=False)`

Calculate [Speech-to-Reverberation Modulation Energy Ratio \(SRMR\)](#).

SRMR is a non-intrusive metric for speech quality and intelligibility based on a modulation spectral representation of the speech signal. This code is translated from [SRMRToolbox](#) and [SRMRpy](#).

#### Parameters

- **preds** ([Tensor](#)) – shape `(..., time)`
- **fs** ([int](#)) – the sampling rate
- **n\_cochlear\_filters** ([int](#)) – Number of filters in the acoustic filterbank
- **low\_freq** ([float](#)) – determines the frequency cutoff for the corresponding gammatone filterbank.
- **min\_cf** ([float](#)) – Center frequency in Hz of the first modulation filter.
- **max\_cf** ([Optional\[float\]](#)) – Center frequency in Hz of the last modulation filter. If None is given, then 30 Hz will be used for `norm=False`, otherwise 128 Hz will be used.
- **norm** ([bool](#)) – Use modulation spectrum energy normalization
- **fast** ([bool](#)) – Use the faster version based on the gammatonegram. Note: this argument is inherited from [SRMRpy](#). As the translated code is based to pytorch, setting `fast=True` may slow down the speed for calculating this metric on GPU.

**Note:** using this metrics requires you to have `gammatone` and `torchaudio` installed. Either install as

```
pip install torchmetrics[audio] or pip install torchaudio and pip install git+https://github.com/detly/gammatone.
```

---

**Note:** This implementation is experimental, and might not be consistent with the matlab implementation [SRM-RToolbox](#), especially the fast implementation. The slow versions, a) fast=False, norm=False, max\_cf=128, b) fast=False, norm=True, max\_cf=30, have a relatively small inconsistence.

---

**Returns**

srmr value, shape (...)

**Return type**

Tensor

**Raises**

**ModuleNotFoundError** – If gammatone or torchaudio package is not installed

**Example**

```
>>> import torch
>>> from torchmetrics.functional.audio import speech_reverberation_modulation_
    ↪energy_ratio
>>> g = torch.manual_seed(1)
>>> preds = torch.randn(8000)
>>> speech_reverberation_modulation_energy_ratio(preds, 8000)
tensor([0.3354], dtype=torch.float64)
```

## 1.23 Accuracy

### 1.23.1 Module Interface

```
class torchmetrics.Accuracy(task: typing_extensions.Literal[binary, multiclass, multilabel], threshold: float
    = 0.5, num_classes: int | None = None, num_labels: int | None = None,
    average: typing_extensions.Literal[micro, macro, weighted, none] | None =
    'micro', multidim_average: typing_extensions.Literal[global, samplewise] =
    'global', top_k: int | None = 1, ignore_index: int | None = None, validate_args:
    bool = True, **kwargs: ~typing.Any)
```

Compute *Accuracy*.

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

This module is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of `BinaryAccuracy`, `MulticlassAccuracy` and `MultilabelAccuracy` for the specific details of each argument influence and examples.

**Legacy Example:**

```
>>> from torch import tensor
>>> target = tensor([0, 1, 2, 3])
>>> preds = tensor([0, 2, 1, 3])
>>> accuracy = Accuracy(task="multiclass", num_classes=4)
>>> accuracy(preds, target)
tensor(0.5000)

>>> target = tensor([0, 1, 2])
>>> preds = tensor([[0.1, 0.9, 0], [0.3, 0.1, 0.6], [0.2, 0.5, 0.3]])
>>> accuracy = Accuracy(task="multiclass", num_classes=3, top_k=2)
>>> accuracy(preds, target)
tensor(0.6667)
```

Initialize task metric.

## BinaryAccuracy

```
class torchmetrics.classification.BinaryAccuracy(threshold=0.5, multidim_average='global',
                                                ignore_index=None, validate_args=True,
                                                **kwargs)
```

Compute *Accuracy* for binary tasks.

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): An int or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (**Tensor**): An int tensor of shape (N, ...)

As output to forward and compute the metric returns the following output:

- **ba** (**Tensor**): If **multidim\_average** is set to **global**, metric returns a scalar value. If **multidim\_average** is set to **samplewise**, the metric returns (N,) vector consisting of a scalar value per sample.

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **threshold** (**float**) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional**[**int**]) – Specifies a target value that is ignored and does not contribute to the metric calculation

- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryAccuracy
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryAccuracy()
>>> metric(preds, target)
tensor(0.6667)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryAccuracy
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryAccuracy()
>>> metric(preds, target)
tensor(0.6667)
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import BinaryAccuracy
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = BinaryAccuracy(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.3333, 0.1667])
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure object and Axes object

#### Raises

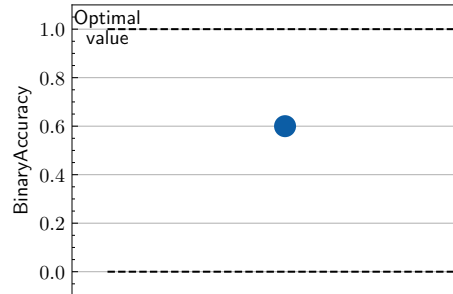
**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinaryAccuracy
>>> metric = BinaryAccuracy()
```

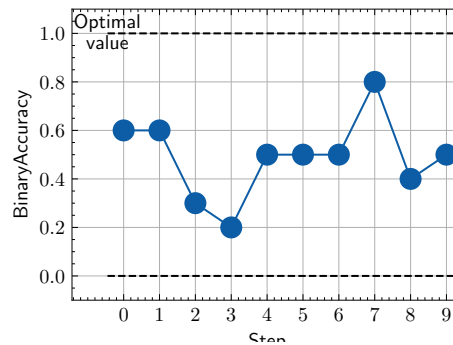
(continues on next page)

(continued from previous page)

```
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryAccuracy
>>> metric = BinaryAccuracy()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassAccuracy

```
class torchmetrics.classification.MulticlassAccuracy(num_classes, top_k=1, average='macro',
                                                    multidim_average='global',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Compute *Accuracy* for multiclass tasks.

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int tensor of shape (N, ...) or float tensor of shape (N, C, ...). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** ([Tensor](#)): An int tensor of shape (N, ...)

As output to forward and compute the metric returns the following output:

- **mca** ([Tensor](#)): A tensor with the accuracy score whose returned shape depends on the **average** and **multidim\_average** arguments:
  - If **multidim\_average** is set to **global**:
    - \* If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
    - \* If **average**=None/'none', the shape will be (C,)
  - If **multidim\_average** is set to **samplewise**:
    - \* If **average**='micro'/'macro'/'weighted', the shape will be (N,)
    - \* If **average**=None/'none', the shape will be (N, C)

### Parameters

- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when **preds** contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassAccuracy
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassAccuracy(num_classes=3)
>>> metric(preds, target)
```

(continues on next page)

(continued from previous page)

```

tensor(0.8333)
>>> mca = MulticlassAccuracy(num_classes=3, average=None)
>>> mca(preds, target)
tensor([0.5000, 1.0000, 1.0000])

```

**Example (preds is float tensor):**

```

>>> from torchmetrics.classification import MulticlassAccuracy
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                 [0.22, 0.61, 0.17],
...                 [0.71, 0.09, 0.20],
...                 [0.05, 0.82, 0.13]])
>>> metric = MulticlassAccuracy(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> mca = MulticlassAccuracy(num_classes=3, average=None)
>>> mca(preds, target)
tensor([0.5000, 1.0000, 1.0000])

```

**Example (multidim tensors):**

```

>>> from torchmetrics.classification import MulticlassAccuracy
>>> target = tensor([[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]])
>>> preds = tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> metric = MulticlassAccuracy(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.5000, 0.2778])
>>> mca = MulticlassAccuracy(num_classes=3, multidim_average='samplewise',
...                           average=None)
>>> mca(preds, target)
tensor([[1.0000, 0.0000, 0.5000],
        [0.0000, 0.3333, 0.5000]])

```

**plot**(val=None, ax=None)

Plot a single or multiple values from the metric.

**Parameters**

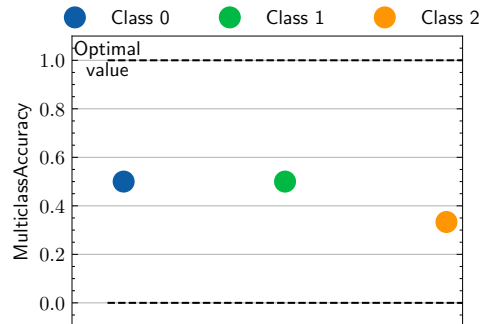
- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

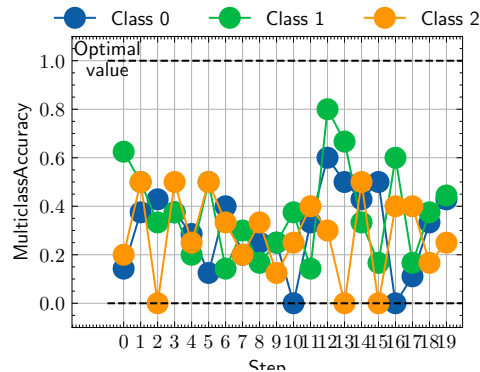
Figure object and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randint
>>> # Example plotting a single value per class
>>> from torchmetrics.classification import MulticlassAccuracy
>>> metric = MulticlassAccuracy(num_classes=3, average=None)
>>> metric.update(randint(3, (20,)), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randint
>>> # Example plotting a multiple values per class
>>> from torchmetrics.classification import MulticlassAccuracy
>>> metric = MulticlassAccuracy(num_classes=3, average=None)
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randint(3, (20,)), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelAccuracy

```
class torchmetrics.classification.MultilabelAccuracy(num_labels, threshold=0.5, average='macro',
multidim_average='global',
ignore_index=None, validate_args=True,
**kwargs)
```

Compute *Accuracy* for multilabel tasks.

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$



Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to forward and update the metric accepts the following input:

- **preds** (`Tensor`): An int or float tensor of shape (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (`Tensor`): An int tensor of shape (N, C, ...)

As output to forward and compute the metric returns the following output:

- **mla** (`Tensor`): A tensor with the accuracy score whose returned shape depends on the **average** and **multidim\_average** arguments:
  - If **multidim\_average** is set to **global**:
    - \* If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
    - \* If **average**=None/'none', the shape will be (C,)
  - If **multidim\_average** is set to **samplewise**:
    - \* If **average**='micro'/'macro'/'weighted', the shape will be (N,)
    - \* If **average**=None/'none', the shape will be (N, C)

#### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelAccuracy
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
```

(continues on next page)

(continued from previous page)

```
>>> metric = MultilabelAccuracy(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> mla = MultilabelAccuracy(num_labels=3, average=None)
>>> mla(preds, target)
tensor([1.0000, 0.5000, 0.5000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelAccuracy
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelAccuracy(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> mla = MultilabelAccuracy(num_labels=3, average=None)
>>> mla(preds, target)
tensor([1.0000, 0.5000, 0.5000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MultilabelAccuracy
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor(
...     [
...         [[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...         [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]],
...     ]
... )
>>> mla = MultilabelAccuracy(num_labels=3, multidim_average='samplewise')
>>> mla(preds, target)
tensor([0.3333, 0.1667])
>>> mla = MultilabelAccuracy(num_labels=3, multidim_average='samplewise',
... ↪ average=None)
>>> mla(preds, target)
tensor([[0.5000, 0.5000, 0.0000],
        [0.0000, 0.0000, 0.5000]])
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

**Parameters**

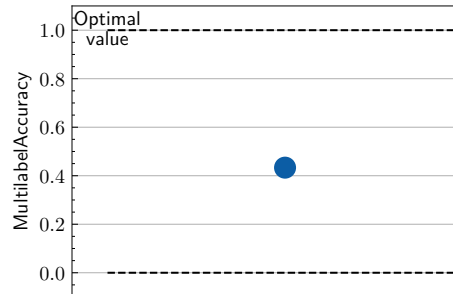
- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

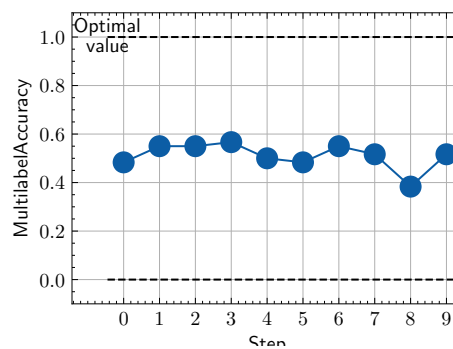
Figure and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelAccuracy
>>> metric = MultilabelAccuracy(num_labels=3)
>>> metric.update(randint(2, (20, 3)), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import MultilabelAccuracy
>>> metric = MultilabelAccuracy(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2, (20, 3)), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.23.2 Functional Interface

`torchmetrics.functional.classification.accuracy(preds, target, task, threshold=0.5, num_classes=None, num_labels=None, average='micro', multidim_average='global', top_k=1, ignore_index=None, validate_args=True)`

Compute *Accuracy*. :rtype: `Tensor`

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either 'binary', 'multiclass' or 'multilabel'. See the documentation of `binary_accuracy()`, `multiclass_accuracy()` and `multilabel_accuracy()` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([0, 1, 2, 3])
>>> preds = tensor([0, 2, 1, 3])
>>> accuracy(preds, target, task="multiclass", num_classes=4)
tensor(0.5000)
```

```
>>> target = tensor([0, 1, 2])
>>> preds = tensor([[0.1, 0.9, 0], [0.3, 0.1, 0.6], [0.2, 0.5, 0.3]])
>>> accuracy(preds, target, task="multiclass", num_classes=3, top_k=2)
tensor(0.6667)
```

#### binary\_accuracy

`torchmetrics.functional.classification.binary_accuracy(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Compute *Accuracy* for binary tasks.

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): (N, ...)

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions

- **target** (`Tensor`) – Tensor with true labels
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type**`Tensor`**Returns**

If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_accuracy
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> binary_accuracy(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_accuracy
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_accuracy(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import binary_accuracy
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> binary_accuracy(preds, target, multidim_average='samplewise')
tensor([0.3333, 0.1667])
```

## multiclass\_accuracy

```
torchmetrics.functional.classification.multiclass_accuracy(preds, target, num_classes,
                                                           average='macro', top_k=1,
                                                           multidim_average='global',
                                                           ignore_index=None,
                                                           validate_args=True)
```

Compute *Accuracy* for multiclass tasks.

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor

- If average=None/'none', the shape will be (C,)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N,)
  - If average=None/'none', the shape will be (N, C)

**Return type**

The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_accuracy
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_accuracy(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_accuracy(preds, target, num_classes=3, average=None)
tensor([0.5000, 1.0000, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_accuracy
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> multiclass_accuracy(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_accuracy(preds, target, num_classes=3, average=None)
tensor([0.5000, 1.0000, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_accuracy
>>> target = tensor([[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]])
>>> preds = tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> multiclass_accuracy(preds, target, num_classes=3, multidim_average=
↳ 'samplewise')
tensor([0.5000, 0.2778])
>>> multiclass_accuracy(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[1.0000, 0.0000, 0.5000],
        [0.0000, 0.3333, 0.5000]])
```

## multilabel\_accuracy

```
torchmetrics.functional.classification.multilabel_accuracy(preds, target, num_labels,
                                                           threshold=0.5, average='macro',
                                                           multidim_average='global',
                                                           ignore_index=None,
                                                           validate_args=True)
```

Compute *Accuracy* for multilabel tasks.

$$\text{Accuracy} = \frac{1}{N} \sum_i^N 1(y_i = \hat{y}_i)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor



- If average=None/'none', the shape will be (C,)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N,)
  - If average=None/'none', the shape will be (N, C)

**Return type**

The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_accuracy
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_accuracy(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_accuracy(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.5000, 0.5000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_accuracy
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_accuracy(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_accuracy(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.5000, 0.5000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multilabel_accuracy
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> multilabel_accuracy(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0.3333, 0.1667])
>>> multilabel_accuracy(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.5000, 0.5000, 0.0000],
        [0.0000, 0.0000, 0.5000]])
```

## 1.24 AUROC

### 1.24.1 Module Interface

```
class torchmetrics.AUROC(task: typing_extensions.Literal[binary, multiclass, multilabel], thresholds: int |
    List[float] | Tensor | None = None, num_classes: int | None = None, num_labels: int
    | None = None, average: typing_extensions.Literal[macro, weighted, none] | None =
    'macro', max_fpr: float | None = None, ignore_index: int | None = None,
    validate_args: bool = True, **kwargs: Any)
```

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC).

The AUROC score summarizes the ROC curve into a single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

This module is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of BinaryAUROC, MulticlassAUROC and MultilabelAUROC for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> preds = tensor([0.13, 0.26, 0.08, 0.19, 0.34])
>>> target = tensor([0, 0, 1, 1, 1])
>>> auroc = AUROC(task="binary")
>>> auroc(preds, target)
tensor(0.5000)
```

```
>>> preds = tensor([[0.90, 0.05, 0.05],
...                 [0.05, 0.90, 0.05],
...                 [0.05, 0.05, 0.90],
...                 [0.85, 0.05, 0.10],
...                 [0.10, 0.10, 0.80]])
>>> target = tensor([0, 1, 1, 2, 2])
>>> auroc = AUROC(task="multiclass", num_classes=3)
>>> auroc(preds, target)
tensor(0.7778)
```

Initialize task metric.

### BinaryAUROC

```
class torchmetrics.classification.BinaryAUROC(max_fpr=None, thresholds=None, ignore_index=None,
    validate_args=True, **kwargs)
```

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) for binary tasks.

The AUROC score summarizes the ROC curve into a single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

As input to forward and update the metric accepts the following input:

- **preds** (Tensor): A float tensor of shape (N, ...) containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.

- **target** ([Tensor](#)): An int tensor of shape (N, ...) containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

As output to `forward` and `compute` the metric returns the following output:

- **b\_auroc** ([Tensor](#)): A single scalar with the auroc score.

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

### Parameters

- **max\_fpr** ([Optional\[float\]](#)) – If not *None*, calculates standardized partial AUC over the range  $[0, \text{max\_fpr}]$ .
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryAUROC
>>> preds = tensor([0, 0.5, 0.7, 0.8])
>>> target = tensor([0, 1, 1, 0])
>>> metric = BinaryAUROC(thresholds=None)
>>> metric(preds, target)
tensor(0.5000)
>>> b_auroc = BinaryAUROC(thresholds=5)
>>> b_auroc(preds, target)
tensor(0.5000)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** ([Union\[Tensor, Sequence\[Tensor\], None\]](#)) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** ([Optional\[Axes\]](#)) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

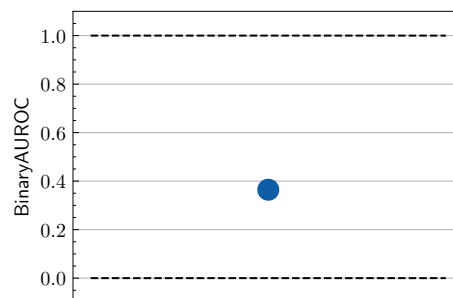
**Returns**

Figure and Axes object

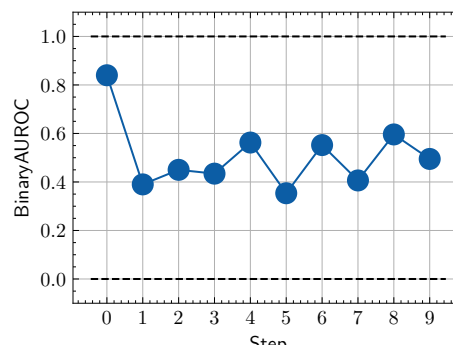
**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single
>>> import torch
>>> from torchmetrics.classification import BinaryAUROC
>>> metric = BinaryAUROC()
>>> metric.update(torch.rand(20,), torch.randint(2, (20,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.classification import BinaryAUROC
>>> metric = BinaryAUROC()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.rand(20,), torch.randint(2, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassAUROC

```
class torchmetrics.classification.MulticlassAUROC(num_classes, average='macro', thresholds=None,
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) for multiclass tasks.

The AUROC score summarizes the ROC curve into a single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (**Tensor**): A float tensor of shape  $(N, C, \dots)$  containing probabilities or logits for each observation. If `preds` has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** (**Tensor**): An int tensor of shape  $(N, \dots)$  containing ground truth labels, and therefore only contain values in the  $[0, n\_classes-1]$  range (except if `ignore_index` is specified).

As output to `forward` and `compute` the metric returns the following output:

- **mc\_aucroc** (**Tensor**): If `average=None|'none'` then a 1d tensor of shape  $(n\_classes, )$  will be returned with aucroc score per class. If `average='macro'|'weighted'` then a single scalar is returned.

Additional dimension  $\dots$  will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the `thresholds` argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the `thresholds` argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

### Parameters

- **num\_classes** (**int**) – Integer specifying the number of classes
- **average** (**Optional**[**Literal**['macro', 'weighted', 'none']]) – Defines the reduction that is applied over classes. Should be one of the following:
  - `macro`: Calculate score for each class and average them
  - `weighted`: calculates score for each class and computes weighted average using their support
  - `"none"` or `None`: calculates score for each class and applies no reduction
- **thresholds** (**Union**[**int**, **List**[**float**], **Tensor**, **None**]) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassAUROC
>>> preds = tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                 [0.05, 0.75, 0.05, 0.05, 0.05],
...                 [0.05, 0.05, 0.75, 0.05, 0.05],
...                 [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = tensor([0, 1, 3, 2])
>>> metric = MulticlassAUROC(num_classes=5, average="macro", thresholds=None)
>>> metric(preds, target)
tensor(0.5333)
>>> mc_aucroc = MulticlassAUROC(num_classes=5, average=None, thresholds=None)
>>> mc_aucroc(preds, target)
tensor([1.0000, 1.0000, 0.3333, 0.3333, 0.0000])
>>> mc_aucroc = MulticlassAUROC(num_classes=5, average="macro", thresholds=5)
>>> mc_aucroc(preds, target)
tensor(0.5333)
>>> mc_aucroc = MulticlassAUROC(num_classes=5, average=None, thresholds=5)
>>> mc_aucroc(preds, target)
tensor([1.0000, 1.0000, 0.3333, 0.3333, 0.0000])
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (*Union[Tensor, Sequence[Tensor], None]*) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (*Optional[Axes]*) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

*Tuple[Figure, Union[Axes, ndarray]]*

#### Returns

Figure and Axes object

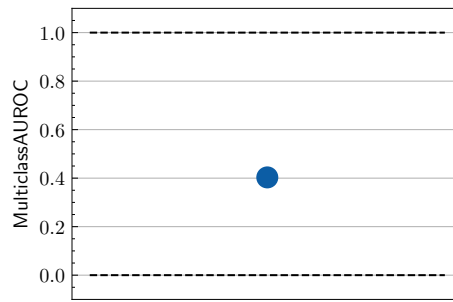
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single
>>> import torch
>>> from torchmetrics.classification import MulticlassAUROC
>>> metric = MulticlassAUROC(num_classes=3)
>>> metric.update(torch.randn(20, 3), torch.randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```

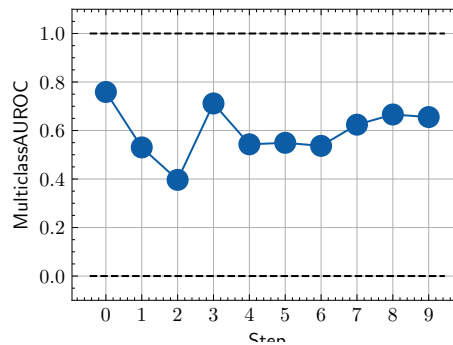
```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.classification import MulticlassAUROC
>>> metric = MulticlassAUROC(num_classes=3)
```

(continues on next page)



(continued from previous page)

```
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.randn(20, 3), torch.randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelAUROC

```
class torchmetrics.classification.MultilabelAUROC(num_labels, average='macro', thresholds=None,
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Compute Area Under the Receiver Operating Characteristic Curve ([ROC AUC](#)) for multilabel tasks.

The AUROC score summarizes the ROC curve into an single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A float tensor of shape  $(N, C, \dots)$  containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** ([Tensor](#)): An int tensor of shape  $(N, C, \dots)$  containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if `ignore_index` is specified).

As output to `forward` and `compute` the metric returns the following output:

- `ml_aucroc` (`Tensor`): If `average=None`|"none" then a 1d tensor of shape  $(n\_classes, )$  will be returned with aucroc score per class. If `average="micro|macro"`|"weighted" then a single scalar is returned.

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the `thresholds` argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the `thresholds` argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

### Parameters

- `num_labels` (`int`) – Integer specifying the number of labels
- `average` (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum score over all labels
  - `macro`: Calculate score for each label and average them
  - `weighted`: calculates score for each label and computes weighted average using their support
  - `"none"` or `None`: calculates score for each label and applies no reduction
- `thresholds` (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- `validate_args` (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- `kwargs` (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelAUROC
>>> preds = tensor([[0.75, 0.05, 0.35],
...                 [0.45, 0.75, 0.05],
...                 [0.05, 0.55, 0.75],
...                 [0.05, 0.65, 0.05]])
>>> target = tensor([[1, 0, 1],
...                  [0, 0, 0],
...                  [0, 1, 1],
...                  [1, 1, 1]])
>>> ml_aucroc = MultilabelAUROC(num_labels=3, average="macro", thresholds=None)
>>> ml_aucroc(preds, target)
```

(continues on next page)



(continued from previous page)

```

tensor(0.6528)
>>> ml_aucroc = MultilabelAUROC(num_labels=3, average=None, thresholds=None)
>>> ml_aucroc(preds, target)
tensor([0.6250, 0.5000, 0.8333])
>>> ml_aucroc = MultilabelAUROC(num_labels=3, average="macro", thresholds=5)
>>> ml_aucroc(preds, target)
tensor(0.6528)
>>> ml_aucroc = MultilabelAUROC(num_labels=3, average=None, thresholds=5)
>>> ml_aucroc(preds, target)
tensor([0.6250, 0.5000, 0.8333])

```

**plot**(val=None, ax=None)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure and Axes object

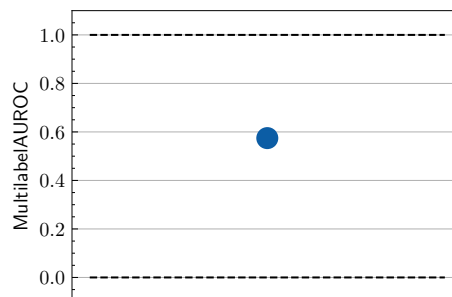
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```

>>> # Example plotting a single
>>> import torch
>>> from torchmetrics.classification import MultilabelAUROC
>>> metric = MultilabelAUROC(num_labels=3)
>>> metric.update(torch.rand(20,3), torch.randint(2, (20,3)))
>>> fig_, ax_ = metric.plot()

```



```

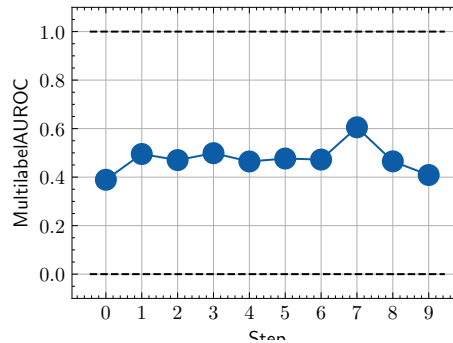
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.classification import MultilabelAUROC

```

(continues on next page)

(continued from previous page)

```
>>> metric = MultilabelAUROC(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.rand(20,3), torch.randint(2, (20,3))))
>>> fig_, ax_ = metric.plot(values)
```



## 1.24.2 Functional Interface

`torchmetrics.functional.auroc(preds, target, task, thresholds=None, num_classes=None, num_labels=None, average='macro', max_fpr=None, ignore_index=None, validate_args=True)`

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC).

The AUROC score summarizes the ROC curve into an single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of `binary_auroc()`, `multiclass_auroc()` and `multilabel_auroc()` for the specific details of each argument influence and examples.

### Return type

`Optional[Tensor]`

### Legacy Example:

```
>>> preds = torch.tensor([0.13, 0.26, 0.08, 0.19, 0.34])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> auroc(preds, target, task='binary')
tensor(0.5000)
```

```
>>> preds = torch.tensor([[0.90, 0.05, 0.05],
...                       [0.05, 0.90, 0.05],
...                       [0.05, 0.05, 0.90],
...                       [0.85, 0.05, 0.10],
...                       [0.10, 0.10, 0.80]])
>>> target = torch.tensor([0, 1, 1, 2, 2])
>>> auroc(preds, target, task='multiclass', num_classes=3)
tensor(0.7778)
```

## binary\_auroc

```
torchmetrics.functional.classification.binary_auroc(preds, target, max_fpr=None, thresholds=None,
                                                    ignore_index=None, validate_args=True)
```

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) for binary tasks.

The AUROC score summarizes the ROC curve into a single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **max\_fpr** (Optional[float]) – If not *None*, calculates standardized partial AUC over the range [0, max\_fpr].
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Return type

Tensor

### Returns

A single scalar with the auroc score

### Example

```
>>> from torchmetrics.functional.classification import binary_auroc
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_auroc(preds, target, thresholds=None)
tensor(0.5000)
>>> binary_auroc(preds, target, thresholds=5)
tensor(0.5000)
```

### multiclass\_auroc

`torchmetrics.functional.classification.multiclass_auroc`(*preds, target, num\_classes, average='macro', thresholds=None, ignore\_index=None, validate\_args=True*)

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) for multiclass tasks.

The AUROC score summarizes the ROC curve into an single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_classes** (int) – Integer specifying the number of classes
- **average** (Optional[Literal['macro', 'weighted', 'none']]) – Defines the reduction that is applied over classes. Should be one of the following:
  - **macro**: Calculate score for each class and average them
  - **weighted**: calculates score for each class and computes weighted average using their support
  - **"none"** or **None**: calculates score for each class and applies no reduction
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.

- If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
- If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
- If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type**`Tensor`**Returns**

If `average=None`|"none" then a 1d tensor of shape (n\_classes, ) will be returned with auroc score per class. If `average="macro"`|"weighted" then a single scalar is returned.

**Example**

```
>>> from torchmetrics.functional.classification import multiclass_auroc
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> multiclass_auroc(preds, target, num_classes=5, average="macro", thresholds=None)
tensor(0.5333)
>>> multiclass_auroc(preds, target, num_classes=5, average=None, thresholds=None)
tensor([1.0000, 1.0000, 0.3333, 0.3333, 0.0000])
>>> multiclass_auroc(preds, target, num_classes=5, average="macro", thresholds=5)
tensor(0.5333)
>>> multiclass_auroc(preds, target, num_classes=5, average=None, thresholds=5)
tensor([1.0000, 1.0000, 0.3333, 0.3333, 0.0000])
```

**multilabel\_auroc**

```
torchmetrics.functional.classification.multilabel_auroc(preds, target, num_labels,
                                                         average='macro', thresholds=None,
                                                         ignore_index=None, validate_args=True)
```

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) for multilabel tasks.

The AUROC score summarizes the ROC curve into a single number that describes the performance of a model for multiple thresholds at the same time. Notably, an AUROC score of 1 is a perfect score and an AUROC score of 0.5 corresponds to random guessing.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.

- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_labels** (int) – Integer specifying the number of labels
- **average** (Optional[Literal['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - micro: Sum score over all labels
  - macro: Calculate score for each label and average them
  - weighted: calculates score for each label and computes weighted average using their support
  - "none" or None: calculates score for each label and applies no reduction
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

#### Return type

Tensor

#### Returns

If *average=None*|"none" then a 1d tensor of shape (n\_classes, ) will be returned with auroc score per class. If *average="micro|macro"*|"weighted" then a single scalar is returned.

### Example

```
>>> from torchmetrics.functional.classification import multilabel_auroc
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> multilabel_auroc(preds, target, num_labels=3, average="macro", thresholds=None)
tensor(0.6528)
>>> multilabel_auroc(preds, target, num_labels=3, average=None, thresholds=None)
tensor([0.6250, 0.5000, 0.8333])
>>> multilabel_auroc(preds, target, num_labels=3, average="macro", thresholds=5)
tensor(0.6528)
>>> multilabel_auroc(preds, target, num_labels=3, average=None, thresholds=5)
tensor([0.6250, 0.5000, 0.8333])
```

## 1.25 Average Precision

### 1.25.1 Module Interface

```
class torchmetrics.AveragePrecision(task: typing_extensions.Literal[binary, multiclass, multilabel],
                                   thresholds: int | List[float] | Tensor | None = None, num_classes: int |
                                   None = None, num_labels: int | None = None, average:
                                   typing_extensions.Literal[macro, weighted, none] | None = 'macro',
                                   ignore_index: int | None = None, validate_args: bool = True,
                                   **kwargs: Any)
```

Compute the average precision (AP) score.

The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum_n (R_n - R_{n-1}) P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of BinaryAveragePrecision, MulticlassAveragePrecision and MultilabelAveragePrecision for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> pred = tensor([0, 0.1, 0.8, 0.4])
>>> target = tensor([0, 1, 1, 1])
>>> average_precision = AveragePrecision(task="binary")
```

(continues on next page)

(continued from previous page)

```
>>> average_precision(pred, target)
tensor(1.)

>>> pred = tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...               [0.05, 0.75, 0.05, 0.05, 0.05],
...               [0.05, 0.05, 0.75, 0.05, 0.05],
...               [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = tensor([0, 1, 3, 2])
>>> average_precision = AveragePrecision(task="multiclass", num_classes=5,
    ↪ average=None)
>>> average_precision(pred, target)
tensor([1.0000, 1.0000, 0.2500, 0.2500, nan])
```

Initialize task metric.

## BinaryAveragePrecision

```
class torchmetrics.classification.BinaryAveragePrecision(thresholds=None, ignore_index=None,
                                                         validate_args=True, **kwargs)
```

Compute the average precision (AP) score for binary tasks.

The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum_n (R_n - R_{n-1}) P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): A float tensor of shape (N, ...) containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (**Tensor**): An int tensor of shape (N, ...) containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

As output to forward and compute the metric returns the following output:

- **bap** (**Tensor**): A single scalar with the average precision score

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

### Parameters

- **thresholds** (**Union[int, List[float], Tensor, None]**) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.



- If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
- If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
- If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryAveragePrecision
>>> preds = tensor([0, 0.5, 0.7, 0.8])
>>> target = tensor([0, 1, 1, 0])
>>> metric = BinaryAveragePrecision(thresholds=None)
>>> metric(preds, target)
tensor(0.5833)
>>> bap = BinaryAveragePrecision(thresholds=5)
>>> bap(preds, target)
tensor(0.6667)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (*Union[Tensor, Sequence[Tensor], None]*) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (*Optional[Axes]*) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

*Tuple[Figure, Union[Axes, ndarray]]*

#### Returns

Figure and Axes object

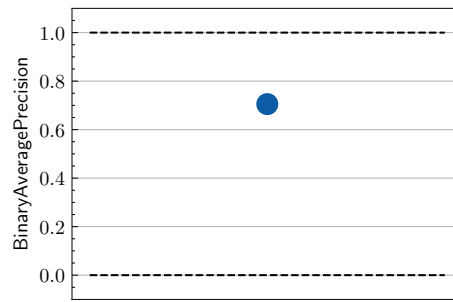
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single
>>> import torch
>>> from torchmetrics.classification import BinaryAveragePrecision
>>> metric = BinaryAveragePrecision()
>>> metric.update(torch.rand(20,), torch.randint(2, (20,)))
>>> fig_, ax_ = metric.plot()
```

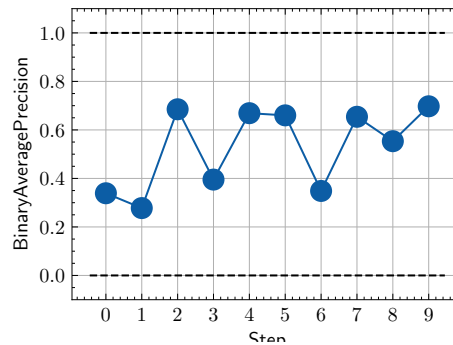
```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.classification import BinaryAveragePrecision
```

(continues on next page)



(continued from previous page)

```
>>> metric = BinaryAveragePrecision()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.rand(20,), torch.randint(2, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassAveragePrecision

```
class torchmetrics.classification.MulticlassAveragePrecision(num_classes, average='macro',
                                                            thresholds=None,
                                                            ignore_index=None,
                                                            validate_args=True, **kwargs)
```

Compute the average precision (AP) score for multiclass tasks.

The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum_n (R_n - R_{n-1}) P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`Tensor`): A float tensor of shape  $(N, C, \dots)$  containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.

- **target** (`Tensor`): An int tensor of shape  $(N, \dots)$  containing ground truth labels, and therefore only contain values in the  $[0, n\_classes-1]$  range (except if *ignore\_index* is specified).

As output to `forward` and `compute` the metric returns the following output:

- **mcap** (`Tensor`): If *average*=`None`|`"none"` then a 1d tensor of shape  $(n\_classes,)$  will be returned with AP score per class. If *average*=`"macro"`|`"weighted"` then a single scalar is returned.

Additional dimension  $\dots$  will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['macro', 'weighted', 'none']]`) – Defines the reduction that is applied over classes. Should be one of the following:
  - `macro`: Calculate score for each class and average them
  - `weighted`: calculates score for each class and computes weighted average using their support
  - `"none"` or `None`: calculates score for each class and applies no reduction
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassAveragePrecision
>>> preds = tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                [0.05, 0.75, 0.05, 0.05, 0.05],
...                [0.05, 0.05, 0.75, 0.05, 0.05],
...                [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = tensor([0, 1, 3, 2])
>>> metric = MulticlassAveragePrecision(num_classes=5, average="macro",
↪ thresholds=None)
>>> metric(preds, target)
```

(continues on next page)

(continued from previous page)

```

tensor(0.6250)
>>> mcap = MulticlassAveragePrecision(num_classes=5, average=None, thresholds=None)
>>> mcap(preds, target)
tensor([1.0000, 1.0000, 0.2500, 0.2500, nan])
>>> mcap = MulticlassAveragePrecision(num_classes=5, average="macro", thresholds=5)
>>> mcap(preds, target)
tensor(0.5000)
>>> mcap = MulticlassAveragePrecision(num_classes=5, average=None, thresholds=5)
>>> mcap(preds, target)
tensor([1.0000, 1.0000, 0.2500, 0.2500, -0.0000])

```

**plot**(val=None, ax=None)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure and Axes object

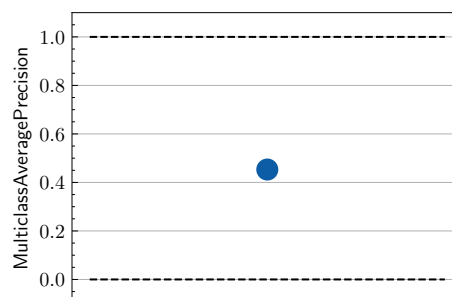
#### Raises

`ModuleNotFoundError` – If *matplotlib* is not installed

```

>>> # Example plotting a single
>>> import torch
>>> from torchmetrics.classification import MulticlassAveragePrecision
>>> metric = MulticlassAveragePrecision(num_classes=3)
>>> metric.update(torch.randn(20, 3), torch.randint(3, (20,)))
>>> fig_, ax_ = metric.plot()

```



```

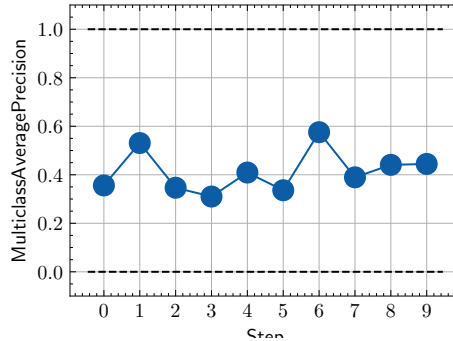
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.classification import MulticlassAveragePrecision

```

(continues on next page)

(continued from previous page)

```
>>> metric = MulticlassAveragePrecision(num_classes=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.randn(20, 3), torch.randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelAveragePrecision

```
class torchmetrics.classification.MultilabelAveragePrecision(num_labels, average='macro',
                                                             thresholds=None,
                                                             ignore_index=None,
                                                             validate_args=True, **kwargs)
```

Compute the average precision (AP) score for multilabel tasks.

The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum_n (R_n - R_{n-1}) P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A float tensor of shape  $(N, C, \dots)$  containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** ([Tensor](#)): An int tensor of shape  $(N, C, \dots)$  containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if `ignore_index` is specified).

As output to `forward` and `compute` the metric returns the following output:

- **mlap** ([Tensor](#)): If `average=None|'none'` then a 1d tensor of shape  $(n\_classes,)$  will be returned with AP score per class. If `average='micro|macro'|'weighted'` then a single scalar is returned.

Additional dimension  $\dots$  will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the `thresholds` argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the `thresholds` argument to either an

integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum score over all labels
  - `macro`: Calculate score for each label and average them
  - `weighted`: calculates score for each label and computes weighted average using their support
  - `"none"` or `None`: calculates score for each label and applies no reduction
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelAveragePrecision
>>> preds = tensor([[0.75, 0.05, 0.35],
...                 [0.45, 0.75, 0.05],
...                 [0.05, 0.55, 0.75],
...                 [0.05, 0.65, 0.05]])
>>> target = tensor([[1, 0, 1],
...                  [0, 0, 0],
...                  [0, 1, 1],
...                  [1, 1, 1]])
>>> metric = MultilabelAveragePrecision(num_labels=3, average="macro",
...                                     thresholds=None)
>>> metric(preds, target)
tensor(0.7500)
>>> mlap = MultilabelAveragePrecision(num_labels=3, average=None, thresholds=None)
>>> mlap(preds, target)
tensor([0.7500, 0.5833, 0.9167])
>>> mlap = MultilabelAveragePrecision(num_labels=3, average="macro", thresholds=5)
>>> mlap(preds, target)
```

(continues on next page)

(continued from previous page)

```

tensor(0.7778)
>>> mlap = MultilabelAveragePrecision(num_labels=3, average=None, thresholds=5)
>>> mlap(preds, target)
tensor([0.7500, 0.6667, 0.9167])

```

**plot**(val=None, ax=None)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union\[Axes, ndarray\]]`

#### Returns

Figure and Axes object

#### Raises

`ModuleNotFoundError` – If *matplotlib* is not installed

```

>>> # Example plotting a single
>>> import torch
>>> from torchmetrics.classification import MultilabelAveragePrecision
>>> metric = MultilabelAveragePrecision(num_labels=3)
>>> metric.update(torch.rand(20,3), torch.randint(2, (20,3)))
>>> fig_, ax_ = metric.plot()

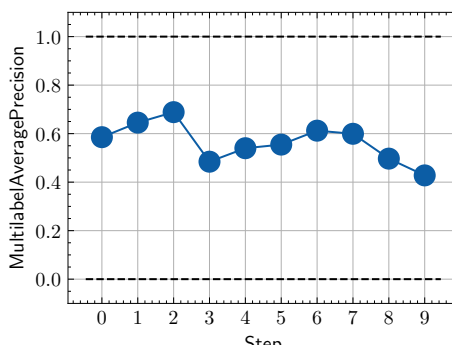
```



```

>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.classification import MultilabelAveragePrecision
>>> metric = MultilabelAveragePrecision(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.rand(20,3), torch.randint(2, (20,3))))
>>> fig_, ax_ = metric.plot(values)

```



### 1.25.2 Functional Interface

`torchmetrics.functional.average_precision(preds, target, task, thresholds=None, num_classes=None, num_labels=None, average='macro', ignore_index=None, validate_args=True)`

Compute the average precision (AP) score.

The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight: `rtype: Optional[Tensor]`

$$AP = \sum n(R_n - R_{n-1})P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either 'binary', 'multiclass' or 'multilabel'. See the documentation of `binary_average_precision()`, `multiclass_average_precision()` and `multilabel_average_precision()` for the specific details of each argument influence and examples.

**Legacy Example:**

```
>>> from torchmetrics.functional.classification import average_precision
>>> pred = torch.tensor([0.0, 1.0, 2.0, 3.0])
>>> target = torch.tensor([0, 1, 1, 1])
>>> average_precision(pred, target, task="binary")
tensor(1.)
```

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> average_precision(pred, target, task="multiclass", num_classes=5,
...                   average=None)
tensor([1.0000, 1.0000, 0.2500, 0.2500, nan])
```



## binary\_average\_precision

```
torchmetrics.functional.classification.binary_average_precision(preds, target, thresholds=None,
                                                                ignore_index=None,
                                                                validate_args=True)
```

Compute the average precision (AP) score for binary tasks.

The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum n(R_n - R_{n-1})P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Return type

[Tensor](#)

### Returns

A single scalar with the average precision score

### Example

```
>>> from torchmetrics.functional.classification import binary_average_precision
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_average_precision(preds, target, thresholds=None)
tensor(0.5833)
>>> binary_average_precision(preds, target, thresholds=5)
tensor(0.6667)
```

### multiclass\_average\_precision

```
torchmetrics.functional.classification.multiclass_average_precision(preds, target, num_classes,
                                                                    average='macro',
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Compute the average precision (AP) score for multiclass tasks.

The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum n(R_n - R_{n-1})P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_classes** (int) – Integer specifying the number of classes
- **average** (Optional[Literal['macro', 'weighted', 'none']]) – Defines the reduction that is applied over classes. Should be one of the following:
  - **macro**: Calculate score for each class and average them
  - **weighted**: calculates score for each class and computes weighted average using their support

- "none" or None: calculates score for each class and applies no reduction
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

**Return type**`Tensor`**Returns**

If *average=None*|"none" then a 1d tensor of shape (n\_classes, ) will be returned with AP score per class. If *average="macro"*|"weighted" then a single scalar is returned.

**Example**

```
>>> from torchmetrics.functional.classification import multiclass_average_precision
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> multiclass_average_precision(preds, target, num_classes=5, average="macro",
↳ thresholds=None)
tensor(0.6250)
>>> multiclass_average_precision(preds, target, num_classes=5, average=None,
↳ thresholds=None)
tensor([1.0000, 1.0000, 0.2500, 0.2500, nan])
>>> multiclass_average_precision(preds, target, num_classes=5, average="macro",
↳ thresholds=5)
tensor(0.5000)
>>> multiclass_average_precision(preds, target, num_classes=5, average=None,
↳ thresholds=5)
tensor([1.0000, 1.0000, 0.2500, 0.2500, -0.0000])
```

## multilabel\_average\_precision

```
torchmetrics.functional.classification.multilabel_average_precision(preds, target, num_labels,
                                                                    average='macro',
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Compute the average precision (AP) score for multilabel tasks.

The AP score summarizes a precision-recall curve as an weighted mean of precisions at each threshold, with the difference in recall from the previous threshold as weight:

$$AP = \sum n(R_n - R_{n-1})P_n$$

where  $P_n, R_n$  is the respective precision and recall at threshold index  $n$ . This value is equivalent to the area under the precision-recall curve (AUPRC).

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **average** ([Optional\[Literal\['micro', 'macro', 'weighted', 'none'\]\]](#)) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum score over all labels
  - **macro**: Calculate score for each label and average them
  - **weighted**: calculates score for each label and computes weighted average using their support
  - **"none"** or **None**: calculates score for each label and applies no reduction
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation

- If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type**`Tensor`**Returns**

If `average=None|'none'` then a 1d tensor of shape (n\_classes, ) will be returned with AP score per class. If `average='micro|macro'|'weighted'` then a single scalar is returned.

**Example**

```
>>> from torchmetrics.functional.classification import multilabel_average_precision
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> multilabel_average_precision(preds, target, num_labels=3, average="macro",
↳ thresholds=None)
tensor(0.7500)
>>> multilabel_average_precision(preds, target, num_labels=3, average=None,
↳ thresholds=None)
tensor([0.7500, 0.5833, 0.9167])
>>> multilabel_average_precision(preds, target, num_labels=3, average="macro",
↳ thresholds=5)
tensor(0.7778)
>>> multilabel_average_precision(preds, target, num_labels=3, average=None,
↳ thresholds=5)
tensor([0.7500, 0.6667, 0.9167])
```

## 1.26 Calibration Error

### 1.26.1 Module Interface

```
class torchmetrics.CalibrationError(task: typing_extensions.Literal[binary, multiclass], n_bins: int = 15,
norm: typing_extensions.Literal[l1, l2, max] = 'l1', num_classes: int |
None = None, ignore_index: int | None = None, validate_args: bool
= True, **kwargs: Any)
```

Top-label Calibration Error.

The expected calibration error can be used to quantify how well a given model is calibrated e.g. how well the predicted output probabilities of the model matches the actual probabilities of the ground truth distribution. Three

different norms are implemented, each corresponding to variations on the calibration error metric.

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|, \text{L1 norm (Expected Calibration Error)}$$

$$\text{MCE} = \max_i (p_i - c_i), \text{Infinity norm (Maximum Calibration Error)}$$

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}, \text{L2 norm (Root Mean Square Calibration Error)}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ . Bins are constructed in an uniform way in the  $[0,1]$  range.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary' or 'multiclass'. See the documentation of `BinaryCalibrationError` and `MulticlassCalibrationError` for the specific details of each argument influence and examples.

Initialize task metric.

## BinaryCalibrationError

```
class torchmetrics.classification.BinaryCalibrationError(n_bins=15, norm='l1',  
                                                         ignore_index=None, validate_args=True,  
                                                         **kwargs)
```

Top-label Calibration Error for binary tasks.

The expected calibration error can be used to quantify how well a given model is calibrated e.g. how well the predicted output probabilities of the model matches the actual probabilities of the ground truth distribution. Three different norms are implemented, each corresponding to variations on the calibration error metric.

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|, \text{L1 norm (Expected Calibration Error)}$$

$$\text{MCE} = \max_i (p_i - c_i), \text{Infinity norm (Maximum Calibration Error)}$$

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}, \text{L2 norm (Root Mean Square Calibration Error)}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ . Bins are constructed in an uniform way in the  $[0,1]$  range.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`Tensor`): A float tensor of shape  $(N, \dots)$  containing probabilities or logits for each observation. If `preds` has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (`Tensor`): An int tensor of shape  $(N, \dots)$  containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if `ignore_index` is specified). The value 1 always encodes the positive class.

As output to `forward` and `compute` the metric returns the following output:

- **bce** (`Tensor`): A scalar tensor containing the calibration error

Additional dimension  $\dots$  will be flattened into the batch dimension.

### Parameters

- **n\_bins** (`int`) – Number of bins to use when computing the metric.
- **norm** (`Literal['l1', 'l2', 'max']`) – Norm used to compare empirical and expected probability bins.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryCalibrationError
>>> preds = tensor([0.25, 0.25, 0.55, 0.75, 0.75])
>>> target = tensor([0, 0, 1, 1, 1])
>>> metric = BinaryCalibrationError(n_bins=2, norm='l1')
>>> metric(preds, target)
tensor(0.2900)
>>> bce = BinaryCalibrationError(n_bins=2, norm='l2')
>>> bce(preds, target)
tensor(0.2918)
>>> bce = BinaryCalibrationError(n_bins=2, norm='max')
>>> bce(preds, target)
tensor(0.3167)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

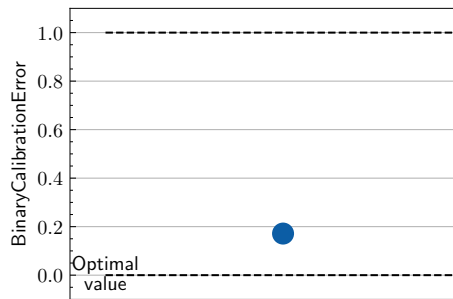
#### Returns

Figure object and Axes object

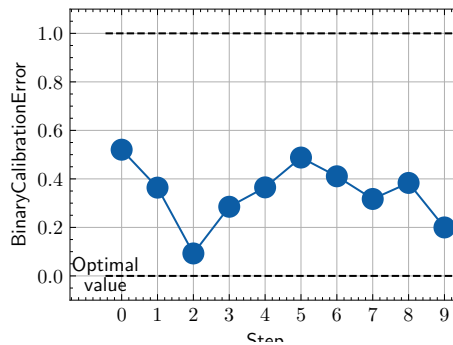
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinaryCalibrationError
>>> metric = BinaryCalibrationError(n_bins=2, norm='l1')
>>> metric.update(rand(10), randint(2, (10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryCalibrationError
>>> metric = BinaryCalibrationError(n_bins=2, norm='l1')
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassCalibrationError

```
class torchmetrics.classification.MulticlassCalibrationError(num_classes, n_bins=15, norm='l1',
                                                            ignore_index=None,
                                                            validate_args=True, **kwargs)
```

Top-label Calibration Error for multiclass tasks.

The expected calibration error can be used to quantify how well a given model is calibrated e.g. how well the predicted output probabilities of the model matches the actual probabilities of the ground truth distribution. Three different norms are implemented, each corresponding to variations on the calibration error metric.

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|, \text{L1 norm (Expected Calibration Error)}$$

$$\text{MCE} = \max_i (p_i - c_i), \text{Infinity norm (Maximum Calibration Error)}$$



$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}, \text{L2 norm (Root Mean Square Calibration Error)}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ . Bins are constructed in an uniform way in the  $[0,1]$  range.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`Tensor`): A float tensor of shape  $(N, C, \dots)$  containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** (`Tensor`): An int tensor of shape  $(N, \dots)$  containing ground truth labels, and therefore only contain values in the  $[0, n\_classes-1]$  range (except if `ignore_index` is specified).

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **mcce** (`Tensor`): A scalar tensor containing the calibration error

#### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **n\_bins** (`int`) – Number of bins to use when computing the metric.
- **norm** (`Literal['l1', 'l2', 'max']`) – Norm used to compare empirical and expected probability bins.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassCalibrationError
>>> preds = tensor([[0.25, 0.20, 0.55],
...                 [0.55, 0.05, 0.40],
...                 [0.10, 0.30, 0.60],
...                 [0.90, 0.05, 0.05]])
>>> target = tensor([0, 1, 2, 0])
>>> metric = MulticlassCalibrationError(num_classes=3, n_bins=3, norm='l1')
>>> metric(preds, target)
tensor(0.2000)
>>> mcce = MulticlassCalibrationError(num_classes=3, n_bins=3, norm='l2')
>>> mcce(preds, target)
tensor(0.2082)
>>> mcce = MulticlassCalibrationError(num_classes=3, n_bins=3, norm='max')
>>> mcce(preds, target)
tensor(0.2333)
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

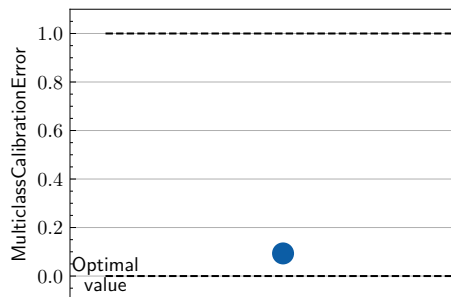
#### Returns

Figure object and Axes object

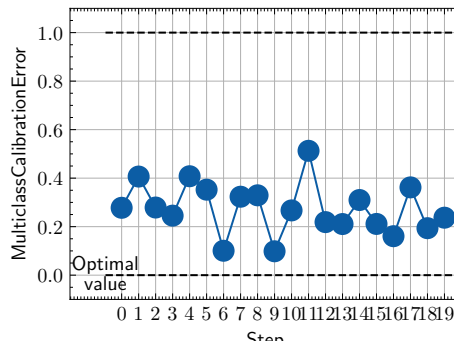
#### Raises

`ModuleNotFoundError` – If `matplotlib` is not installed

```
>>> from torch import randn, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MulticlassCalibrationError
>>> metric = MulticlassCalibrationError(num_classes=3, n_bins=3, norm='l1')
>>> metric.update(randn(20,3).softmax(dim=-1), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randn, randint
>>> # Example plotting a multiple values
>>> from torchmetrics.classification import MulticlassCalibrationError
>>> metric = MulticlassCalibrationError(num_classes=3, n_bins=3, norm='l1')
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randn(20,3).softmax(dim=-1), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.26.2 Functional Interface

`torchmetrics.functional.calibration_error(preds, target, task, n_bins=15, norm='l1', num_classes=None, ignore_index=None, validate_args=True)`

Top-label Calibration Error.

The expected calibration error can be used to quantify how well a given model is calibrated e.g. how well the predicted output probabilities of the model matches the actual probabilities of the ground truth distribution. Three different norms are implemented, each corresponding to variations on the calibration error metric. :rtype: `Tensor`

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|, \text{L1 norm (Expected Calibration Error)}$$

$$\text{MCE} = \max_i (p_i - c_i), \text{Infinity norm (Maximum Calibration Error)}$$

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}, \text{L2 norm (Root Mean Square Calibration Error)}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ . Bins are constructed in a uniform way in the  $[0,1]$  range.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either `'binary'` or `'multiclass'`. See the documentation of `binary_calibration_error()` and `multiclass_calibration_error()` for the specific details of each argument influence and examples.

#### `binary_calibration_error`

`torchmetrics.functional.classification.binary_calibration_error(preds, target, n_bins=15, norm='l1', ignore_index=None, validate_args=True)`

Top-label Calibration Error for binary tasks.

The expected calibration error can be used to quantify how well a given model is calibrated e.g. how well the predicted output probabilities of the model matches the actual probabilities of the ground truth distribution. Three different norms are implemented, each corresponding to variations on the calibration error metric.

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|, \text{L1 norm (Expected Calibration Error)}$$

$$\text{MCE} = \max_i (p_i - c_i), \text{Infinity norm (Maximum Calibration Error)}$$

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}, \text{L2 norm (Root Mean Square Calibration Error)}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ . Bins are constructed in an uniform way in the  $[0,1]$  range.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **n\_bins** (int) – Number of bins to use when computing the metric.
- **norm** (Literal['l1', 'l2', 'max']) – Norm used to compare empirical and expected probability bins.
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

#### Return type

Tensor

#### Example

```
>>> from torchmetrics.functional.classification import binary_calibration_error
>>> preds = torch.tensor([0.25, 0.25, 0.55, 0.75, 0.75])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> binary_calibration_error(preds, target, n_bins=2, norm='l1')
tensor(0.2900)
>>> binary_calibration_error(preds, target, n_bins=2, norm='l2')
tensor(0.2918)
>>> binary_calibration_error(preds, target, n_bins=2, norm='max')
tensor(0.3167)
```

## multiclass\_calibration\_error

```
torchmetrics.functional.classification.multiclass_calibration_error(preds, target, num_classes,
                                                                    n_bins=15, norm='l1',
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Top-label Calibration Error for multiclass tasks.

The expected calibration error can be used to quantify how well a given model is calibrated e.g. how well the predicted output probabilities of the model matches the actual probabilities of the ground truth distribution. Three different norms are implemented, each corresponding to variations on the calibration error metric.

$$\text{ECE} = \sum_i^N b_i \| (p_i - c_i) \|, \text{L1 norm (Expected Calibration Error)}$$

$$\text{MCE} = \max_i (p_i - c_i), \text{Infinity norm (Maximum Calibration Error)}$$

$$\text{RMSCE} = \sqrt{\sum_i^N b_i (p_i - c_i)^2}, \text{L2 norm (Root Mean Square Calibration Error)}$$

Where  $p_i$  is the top-1 prediction accuracy in bin  $i$ ,  $c_i$  is the average confidence of predictions in bin  $i$ , and  $b_i$  is the fraction of data points in bin  $i$ . Bins are constructed in an uniform way in the  $[0,1]$  range.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the  $[0, \text{n\_classes}-1]$  range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **n\_bins** ([int](#)) – Number of bins to use when computing the metric.
- **norm** ([Literal](#)['l1', 'l2', 'max']) – Norm used to compare empirical and expected probability bins.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Return type

[Tensor](#)

### Example

```
>>> from torchmetrics.functional.classification import multiclass_calibration_error
>>> preds = torch.tensor([[0.25, 0.20, 0.55],
...                       [0.55, 0.05, 0.40],
...                       [0.10, 0.30, 0.60],
...                       [0.90, 0.05, 0.05]])
>>> target = torch.tensor([0, 1, 2, 0])
>>> multiclass_calibration_error(preds, target, num_classes=3, n_bins=3, norm='l1')
tensor(0.2000)
>>> multiclass_calibration_error(preds, target, num_classes=3, n_bins=3, norm='l2')
tensor(0.2082)
>>> multiclass_calibration_error(preds, target, num_classes=3, n_bins=3, norm='max')
tensor(0.2333)
```

## 1.27 Cohen Kappa

### 1.27.1 Module Interface

#### CohenKappa

```
class torchmetrics.CohenKappa(task: typing_extensions.Literal[binary, multiclass], threshold: float = 0.5,
                              num_classes: int | None = None, weights: typing_extensions.Literal[linear,
                              quadratic, none] | None = None, ignore_index: int | None = None,
                              validate_args: bool = True, **kwargs: Any)
```

Calculate Cohen's kappa score that measures inter-annotator agreement.

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary' or 'multiclass'. See the documentation of `BinaryCohenKappa` and `MulticlassCohenKappa` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> cohenkappa = CohenKappa(task="multiclass", num_classes=2)
>>> cohenkappa(preds, target)
tensor(0.5000)
```

Initialize task metric.

## BinaryCohenKappa

```
class torchmetrics.classification.BinaryCohenKappa(threshold=0.5, ignore_index=None,
                                                    weights=None, validate_args=True, **kwargs)
```

Calculate [Cohen's kappa score](#) that measures inter-annotator agreement for binary tasks.

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A int or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** ([Tensor](#)): An int tensor of shape (N, ...).

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **bck** ([Tensor](#)): A tensor containing cohen kappa score

### Parameters

- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **weights** ([Optional\[Literal\['linear', 'quadratic', 'none'\]\]](#)) – Weighting type to calculate the score. Choose from:
  - None or 'none': no weighting
  - 'linear': linear weighting
  - 'quadratic': quadratic weighting
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryCohenKappa
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> metric = BinaryCohenKappa()
>>> metric(preds, target)
tensor(0.5000)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinaryCohenKappa
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0.35, 0.85, 0.48, 0.01])
>>> metric = BinaryCohenKappa()
>>> metric(preds, target)
tensor(0.5000)
```

**plot**(*val=None*, *ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

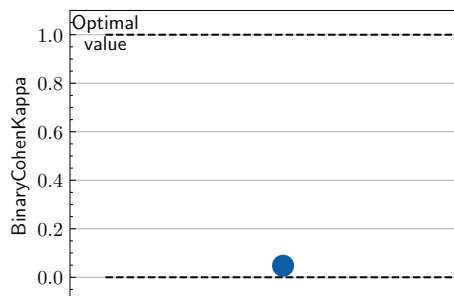
#### Returns

Figure object and Axes object

#### Raises

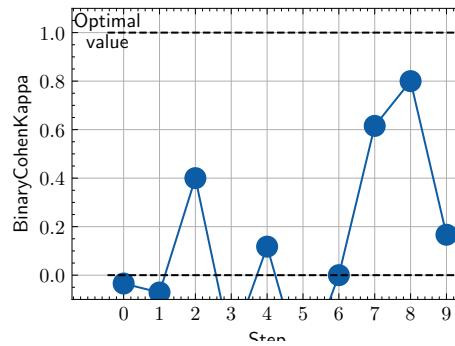
**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinaryCohenKappa
>>> metric = BinaryCohenKappa()
>>> metric.update(rand(10), randint(2, (10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryCohenKappa
>>> metric = BinaryCohenKappa()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2, (10,))))
>>> fig_, ax_ = metric.plot(values)
```





## MulticlassCohenKappa

```
class torchmetrics.classification.MulticlassCohenKappa(num_classes, ignore_index=None,
                                                         weights=None, validate_args=True,
                                                         **kwargs)
```

Calculate [Cohen's kappa score](#) that measures inter-annotator agreement for multiclass tasks.

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

As input to `forward` and `update` the metric accepts the following input:

- `preds` ([Tensor](#)): Either an int tensor of shape  $(N, \dots)$  or float tensor of shape  $((N, C, \dots))$ . If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- `target` ([Tensor](#)): An int tensor of shape  $(N, \dots)$ .

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- `mcck` ([Tensor](#)): A tensor containing cohen kappa score

### Parameters

- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **weights** ([Optional\[Literal\['linear', 'quadratic', 'none'\]\]](#)) – Weighting type to calculate the score. Choose from:
  - None or 'none': no weighting
  - 'linear': linear weighting
  - 'quadratic': quadratic weighting
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

Example (pred is integer tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassCohenKappa
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassCohenKappa(num_classes=3)
>>> metric(preds, target)
tensor(0.6364)
```

Example (pred is float tensor):

```
>>> from torchmetrics.classification import MulticlassCohenKappa
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> metric = MulticlassCohenKappa(num_classes=3)
>>> metric(preds, target)
tensor(0.6364)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (*Union[Tensor, Sequence[Tensor], None]*) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (*Optional[Axes]*) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

*Tuple[Figure, Union[Axes, ndarray]]*

#### Returns

Figure object and Axes object

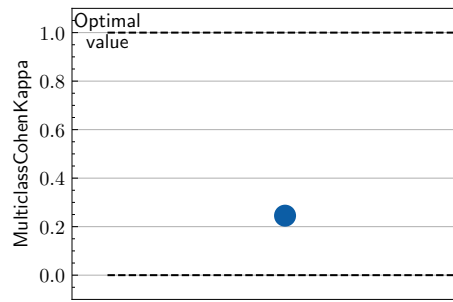
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randn, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MulticlassCohenKappa
>>> metric = MulticlassCohenKappa(num_classes=3)
>>> metric.update(randn(20,3).softmax(dim=-1), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```

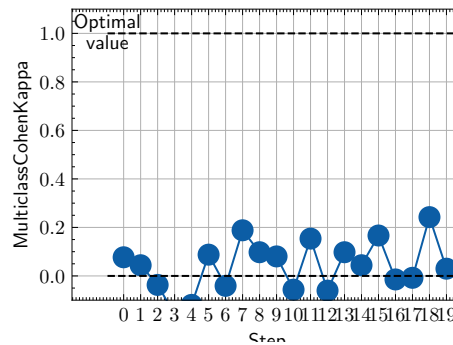
```
>>> from torch import randn, randint
>>> # Example plotting a multiple values
>>> from torchmetrics.classification import MulticlassCohenKappa
>>> metric = MulticlassCohenKappa(num_classes=3)
>>> values = []
```

(continues on next page)



(continued from previous page)

```
>>> for _ in range(20):
...     values.append(metric(randn(20,3).softmax(dim=-1), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## 1.27.2 Functional Interface

### cohen\_kappa

`torchmetrics.functional.cohen_kappa(preds, target, task, threshold=0.5, num_classes=None, weights=None, ignore_index=None, validate_args=True)`

Calculate [Cohen's kappa score](#) that measures inter-annotator agreement. It is defined as. :rtype: `Tensor`

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either `'binary'` or `'multiclass'`. See the documentation of `binary_cohen_kappa()` and `multiclass_cohen_kappa()` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([1, 1, 0, 0])
```

(continues on next page)

(continued from previous page)

```
>>> preds = tensor([0, 1, 0, 0])
>>> cohen_kappa(preds, target, task="multiclass", num_classes=2)
tensor(0.5000)
```

## binary\_cohen\_kappa

`torchmetrics.functional.classification.binary_cohen_kappa(preds, target, threshold=0.5, weights=None, ignore_index=None, validate_args=True)`

Calculate [Cohen's kappa score](#) that measures inter-annotator agreement for binary tasks.

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **weights** ([Optional\[Literal\['linear', 'quadratic', 'none'\]\]](#)) – Weighting type to calculate the score. Choose from:
  - None or 'none': no weighting
  - 'linear': linear weighting
  - 'quadratic': quadratic weighting
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Return type

[Tensor](#)

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_cohen_kappa
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> binary_cohen_kappa(preds, target)
tensor(0.5000)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import binary_cohen_kappa
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0.35, 0.85, 0.48, 0.01])
>>> binary_cohen_kappa(preds, target)
tensor(0.5000)
```

## multiclass\_cohen\_kappa

`torchmetrics.functional.classification.multiclass_cohen_kappa(preds, target, num_classes, weights=None, ignore_index=None, validate_args=True)`

Calculate Cohen's kappa score that measures inter-annotator agreement for multiclass tasks.

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where  $p_o$  is the empirical probability of agreement and  $p_e$  is the expected agreement when both annotators assign labels randomly. Note that  $p_e$  is estimated using a per-annotator empirical prior over the class labels.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **weights** ([Optional\[Literal\['linear', 'quadratic', 'none'\]\]](#)) – Weighting type to calculate the score. Choose from:
  - None or 'none': no weighting
  - 'linear': linear weighting
  - 'quadratic': quadratic weighting
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Return type**

Tensor

**Example (pred is integer tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_cohen_kappa
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_cohen_kappa(preds, target, num_classes=3)
tensor(0.6364)
```

**Example (pred is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_cohen_kappa
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> multiclass_cohen_kappa(preds, target, num_classes=3)
tensor(0.6364)
```

## 1.28 Confusion Matrix

### 1.28.1 Module Interface

#### ConfusionMatrix

```
class torchmetrics.ConfusionMatrix(task: typing_extensions.Literal[binary, multiclass, multilabel],
                                   threshold: float = 0.5, num_classes: int | None = None, num_labels:
                                   int | None = None, normalize: typing_extensions.Literal[true, pred, all,
                                   none] | None = None, ignore_index: int | None = None, validate_args:
                                   bool = True, **kwargs: Any)
```

Compute the [confusion matrix](#).

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of `BinaryConfusionMatrix`, `MulticlassConfusionMatrix` and `MultilabelConfusionMatrix()` for the specific details of each argument influence and examples.

**Legacy Example:**

```
>>> from torch import tensor
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> confmat = ConfusionMatrix(task="binary", num_classes=2)
>>> confmat(preds, target)
tensor([[2, 0],
        [1, 1]])
```

```
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> confmat = ConfusionMatrix(task="multiclass", num_classes=3)
>>> confmat(preds, target)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

```
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> confmat = ConfusionMatrix(task="multilabel", num_labels=3)
>>> confmat(preds, target)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

Initialize task metric.

## BinaryConfusionMatrix

```
class torchmetrics.classification.BinaryConfusionMatrix(threshold=0.5, ignore_index=None,
                                                         normalize=None, validate_args=True,
                                                         **kwargs)
```

Compute the [confusion matrix](#) for binary tasks.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** ([Tensor](#)): An int tensor of shape (N, ...).

As output to forward and compute the metric returns the following output:

- **confusion\_matrix** ([Tensor](#)): A tensor containing a (2, 2) matrix

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** ([Optional\[Literal\['true', 'pred', 'all', 'none'\]\]](#)) – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torchmetrics.classification import BinaryConfusionMatrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0, 1, 0, 0])
>>> bcm = BinaryConfusionMatrix()
>>> bcm(preds, target)
tensor([[2, 0],
        [1, 1]])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryConfusionMatrix
>>> target = torch.tensor([1, 1, 0, 0])
>>> preds = torch.tensor([0.35, 0.85, 0.48, 0.01])
>>> bcm = BinaryConfusionMatrix()
>>> bcm(preds, target)
tensor([[2, 0],
        [1, 1]])
```

**plot** (*val=None, ax=None, add\_text=True, labels=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (*Optional[Tensor]*) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (*Optional[Axes]*) – An matplotlib axis object. If provided will add plot to that axis
- **add\_text** (*bool*) – if the value of each cell should be added to the plot
- **labels** (*Optional[List[str]]*) – a list of strings, if provided will be added to the plot to indicate the different classes

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

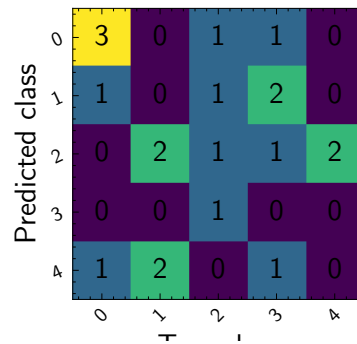
Figure and Axes object

#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randint
>>> from torchmetrics.classification import MulticlassConfusionMatrix
>>> metric = MulticlassConfusionMatrix(num_classes=5)
>>> metric.update(randint(5, (20,)), randint(5, (20,)))
>>> fig_, ax_ = metric.plot()
```





## MulticlassConfusionMatrix

**class** torchmetrics.classification.**MulticlassConfusionMatrix**(*num\_classes*, *ignore\_index=None*, *normalize=None*, *validate\_args=True*, *\*\*kwargs*)

Compute the [confusion matrix](#) for multiclass tasks.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** ([Tensor](#)): An int tensor of shape (N, ...).

As output to forward and compute the metric returns the following output:

- **confusion\_matrix**: [num\_classes, num\_classes] matrix

### Parameters

- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** ([Optional\[Literal\['true', 'pred', 'all', 'none'\]\]](#)) – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (pred is integer tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassConfusionMatrix
```

(continues on next page)

(continued from previous page)

```
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassConfusionMatrix(num_classes=3)
>>> metric(preds, target)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

**Example (pred is float tensor):**

```
>>> from torchmetrics.classification import MulticlassConfusionMatrix
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> metric = MulticlassConfusionMatrix(num_classes=3)
>>> metric(preds, target)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

**plot**(*val=None, ax=None, add\_text=True, labels=None*)

Plot a single or multiple values from the metric.

**Parameters**

- **val** (*Optional*[Tensor]) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (*Optional*[Axes]) – An matplotlib axis object. If provided will add plot to that axis
- **add\_text** (*bool*) – if the value of each cell should be added to the plot
- **labels** (*Optional*[List[str]]) – a list of strings, if provided will be added to the plot to indicate the different classes

**Return type**

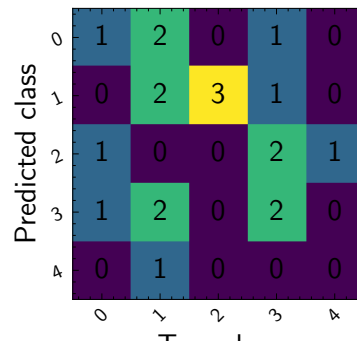
Tuple[Figure, Union[Axes, ndarray]]

**Returns**

Figure and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randint
>>> from torchmetrics.classification import MulticlassConfusionMatrix
>>> metric = MulticlassConfusionMatrix(num_classes=5)
>>> metric.update(randint(5, (20,)), randint(5, (20,)))
>>> fig_, ax_ = metric.plot()
```



## MultilabelConfusionMatrix

```
class torchmetrics.classification.MultilabelConfusionMatrix(num_labels, threshold=0.5,
                                                             ignore_index=None,
                                                             normalize=None, validate_args=True,
                                                             **kwargs)
```

Compute the [confusion matrix](#) for multilabel tasks.

As input to ‘update’ the metric accepts the following input:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

As output of ‘compute’ the metric returns the following output:

- **confusion matrix**: [num\_labels,2,2] matrix

### Parameters

- **num\_classes** – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **normalize** ([Optional\[Literal\['true', 'pred', 'all', 'none'\]\]](#)) – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelConfusionMatrix
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelConfusionMatrix(num_labels=3)
>>> metric(preds, target)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelConfusionMatrix
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelConfusionMatrix(num_labels=3)
>>> metric(preds, target)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

**plot**(*val=None, ax=None, add\_text=True, labels=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (*Optional*[Tensor]) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (*Optional*[Axes]) – An matplotlib axis object. If provided will add plot to that axis
- **add\_text** (*bool*) – if the value of each cell should be added to the plot
- **labels** (*Optional*[List[str]]) – a list of strings, if provided will be added to the plot to indicate the different classes

#### Return type

Tuple[Figure, Union[Axes, ndarray]]

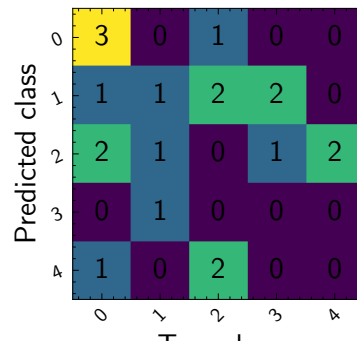
#### Returns

Figure and Axes object

#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randint
>>> from torchmetrics.classification import MulticlassConfusionMatrix
>>> metric = MulticlassConfusionMatrix(num_classes=5)
>>> metric.update(randint(5, (20,)), randint(5, (20,)))
>>> fig_, ax_ = metric.plot()
```



## 1.28.2 Functional Interface

### confusion\_matrix

`torchmetrics.functional.confusion_matrix(preds, target, task, threshold=0.5, num_classes=None, num_labels=None, normalize=None, ignore_index=None, validate_args=True)`

Compute the [confusion matrix](#).

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of `binary_confusion_matrix()`, `multiclass_confusion_matrix()` and `multilabel_confusion_matrix()` for the specific details of each argument influence and examples.

#### Return type

Tensor

#### Legacy Example:

```
>>> from torch import tensor
>>> from torchmetrics.classification import ConfusionMatrix
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> confmat = ConfusionMatrix(task="binary")
>>> confmat(preds, target)
tensor([[2, 0],
        [1, 1]])

>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> confmat = ConfusionMatrix(task="multiclass", num_classes=3)
>>> confmat(preds, target)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])

>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> confmat = ConfusionMatrix(task="multilabel", num_labels=3)
>>> confmat(preds, target)
```

(continues on next page)

(continued from previous page)

```
tensor([[[[1, 0], [0, 1]],
         [[1, 0], [1, 0]],
         [[0, 1], [0, 1]]]])
```

## binary\_confusion\_matrix

`torchmetrics.functional.classification.binary_confusion_matrix(preds, target, threshold=0.5, normalize=None, ignore_index=None, validate_args=True)`

Compute the [confusion matrix](#) for binary tasks.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **normalize** ([Optional](#)[[Literal](#)['true', 'pred', 'all', 'none']]) – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

### Return type

[Tensor](#)

### Returns

A [2, 2] tensor

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_confusion_matrix
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
```

(continues on next page)

(continued from previous page)

```
>>> binary_confusion_matrix(preds, target)
tensor([[2, 0],
        [1, 1]])
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import binary_confusion_matrix
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0.35, 0.85, 0.48, 0.01])
>>> binary_confusion_matrix(preds, target)
tensor([[2, 0],
        [1, 1]])
```

### multiclass\_confusion\_matrix

`torchmetrics.functional.classification.multiclass_confusion_matrix(preds, target, num_classes, normalize=None, ignore_index=None, validate_args=True)`

Compute the [confusion matrix](#) for multiclass tasks.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **normalize** ([Optional\[Literal\['true', 'pred', 'all', 'none'\]\]](#)) – Normalization mode for confusion matrix. Choose from:
  - None or 'none': no normalization (default)
  - 'true': normalization over the targets (most commonly used)
  - 'pred': normalization over the predictions
  - 'all': normalization over the whole matrix
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Return type

[Tensor](#)

#### Returns

A [num\_classes, num\_classes] tensor

Example (pred is integer tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_confusion_
    ↪matrix
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_confusion_matrix(preds, target, num_classes=3)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

Example (pred is float tensor):

```
>>> from torchmetrics.functional.classification import multiclass_confusion_
    ↪matrix
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> multiclass_confusion_matrix(preds, target, num_classes=3)
tensor([[1, 1, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

## multilabel\_confusion\_matrix

`torchmetrics.functional.classification.multilabel_confusion_matrix`(*preds, target, num\_labels, threshold=0.5, normalize=None, ignore\_index=None, validate\_args=True*)

Compute the `confusion matrix` for multilabel tasks.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): (N, C, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **normalize** (`Optional[Literal['true', 'pred', 'all', 'none']]`) – Normalization mode for confusion matrix. Choose from:



- None or 'none': no normalization (default)
- 'true': normalization over the targets (most commonly used)
- 'pred': normalization over the predictions
- 'all': normalization over the whole matrix
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type**`Tensor`**Returns**A `[num_labels, 2, 2]` tensor**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_confusion_
    ↪matrix
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_confusion_matrix(preds, target, num_labels=3)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_confusion_
    ↪matrix
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_confusion_matrix(preds, target, num_labels=3)
tensor([[[1, 0], [0, 1]],
        [[1, 0], [1, 0]],
        [[0, 1], [0, 1]]])
```

## 1.29 Coverage Error

### 1.29.1 Module Interface

```
class torchmetrics.classification.MultilabelCoverageError(num_labels, ignore_index=None,
                                                         validate_args=True, **kwargs)
```

Compute `Multilabel coverage error`.

The score measure how far we need to go through the ranked scores to cover all true labels. The best value is equal to the average number of labels in the target tensor per sample.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`Tensor`): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (`Tensor`): An int tensor of shape  $(N, C, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if *ignore\_index* is specified).

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to forward and compute the metric returns the following output:

- **mlce** (`Tensor`): A tensor containing the multilabel coverage error.

#### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Example

```
>>> from torchmetrics.classification import MultilabelCoverageError
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> mlce = MultilabelCoverageError(num_labels=5)
>>> mlce(preds, target)
tensor(3.9000)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure object and Axes object

#### Raises

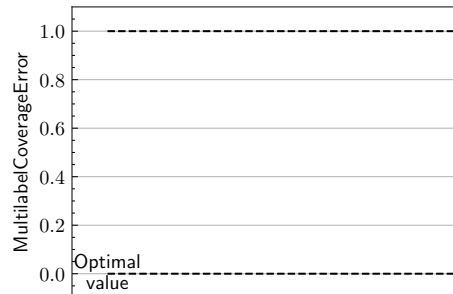
`ModuleNotFoundError` – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelCoverageError
>>> metric = MultilabelCoverageError(num_labels=3)
```

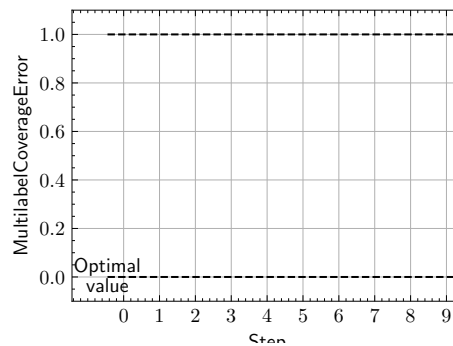
(continues on next page)

(continued from previous page)

```
>>> metric.update(rand(20, 3), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import MultilabelCoverageError
>>> metric = MultilabelCoverageError(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(20, 3), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```



## 1.29.2 Functional Interface

`torchmetrics.functional.classification.multilabel_coverage_error(preds, target, num_labels, ignore_index=None, validate_args=True)`

Compute multilabel coverage error [1].

The score measure how far we need to go through the ranked scores to cover all true labels. The best value is equal to the average number of labels in the target tensor per sample.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply

sigmoid per element.

- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_labels** (int) – Integer specifying the number of labels
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

#### Return type

Tensor

#### Example

```
>>> from torchmetrics.functional.classification import multilabel_coverage_error
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> multilabel_coverage_error(preds, target, num_labels=5)
tensor(3.9000)
```

#### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

## 1.30 Dice

### 1.30.1 Module Interface

```
class torchmetrics.Dice(zero_division=0, num_classes=None, threshold=0.5, average='micro',
                        mdmc_average='global', ignore_index=None, top_k=None, multiclass=None,
                        **kwargs)
```

Compute [Dice](#).

$$\text{Dice} = \frac{2 * \text{TP}}{2 * \text{TP} + \text{FP} + \text{FN}}$$

Where TP and FP represent the number of true positives and false positives respectively.

It is recommend set *ignore\_index* to index of background class.

The reduction method (how the precision scores are aggregated) is controlled by the *average* parameter, and additionally by the *mdmc\_average* parameter in the multi-dimensional multi-class case.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): Predictions from model (probabilities, logits or labels)
- **target** ([Tensor](#)): Ground truth values

As output to forward and compute the metric returns the following output:

- **dice** ([Tensor](#)): A tensor containing the dice score.
  - If **average** in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned
  - If **average** in ['none', None], the shape will be (C, ), where C stands for the number of classes

### Parameters

- **num\_classes** – Number of classes. Necessary for 'macro', and None average methods.
- **threshold** – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **zero\_division** – The value to use for the score if denominator equals zero.
- **average** – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (**tp** + **fn**).
  - 'none' or None: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of **mdmc\_average**.

---

- **mdmc\_average** – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the **average** parameter). Should be one of the following:
  - None [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the N axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... as the N dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the N and ... dimensions of the inputs are flattened into a new N\_X sample axis, i.e. the inputs are treated as if they were (N\_X, C). From here on the **average** parameter applies as usual.
- **ignore\_index** – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and **average=None** or 'none', the score for the ignored class will be returned as nan.

- **top\_k** – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (None) will be interpreted as 1 for these inputs. Should be left at default (None) for all other types of inputs.
- **multiclass** – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises**

- **ValueError** – If average is none of "micro", "macro", "samples", "none", None.
- **ValueError** – If mdmc\_average is not one of None, "samplewise", "global".
- **ValueError** – If average is set but num\_classes is not provided.
- **ValueError** – If num\_classes is set and ignore\_index is not in the range [0, num\_classes).

**Example**

```
>>> from torch import tensor
>>> from torchmetrics.classification import Dice
>>> preds = tensor([2, 0, 2, 1])
>>> target = tensor([1, 1, 2, 0])
>>> dice = Dice(average='micro')
>>> dice(preds, target)
tensor(0.2500)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

**Parameters**

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union\[Axes, ndarray\]]`

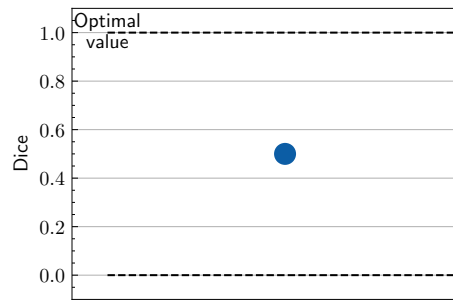
**Returns**

Figure object and Axes object

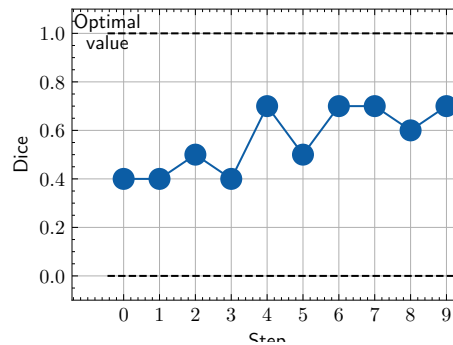
**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> from torch import randint
>>> from torchmetrics.classification import Dice
>>> metric = Dice()
>>> metric.update(randint(2, (10,)), randint(2, (10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torch import randint
>>> from torchmetrics.classification import Dice
>>> metric = Dice()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2,(10,)), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.30.2 Functional Interface

`torchmetrics.functional.dice(preds, target, zero_division=0, average='micro', mdmc_average='global', threshold=0.5, top_k=None, num_classes=None, multiclass=None, ignore_index=None)`

Compute [Dice](#).

$$\text{Dice} = \frac{2 * \text{TP}}{2 * \text{TP} + \text{FP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

It is recommend set `ignore_index` to index of background class.

The reduction method (how the recall scores are aggregated) is controlled by the `average` parameter, and additionally by the `mdmc_average` parameter in the multi-dimensional multi-class case.

#### Parameters

- **preds** (`Tensor`) – Predictions from model (probabilities, logits or labels)
- **target** (`Tensor`) – Ground truth values
- **zero\_division** (`int`) – The value to use for the score if denominator equals zero
- **average** (`Optional[str]`) – Defines the reduction that is applied. Should be one of the following:
  - 'micro' [default]: Calculate the metric globally, across all samples and classes.
  - 'macro': Calculate the metric for each class separately, and average the metrics across classes (with equal weights for each class).
  - 'weighted': Calculate the metric for each class separately, and average the metrics across classes, weighting each class by its support (`tp + fn`).
  - 'none' or `None`: Calculate the metric for each class separately, and return the metric for every class.
  - 'samples': Calculate the metric for each sample, and average the metrics across samples (with equal weights for each sample).

---

**Note:** What is considered a sample in the multi-dimensional multi-class case depends on the value of `mdmc_average`.

---

---

**Note:** If 'none' and a given class doesn't occur in the `preds` or `target`, the value for the class will be `nan`.

---

- **mdmc\_average** (`Optional[str]`) – Defines how averaging is done for multi-dimensional multi-class inputs (on top of the `average` parameter). Should be one of the following:
  - `None` [default]: Should be left unchanged if your data is not multi-dimensional multi-class.
  - 'samplewise': In this case, the statistics are computed separately for each sample on the `N` axis, and then averaged over samples. The computation for each sample is done by treating the flattened extra axes ... as the `N` dimension within the sample, and computing the metric for the sample based on that.
  - 'global': In this case the `N` and ... dimensions of the inputs are flattened into a new `N_X` sample axis, i.e. the inputs are treated as if they were `(N_X, C)`. From here on the `average` parameter applies as usual.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score, regardless of reduction method. If an index is ignored, and `average=None` or 'none', the score for the ignored class will be returned as `nan`.
- **num\_classes** (`Optional[int]`) – Number of classes. Necessary for 'macro', 'weighted' and `None` average methods.
- **threshold** (`float`) – Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs. Default value of 0.5 corresponds to input being probabilities.
- **top\_k** (`Optional[int]`) – Number of the highest probability or logit score predictions considered finding the correct label, relevant only for (multi-dimensional) multi-class inputs. The default value (`None`) will be interpreted as 1 for these inputs.



Should be left at default (None) for all other types of inputs.

- **multiclass** (Optional[bool]) – Used only in certain special cases, where you want to treat inputs as a different type than what they appear to be.

#### Return type

Tensor

#### Returns

The shape of the returned tensor depends on the average parameter

- If average in ['micro', 'macro', 'weighted', 'samples'], a one-element tensor will be returned
- If average in ['none', None], the shape will be (C,), where C stands for the number of classes

#### Raises

- **ValueError** – If average is not one of "micro", "macro", "weighted", "samples", "none" or None
- **ValueError** – If mdmc\_average is not one of None, "samplewise", "global".
- **ValueError** – If average is set but num\_classes is not provided.
- **ValueError** – If num\_classes is set and ignore\_index is not in the range [0, num\_classes).

#### Example

```
>>> from torchmetrics.functional.classification import dice
>>> preds = torch.tensor([2, 0, 2, 1])
>>> target = torch.tensor([1, 1, 2, 0])
>>> dice(preds, target, average='micro')
tensor(0.2500)
```

## 1.31 Exact Match

### 1.31.1 Module Interface

#### ExactMatch

```
class torchmetrics.ExactMatch(task: typing_extensions.Literal[binary, multiclass, multilabel], threshold:
    float = 0.5, num_classes: int | None = None, num_labels: int | None = None,
    multidim_average: typing_extensions.Literal[global, samplewise] = 'global',
    ignore_index: int | None = None, validate_args: bool = True, **kwargs:
    ~typing.Any)
```

Compute Exact match (also known as subset accuracy).

Exact Match is a stricter version of accuracy where all labels have to match exactly for the sample to be correctly classified.

This module is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'multiclass' or multilabel. See the documentation of MulticlassExactMatch and MultilabelExactMatch for the specific details of each argument influence and examples.

**Legacy Example:**

```
>>> from torch import tensor
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 1], [2, 1], [0, 2]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = ExactMatch(task="multiclass", num_classes=3, multidim_average=
↳ 'global')
>>> metric(preds, target)
tensor(0.5000)
```

```
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 1], [2, 1], [0, 2]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = ExactMatch(task="multiclass", num_classes=3, multidim_average=
↳ 'samplewise')
>>> metric(preds, target)
tensor([1., 0.])
```

Initialize task metric.

**MulticlassExactMatch**

```
class torchmetrics.classification.MulticlassExactMatch(num_classes, multidim_average='global',
ignore_index=None, validate_args=True,
**kwargs)
```

Compute Exact match (also known as subset accuracy) for multiclass tasks.

Exact Match is a stricter version of accuracy where all labels have to match exactly for the sample to be correctly classified.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): An int tensor of shape (N, ...) or float tensor of shape (N, C, ...). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (**Tensor**): An int tensor of shape (N, ...).

As output to forward and compute the metric returns the following output:

- **mcem** (**Tensor**): A tensor whose returned shape depends on the `multidim_average` argument:
  - If `multidim_average` is set to `global` the output will be a scalar tensor
  - If `multidim_average` is set to `samplewise` the output will be a tensor of shape (N,)

**Parameters**

- **num\_classes** (**int**) – Integer specifying the number of labels
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional**[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation

- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Example (multidim tensors):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassExactMatch
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 1], [2, 1], [0, 2]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassExactMatch(num_classes=3, multidim_average='global')
>>> metric(preds, target)
tensor(0.5000)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MulticlassExactMatch
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 1], [2, 1], [0, 2]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassExactMatch(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([1., 0.])
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

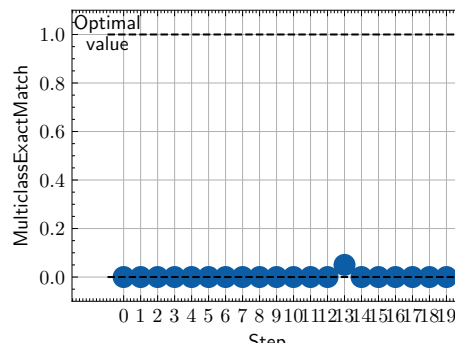
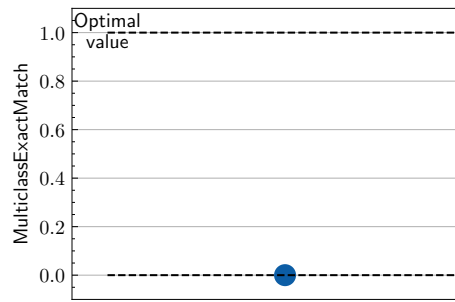
Figure object and Axes object

#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value per class
>>> from torch import randint
>>> from torchmetrics.classification import MulticlassExactMatch
>>> metric = MulticlassExactMatch(num_classes=3)
>>> metric.update(randint(3, (20,5)), randint(3, (20,5)))
>>> fig_, ax_ = metric.plot()
```

```
>>> from torch import randint
>>> # Example plotting a multiple values per class
>>> from torchmetrics.classification import MulticlassExactMatch
>>> metric = MulticlassExactMatch(num_classes=3)
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randint(3, (20,5)), randint(3, (20,5))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelExactMatch

```
class torchmetrics.classification.MultilabelExactMatch(num_labels, threshold=0.5,
                                                         multidim_average='global',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Compute Exact match (also known as subset accuracy) for multilabel tasks.

Exact Match is a stricter version of accuracy where all labels have to match exactly for the sample to be correctly classified.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int tensor or float tensor of shape (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** ([Tensor](#)): An int tensor of shape (N, C, ...).

As output to forward and compute the metric returns the following output:

- **mlem** ([Tensor](#)): A tensor whose returned shape depends on the **multidim\_average** argument:
  - If **multidim\_average** is set to **global** the output will be a scalar tensor
  - If **multidim\_average** is set to **samplewise** the output will be a tensor of shape (N,)

### Parameters

- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions

- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelExactMatch
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelExactMatch(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelExactMatch
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelExactMatch(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MultilabelExactMatch
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...               [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = MultilabelExactMatch(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0., 0.])
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

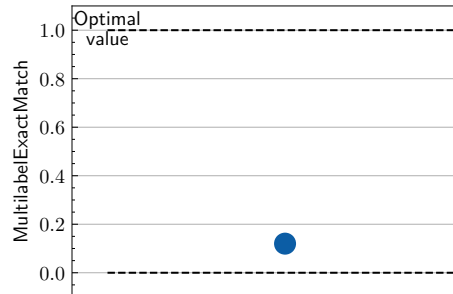
`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

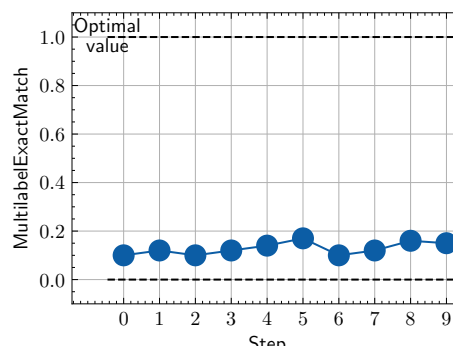
Figure and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> from torch import rand, randint
>>> from torchmetrics.classification import MultilabelExactMatch
>>> metric = MultilabelExactMatch(num_labels=3)
>>> metric.update(randint(2, (20, 3, 5)), randint(2, (20, 3, 5)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torch import rand, randint
>>> from torchmetrics.classification import MultilabelExactMatch
>>> metric = MultilabelExactMatch(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2, (20, 3, 5)), randint(2, (20, 3, 5))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.31.2 Functional Interface

#### exact\_match

```
torchmetrics.functional.classification.multilabel_exact_match(preds, target, num_labels,
                                                              threshold=0.5,
                                                              multidim_average='global',
                                                              ignore_index=None,
                                                              validate_args=True)
```

Compute Exact match (also known as subset accuracy) for multilabel tasks.

Exact Match is a stricter version of accuracy where all labels have to match exactly for the sample to be correctly classified.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

#### Parameters

- **preds** (**Tensor**) – Tensor with predictions
- **target** (**Tensor**) – Tensor with true labels
- **num\_labels** (**int**) – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional**[**int**]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

#### Returns

- If **multidim\_average** is set to **global** the output will be a scalar tensor
- If **multidim\_average** is set to **samplewise** the output will be a tensor of shape (N,)

#### Return type

The returned shape depends on the **multidim\_average** argument

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_exact_match
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
```

(continues on next page)

(continued from previous page)

```
>>> multilabel_exact_match(preds, target, num_labels=3)
tensor(0.5000)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multilabel_exact_match
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_exact_match(preds, target, num_labels=3)
tensor(0.5000)
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multilabel_exact_match
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> multilabel_exact_match(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0., 0.])
```

## multiclass\_exact\_match

`torchmetrics.functional.classification.multiclass_exact_match(preds, target, num_classes, multidim_average='global', ignore_index=None, validate_args=True)`

Compute Exact match (also known as subset accuracy) for multiclass tasks.

Exact Match is a stricter version of accuracy where all labels have to match exactly for the sample to be correctly classified.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of labels
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation



- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global` the output will be a scalar tensor
- If `multidim_average` is set to `samplewise` the output will be a tensor of shape `(N,)`

#### Return type

The returned shape depends on the `multidim_average` argument

#### Example (multidim tensors):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_exact_match
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 1], [2, 1], [0, 2]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_exact_match(preds, target, num_classes=3, multidim_average=
↳ 'global')
tensor(0.5000)
```

#### Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multiclass_exact_match
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 1], [2, 1], [0, 2]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_exact_match(preds, target, num_classes=3, multidim_average=
↳ 'samplewise')
tensor([1., 0.])
```

## multilabel\_exact\_match

```
torchmetrics.functional.classification.multilabel_exact_match(preds, target, num_labels,
                                                              threshold=0.5,
                                                              multidim_average='global',
                                                              ignore_index=None,
                                                              validate_args=True)
```

Compute Exact match (also known as subset accuracy) for multilabel tasks.

Exact Match is a stricter version of accuracy where all labels have to match exactly for the sample to be correctly classified.

Accepts the following input tensors:

- **preds** (int or float tensor): `(N, C, ...)`. If `preds` is a floating point tensor with values outside `[0,1]` range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): `(N, C, ...)`

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels

- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global` the output will be a scalar tensor
- If `multidim_average` is set to `samplewise` the output will be a tensor of shape `(N,)`

#### Return type

The returned shape depends on the `multidim_average` argument

#### Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_exact_match
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_exact_match(preds, target, num_labels=3)
tensor(0.5000)
```

#### Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multilabel_exact_match
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_exact_match(preds, target, num_labels=3)
tensor(0.5000)
```

#### Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multilabel_exact_match
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> multilabel_exact_match(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0., 0.]
```

## 1.32 F-1 Score

### 1.32.1 Module Interface

#### F1Score

```
class torchmetrics.F1Score(task: typing_extensions.Literal[binary, multiclass, multilabel], threshold: float = 0.5, num_classes: int | None = None, num_labels: int | None = None, average: typing_extensions.Literal[micro, macro, weighted, none] | None = 'micro', multidim_average: typing_extensions.Literal[global, samplewise] | None = 'global', top_k: int | None = 1, ignore_index: int | None = None, validate_args: bool = True, **kwargs: ~typing.Any)
```

Compute F-1 score.

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

The metric is only proper defined when  $TP + FP \neq 0 \wedge TP + FN \neq 0$  where TP, FP and FN represent the number of true positives, false positives and false negatives respectively. If this case is encountered for any class/label, the metric for that class/label will be set to 0 and the overall metric may therefore be affected in turn.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of BinaryF1Score, MulticlassF1Score and MultilabelF1Score for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([0, 1, 2, 0, 1, 2])
>>> preds = tensor([0, 2, 1, 0, 0, 1])
>>> f1 = F1Score(task="multiclass", num_classes=3)
>>> f1(preds, target)
tensor(0.3333)
```

Initialize task metric.

#### BinaryF1Score

```
class torchmetrics.classification.BinaryF1Score(threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True, **kwargs)
```

Compute F-1 score for binary tasks.

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

The metric is only proper defined when  $TP + FP \neq 0 \wedge TP + FN \neq 0$  where TP, FP and FN represent the number of true positives, false positives and false negatives respectively. If this case is encountered a score of 0 is returned.

As input to forward and update the metric accepts the following input:

- **preds** (`Tensor`): An int or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.

- **target** (`Tensor`): An int tensor of shape (N, ...)

As output to forward and compute the metric returns the following output:

- **bfls** (`Tensor`): A tensor whose returned shape depends on the `multidim_average` argument:
  - If `multidim_average` is set to `global`, the metric returns a scalar value.
  - If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

#### Parameters

- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryF1Score
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryF1Score()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinaryF1Score
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryF1Score()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import BinaryF1Score
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = BinaryF1Score(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.5000, 0.0000])
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

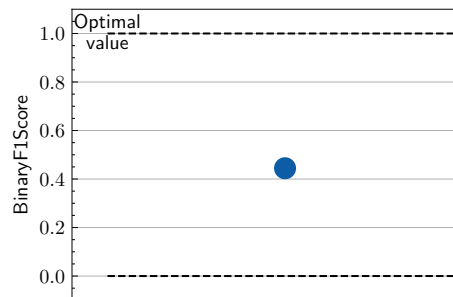
#### Returns

Figure object and Axes object

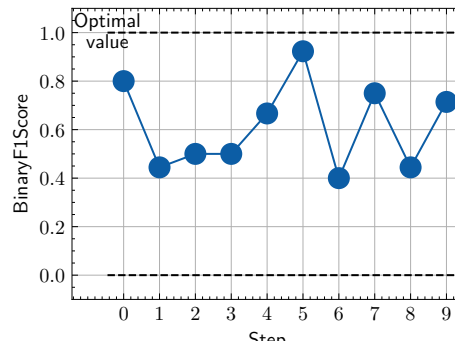
#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinaryF1Score
>>> metric = BinaryF1Score()
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryF1Score
>>> metric = BinaryF1Score()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassF1Score

```
class torchmetrics.classification.MulticlassF1Score(num_classes, top_k=1, average='macro',
                                                    multidim_average='global', ignore_index=None,
                                                    validate_args=True, **kwargs)
```

Compute F-1 score for multiclass tasks.

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

The metric is only proper defined when  $TP + FP \neq 0 \wedge TP + FN \neq 0$  where TP, FP and FN represent the number of true positives, false positives and false negatives respectively. If this case is encountered for any class, the metric for that class will be set to 0 and the overall metric may therefore be affected in turn.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int tensor of shape (N, ...) or float tensor of shape (N, C, ...). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** ([Tensor](#)): An int tensor of shape (N, ...)

As output to forward and compute the metric returns the following output:

- **mcfs** ([Tensor](#)): A tensor whose returned shape depends on the `average` and `multidim_average` arguments:
  - If `multidim_average` is set to `global`:
    - \* If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
    - \* If `average=None/'none'`, the shape will be (C,)
  - If `multidim_average` is set to `samplewise`:
    - \* If `average='micro'/'macro'/'weighted'`, the shape will be (N,)
    - \* If `average=None/'none'`, the shape will be (N, C)

### Parameters

- **preds** – Tensor with predictions
- **target** – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:

- micro: Sum statistics over all labels
- macro: Calculate statistics for each label and average them
- weighted: calculates statistics for each label and computes weighted average using their support
- "none" or None: calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassF1Score
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassF1Score(num_classes=3)
>>> metric(preds, target)
tensor(0.7778)
>>> mcf1s = MulticlassF1Score(num_classes=3, average=None)
>>> mcf1s(preds, target)
tensor([0.6667, 0.6667, 1.0000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MulticlassF1Score
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                 [0.22, 0.61, 0.17],
...                 [0.71, 0.09, 0.20],
...                 [0.05, 0.82, 0.13]])
>>> metric = MulticlassF1Score(num_classes=3)
>>> metric(preds, target)
tensor(0.7778)
>>> mcf1s = MulticlassF1Score(num_classes=3, average=None)
>>> mcf1s(preds, target)
tensor([0.6667, 0.6667, 1.0000])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MulticlassF1Score
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
```

(continues on next page)

(continued from previous page)

```
>>> preds = tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> metric = MulticlassF1Score(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.4333, 0.2667])
>>> mcf1s = MulticlassF1Score(num_classes=3, multidim_average='samplewise',
                               average=None)
>>> mcf1s(preds, target)
tensor([[0.8000, 0.0000, 0.5000],
        [0.0000, 0.4000, 0.4000]])
```

**plot**(*val=None*, *ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

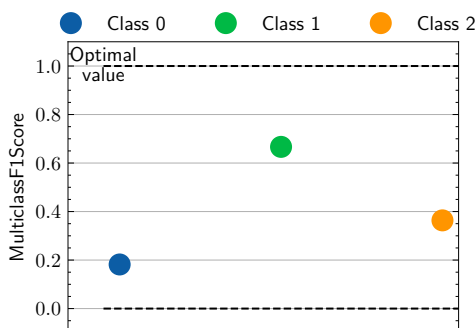
#### Returns

Figure object and Axes object

#### Raises

`ModuleNotFoundError` – If *matplotlib* is not installed

```
>>> from torch import randint
>>> # Example plotting a single value per class
>>> from torchmetrics.classification import MulticlassF1Score
>>> metric = MulticlassF1Score(num_classes=3, average=None)
>>> metric.update(randint(3, (20,)), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```



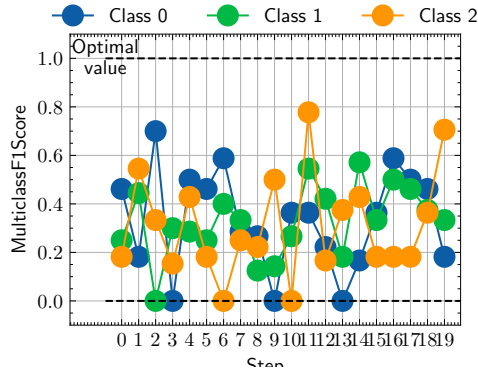
```
>>> from torch import randint
>>> # Example plotting a multiple values per class
>>> from torchmetrics.classification import MulticlassF1Score
>>> metric = MulticlassF1Score(num_classes=3, average=None)
```

(continues on next page)



(continued from previous page)

```
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randint(3, (20,)), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelF1Score

```
class torchmetrics.classification.MultilabelF1Score(num_labels, threshold=0.5, average='macro',
                                                    multidim_average='global', ignore_index=None,
                                                    validate_args=True, **kwargs)
```

Compute F-1 score for multilabel tasks.

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

The metric is only proper defined when  $TP + FP \neq 0 \wedge TP + FN \neq 0$  where TP, FP and FN represent the number of true positives, false positives and false negatives respectively. If this case is encountered for any label, the metric for that label will be set to 0 and the overall metric may therefore be affected in turn.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int or float tensor of shape  $(N, C, \dots)$ . If preds is a floating point tensor with values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** ([Tensor](#)): An int tensor of shape  $(N, C, \dots)$ .

As output to forward and compute the metric returns the following output:

- **mlf1s** ([Tensor](#)): A tensor whose returned shape depends on the **average** and **multidim\_average** arguments:
  - If **multidim\_average** is set to **global**:
    - \* If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
    - \* If **average**=None/'none', the shape will be  $(C,)$
  - If **multidim\_average** is set to **samplewise**:
    - \* If **average**='micro'/'macro'/'weighted', the shape will be  $(N,)$
    - \* If **average**=None/'none', the shape will be  $(N, C)$

### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelF1Score
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelF1Score(num_labels=3)
>>> metric(preds, target)
tensor(0.5556)
>>> mlf1s = MultilabelF1Score(num_labels=3, average=None)
>>> mlf1s(preds, target)
tensor([1.0000, 0.0000, 0.6667])
```

### Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelF1Score
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelF1Score(num_labels=3)
>>> metric(preds, target)
tensor(0.5556)
>>> mlf1s = MultilabelF1Score(num_labels=3, average=None)
>>> mlf1s(preds, target)
tensor([1.0000, 0.0000, 0.6667])
```

### Example (multidim tensors):

```
>>> from torchmetrics.classification import MultilabelF1Score
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                 [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = MultilabelF1Score(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.4444, 0.0000])
>>> mlf1s = MultilabelF1Score(num_labels=3, multidim_average='samplewise',
↪ average=None)
>>> mlf1s(preds, target)
tensor([[0.6667, 0.6667, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

**plot**(*val=None*, *ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

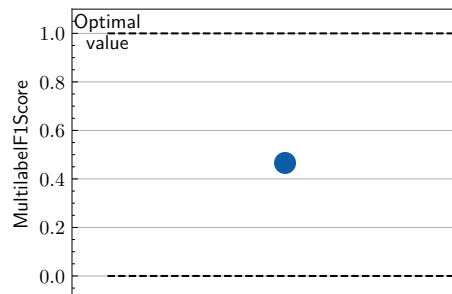
#### Returns

Figure and Axes object

#### Raises

`ModuleNotFoundError` – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelF1Score
>>> metric = MultilabelF1Score(num_labels=3)
>>> metric.update(randint(2, (20, 3)), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```

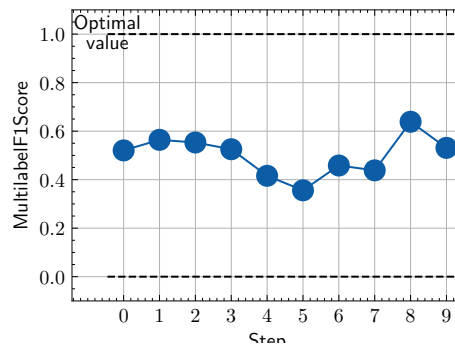


```
>>> from torch import rand, randint
>>> # Example plotting multiple values
```

(continues on next page)

(continued from previous page)

```
>>> from torchmetrics.classification import MultilabelF1Score
>>> metric = MultilabelF1Score(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2, (20, 3)), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```



## 1.32.2 Functional Interface

### f1\_score

`torchmetrics.functional.f1_score(preds, target, task, threshold=0.5, num_classes=None, num_labels=None, average='micro', multidim_average='global', top_k=1, ignore_index=None, validate_args=True)`

Compute F-1 score. :rtype: `Tensor`

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either `'binary'`, `'multiclass'` or `'multilabel'`. See the documentation of `binary_f1_score()`, `multiclass_f1_score()` and `multilabel_f1_score()` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([0, 1, 2, 0, 1, 2])
>>> preds = tensor([0, 2, 1, 0, 0, 1])
>>> f1_score(preds, target, task="multiclass", num_classes=3)
tensor(0.3333)
```

## binary\_f1\_score

`torchmetrics.functional.classification.binary_f1_score(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Compute F-1 score for binary tasks.

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

### Return type

[Tensor](#)

### Returns

If **multidim\_average** is set to **global**, the metric returns a scalar value. If **multidim\_average** is set to **samplewise**, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_f1_score
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> binary_f1_score(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_f1_score
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_f1_score(preds, target)
tensor(0.6667)
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import binary_f1_score
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...               [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> binary_f1_score(preds, target, multidim_average='samplewise')
tensor([0.5000, 0.0000])
```

## multiclass\_f1\_score

`torchmetrics.functional.classification.multiclass_f1_score(preds, target, num_classes, average='macro', top_k=1, multidim_average='global', ignore_index=None, validate_args=True)`

Compute F-1 score for multiclass tasks.

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:

- `global`: Additional dimensions are flattened along the batch dimension
- `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **`ignore_index`** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **`validate_args`** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

### Return type

The returned shape depends on the `average` and `multidim_average` arguments

### Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_f1_score
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_f1_score(preds, target, num_classes=3)
tensor(0.7778)
>>> multiclass_f1_score(preds, target, num_classes=3, average=None)
tensor([0.6667, 0.6667, 1.0000])
```

### Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multiclass_f1_score
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> multiclass_f1_score(preds, target, num_classes=3)
tensor(0.7778)
>>> multiclass_f1_score(preds, target, num_classes=3, average=None)
tensor([0.6667, 0.6667, 1.0000])
```

### Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multiclass_f1_score
>>> target = tensor([[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]])
>>> preds = tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> multiclass_f1_score(preds, target, num_classes=3, multidim_average=
↪ 'samplewise')
```

(continues on next page)

(continued from previous page)

```

tensor([0.4333, 0.2667])
>>> multiclass_f1_score(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.8000, 0.0000, 0.5000],
        [0.0000, 0.4000, 0.4000]])

```

## multilabel\_f1\_score

```

torchmetrics.functional.classification.multilabel_f1_score(preds, target, num_labels,
                                                            threshold=0.5, average='macro',
                                                            multidim_average='global',
                                                            ignore_index=None,
                                                            validate_args=True)

```

Compute F-1 score for multilabel tasks.

$$F_1 = 2 \frac{\text{precision} * \text{recall}}{(\text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional\[Literal\['micro', 'macro', 'weighted', 'none'\]\]](#)) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal\['global', 'samplewise'\]](#)) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation



- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

#### Return type

The returned shape depends on the `average` and `multidim_average` arguments

#### Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_f1_score
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_f1_score(preds, target, num_labels=3)
tensor(0.5556)
>>> multilabel_f1_score(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.6667])
```

#### Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multilabel_f1_score
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_f1_score(preds, target, num_labels=3)
tensor(0.5556)
>>> multilabel_f1_score(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.6667])
```

#### Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multilabel_f1_score
>>> target = tensor([[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]])
>>> preds = tensor([[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...               [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]])
>>> multilabel_f1_score(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0.4444, 0.0000])
>>> multilabel_f1_score(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.6667, 0.6667, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

## 1.33 F-Beta Score

### 1.33.1 Module Interface

#### FBetaScore

```
class torchmetrics.FBetaScore(task: typing_extensions.Literal[binary, multiclass, multilabel], beta: float = 1.0, threshold: float = 0.5, num_classes: int | None = None, num_labels: int | None = None, average: typing_extensions.Literal[micro, macro, weighted, none] | None = 'micro', multidim_average: typing_extensions.Literal[global, samplewise] | None = 'global', top_k: int | None = 1, ignore_index: int | None = None, validate_args: bool = True, **kwargs: ~typing.Any)
```

Compute F-score metric.

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

The metric is only proper defined when  $TP + FP \neq 0 \wedge TP + FN \neq 0$  where TP, FP and FN represent the number of true positives, false positives and false negatives respectively. If this case is encountered for any class/label, the metric for that class/label will be set to 0 and the overall metric may therefore be affected in turn.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or 'multilabel'. See the documentation of `binary_fbeta_score()`, `multiclass_fbeta_score()` and `multilabel_fbeta_score()` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([0, 1, 2, 0, 1, 2])
>>> preds = tensor([0, 2, 1, 0, 0, 1])
>>> f_beta = FBetaScore(task="multiclass", num_classes=3, beta=0.5)
>>> f_beta(preds, target)
tensor(0.3333)
```

Initialize task metric.

#### BinaryFBetaScore

```
class torchmetrics.classification.BinaryFBetaScore(beta, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True, **kwargs)
```

Compute F-score metric for binary tasks.

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

The metric is only proper defined when  $TP + FP \neq 0 \wedge TP + FN \neq 0$  where TP, FP and FN represent the number of true positives, false positives and false negatives respectively. If this case is encountered a score of 0 is returned.

As input to forward and update the metric accepts the following input:

- **preds** (`Tensor`): An int tensor or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.

- **target** (`Tensor`): An int tensor of shape (N, ...).

As output to forward and compute the metric returns the following output:

- **bfbfs** (`Tensor`): A tensor whose returned shape depends on the `multidim_average` argument:
  - If `multidim_average` is set to `global` the output will be a scalar tensor
  - If `multidim_average` is set to `samplewise` the output will be a tensor of shape (N,) consisting of a scalar value per sample.

#### Parameters

- **beta** (`float`) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryFBetaScore
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryFBetaScore(beta=2.0)
>>> metric(preds, target)
tensor(0.6667)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryFBetaScore
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryFBetaScore(beta=2.0)
>>> metric(preds, target)
tensor(0.6667)
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import BinaryFBetaScore
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = BinaryFBetaScore(beta=2.0, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.5882, 0.0000])
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

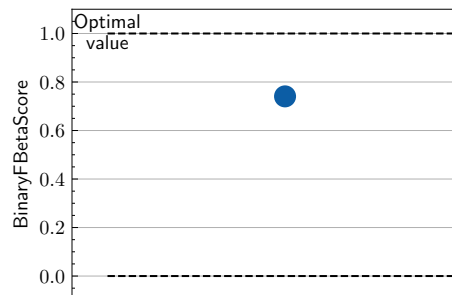
#### Returns

Figure object and Axes object

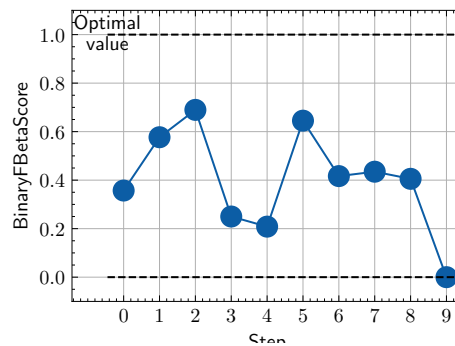
#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinaryFBetaScore
>>> metric = BinaryFBetaScore(beta=2.0)
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryFBetaScore
>>> metric = BinaryFBetaScore(beta=2.0)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassFBetaScore

```
class torchmetrics.classification.MulticlassFBetaScore(beta, num_classes, top_k=1,
                                                    average='macro',
                                                    multidim_average='global',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Compute **F-score** metric for multiclass tasks.

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

The metric is only proper defined when  $TP + FP \neq 0 \wedge TP + FN \neq 0$  where TP, FP and FN represent the number of true positives, false positives and false negatives respectively. If this case is encountered for any class, the metric for that class will be set to 0 and the overall metric may therefore be affected in turn.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): An int tensor of shape (N, ...) or float tensor of shape (N, C, ...). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (**Tensor**): An int tensor of shape (N, ...).

As output to forward and compute the metric returns the following output:

- **mcfbfs** (**Tensor**): A tensor whose returned shape depends on the `average` and `multidim_average` arguments:
  - If `multidim_average` is set to `global`:
    - \* If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
    - \* If `average=None/'none'`, the shape will be (C,)
  - If `multidim_average` is set to `samplewise`:
    - \* If `average='micro'/'macro'/'weighted'`, the shape will be (N,)
    - \* If `average=None/'none'`, the shape will be (N, C)

### Parameters

- **beta** (**float**) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **num\_classes** (**int**) – Integer specifying the number of classes

- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassFBetaScore
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassFBetaScore(beta=2.0, num_classes=3)
>>> metric(preds, target)
tensor(0.7963)
>>> mcfbs = MulticlassFBetaScore(beta=2.0, num_classes=3, average=None)
>>> mcfbs(preds, target)
tensor([0.5556, 0.8333, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MulticlassFBetaScore
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                 [0.22, 0.61, 0.17],
...                 [0.71, 0.09, 0.20],
...                 [0.05, 0.82, 0.13]])
>>> metric = MulticlassFBetaScore(beta=2.0, num_classes=3)
>>> metric(preds, target)
tensor(0.7963)
>>> mcfbs = MulticlassFBetaScore(beta=2.0, num_classes=3, average=None)
>>> mcfbs(preds, target)
tensor([0.5556, 0.8333, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MulticlassFBetaScore
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassFBetaScore(beta=2.0, num_classes=3, multidim_average=
↳ 'samplewise')
>>> metric(preds, target)
tensor([0.4697, 0.2706])
>>> mcfbs = MulticlassFBetaScore(beta=2.0, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
>>> mcfbs(preds, target)
tensor([[0.9091, 0.0000, 0.5000],
        [0.0000, 0.3571, 0.4545]])
```

**plot**(*val=None*, *ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

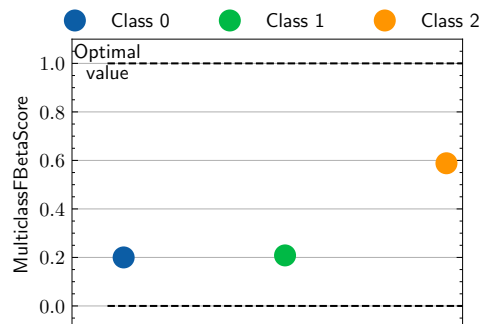
#### Returns

Figure object and Axes object

#### Raises

**`ModuleNotFoundError`** – If *matplotlib* is not installed

```
>>> from torch import randint
>>> # Example plotting a single value per class
>>> from torchmetrics.classification import MulticlassFBetaScore
>>> metric = MulticlassFBetaScore(num_classes=3, beta=2.0, average=None)
>>> metric.update(randint(3, (20,)), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```

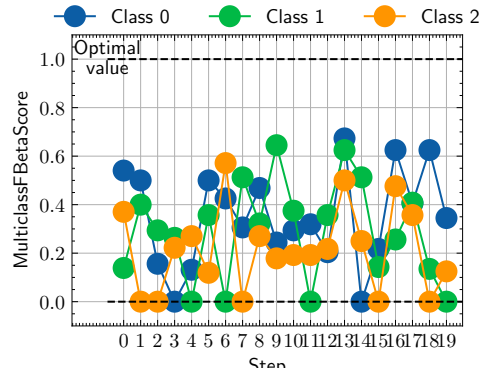


```
>>> from torch import randint
>>> # Example plotting a multiple values per class
```

(continues on next page)

(continued from previous page)

```
>>> from torchmetrics.classification import MulticlassFBetaScore
>>> metric = MulticlassFBetaScore(num_classes=3, beta=2.0, average=None)
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randint(3, (20,)), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelFBetaScore

```
class torchmetrics.classification.MultilabelFBetaScore(beta, num_labels, threshold=0.5,
                                                       average='macro',
                                                       multidim_average='global',
                                                       ignore_index=None, validate_args=True,
                                                       **kwargs)
```

Compute **F-score** metric for multilabel tasks.

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

The metric is only proper defined when  $TP + FP \neq 0 \wedge TP + FN \neq 0$  where TP, FP and FN represent the number of true positives, false positives and false negatives respectively. If this case is encountered for any label, the metric for that label will be set to 0 and the overall metric may therefore be affected in turn.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): An int or float tensor of shape  $(N, C, \dots)$ . If preds is a floating point tensor with values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (**Tensor**): An int tensor of shape  $(N, C, \dots)$ .

As output to forward and compute the metric returns the following output:

- **mlfbs** (**Tensor**): A tensor whose returned shape depends on the **average** and **multidim\_average** arguments:
  - If **multidim\_average** is set to **global**:
    - \* If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
    - \* If **average**=None/'none', the shape will be  $(C,)$
  - If **multidim\_average** is set to **samplewise**:



- \* If `average='micro'/'macro'/'weighted'`, the shape will be  $(N,)$
- \* If `average=None/'none'`, the shape will be  $(N, C)$

### Parameters

- **beta** (`float`) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **num\_labels** (`int`) – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the  $N$  axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelFBetaScore
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelFBetaScore(beta=2.0, num_labels=3)
>>> metric(preds, target)
tensor(0.6111)
>>> mlfbs = MultilabelFBetaScore(beta=2.0, num_labels=3, average=None)
>>> mlfbs(preds, target)
tensor([1.0000, 0.0000, 0.8333])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelFBetaScore
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelFBetaScore(beta=2.0, num_labels=3)
>>> metric(preds, target)
tensor(0.6111)
```

(continues on next page)

(continued from previous page)

```
>>> mlfbs = MultilabelFBetaScore(beta=2.0, num_labels=3, average=None)
>>> mlfbs(preds, target)
tensor([1.0000, 0.0000, 0.8333])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MultilabelFBetaScore
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                 [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = MultilabelFBetaScore(num_labels=3, beta=2.0, multidim_average=
↳ 'samplewise')
>>> metric(preds, target)
tensor([0.5556, 0.0000])
>>> mlfbs = MultilabelFBetaScore(num_labels=3, beta=2.0, multidim_average=
↳ 'samplewise', average=None)
>>> mlfbs(preds, target)
tensor([[0.8333, 0.8333, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

**Parameters**

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

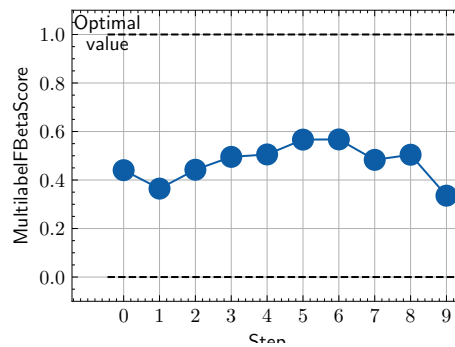
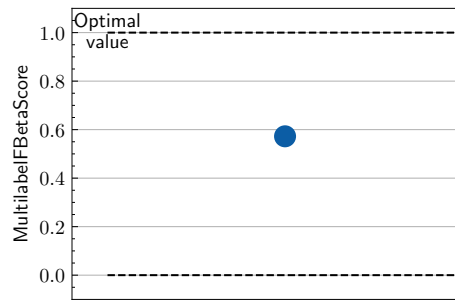
**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

Figure and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelFBetaScore
>>> metric = MultilabelFBetaScore(num_labels=3, beta=2.0)
>>> metric.update(randint(2, (20, 3)), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```

```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import MultilabelFBetaScore
>>> metric = MultilabelFBetaScore(num_labels=3, beta=2.0)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2, (20, 3)), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.33.2 Functional Interface

#### fbeta\_score

`torchmetrics.functional.fbeta_score(preds, target, task, beta=1.0, threshold=0.5, num_classes=None, num_labels=None, average='micro', multidim_average='global', top_k=1, ignore_index=None, validate_args=True)`

Compute F-score metric. :rtype: `Tensor`

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either 'binary', 'multiclass' or 'multilabel'. See the documentation of `binary_fbeta_score()`, `multiclass_fbeta_score()` and `multilabel_fbeta_score()` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([0, 1, 2, 0, 1, 2])
>>> preds = tensor([0, 2, 1, 0, 0, 1])
>>> fbeta_score(preds, target, task="multiclass", num_classes=3, beta=0.5)
tensor(0.3333)
```

## binary\_fbeta\_score

```
torchmetrics.functional.classification.binary_fbeta_score(preds, target, beta, threshold=0.5,
                                                           multidim_average='global',
                                                           ignore_index=None,
                                                           validate_args=True)
```

Compute **F-score** metric for binary tasks.

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** (**Tensor**) – Tensor with predictions
- **target** (**Tensor**) – Tensor with true labels
- **beta** (**float**) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **threshold** (**float**) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional**[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Return type

**Tensor**

### Returns

If **multidim\_average** is set to **global**, the metric returns a scalar value. If **multidim\_average** is set to **samplewise**, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_fbeta_score
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> binary_fbeta_score(preds, target, beta=2.0)
tensor(0.6667)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import binary_fbeta_score
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_fbeta_score(preds, target, beta=2.0)
tensor(0.6667)
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import binary_fbeta_score
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> binary_fbeta_score(preds, target, beta=2.0, multidim_average='samplewise')
tensor([0.5882, 0.0000])
```

## multiclass\_fbeta\_score

`torchmetrics.functional.classification.multiclass_fbeta_score(preds, target, beta, num_classes, average='macro', top_k=1, multidim_average='global', ignore_index=None, validate_args=True)`

Compute F-score metric for multiclass tasks.

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (`N, ...`) (int tensor) or (`N, C, ...`) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (`N, ...`)

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **beta** (`float`) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction

- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

#### Return type

The returned shape depends on the `average` and `multidim_average` arguments

#### Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_fbeta_score
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3)
tensor(0.7963)
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3, average=None)
tensor([0.5556, 0.8333, 1.0000])
```

#### Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multiclass_fbeta_score
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                 [0.22, 0.61, 0.17],
...                 [0.71, 0.09, 0.20],
...                 [0.05, 0.82, 0.13]])
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3)
tensor(0.7963)
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3, average=None)
tensor([0.5556, 0.8333, 1.0000])
```

#### Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multiclass_fbeta_score
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3, multidim_
    ↪average='samplewise')
tensor([0.4697, 0.2706])
>>> multiclass_fbeta_score(preds, target, beta=2.0, num_classes=3, multidim_
    ↪average='samplewise', average=None)
tensor([[0.9091, 0.0000, 0.5000],
        [0.0000, 0.3571, 0.4545]])
```

## multilabel\_fbeta\_score

`torchmetrics.functional.classification.multilabel_fbeta_score`(*preds, target, beta, num\_labels, threshold=0.5, average='macro', multidim\_average='global', ignore\_index=None, validate\_args=True*)

Compute F-score metric for multilabel tasks.

$$F_{\beta} = (1 + \beta^2) * \frac{\text{precision} * \text{recall}}{(\beta^2 * \text{precision}) + \text{recall}}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

### Parameters

- **preds** (**Tensor**) – Tensor with predictions
- **target** (**Tensor**) – Tensor with true labels
- **beta** (**float**) – Weighting between precision and recall in calculation. Setting to 1 corresponds to equal weight
- **num\_labels** (**int**) – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **average** (**Optional[Literal['micro', 'macro', 'weighted', 'none']]**) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **multidim\_average** (**Literal['global', 'samplewise']**) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension

- **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

### Return type

The returned shape depends on the `average` and `multidim_average` arguments

#### Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_fbeta_score
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_fbeta_score(preds, target, beta=2.0, num_labels=3)
tensor(0.6111)
>>> multilabel_fbeta_score(preds, target, beta=2.0, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.8333])
```

#### Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multilabel_fbeta_score
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_fbeta_score(preds, target, beta=2.0, num_labels=3)
tensor(0.6111)
>>> multilabel_fbeta_score(preds, target, beta=2.0, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.8333])
```

#### Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multilabel_fbeta_score
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> multilabel_fbeta_score(preds, target, num_labels=3, beta=2.0, multidim_
↪ average='samplewise')
tensor([0.5556, 0.0000])
>>> multilabel_fbeta_score(preds, target, num_labels=3, beta=2.0, multidim_
↪ average='samplewise', average=None)
```

(continues on next page)



(continued from previous page)

```
tensor([[0.8333, 0.8333, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

## 1.34 Group Fairness

### 1.34.1 Module Interface

#### BinaryFairness

```
class torchmetrics.classification.BinaryFairness(num_groups, task='all', threshold=0.5,
                                                  ignore_index=None, validate_args=True,
                                                  **kwargs)
```

Computes [Demographic parity](#) and [Equal opportunity](#) ratio for binary classification problems.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **groups** (int tensor): (N, ...). The group identifiers should be 0, 1, ..., (num\_groups - 1).
- **target** (int tensor): (N, ...).

The additional dimensions are flattened along the batch dimension.

This class computes the ratio between positivity rates and true positives rates for different groups. If more than two groups are present, the disparity between the lowest and highest group is reported. A disparity between positivity rates indicates a potential violation of demographic parity, and between true positive rates indicates a potential violation of equal opportunity.

The lowest rate is divided by the highest, so a lower value means more discrimination against the numerator. In the results this is also indicated as the key of dict is {metric}\_{identifier\_low\_group}\_{identifier\_high\_group}.

#### Parameters

- **num\_groups** (**int**) – The number of groups.
- **task** (**Literal**['demographic\_parity', 'equal\_opportunity', 'all']) – The task to compute. Can be either demographic\_parity or equal\_opportunity or all.
- **threshold** (**float**) – Threshold for transforming probability to binary {0,1} predictions.
- **ignore\_index** (**Optional[int]**) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.
- **kwargs** (**Any**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Returns

The metric returns a dict where the key identifies the metric and groups with the lowest and highest true positives rates as follows: {metric}\_{identifier\_low\_group}\_{identifier\_high\_group}. The value is a tensor with the disparity rate.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import BinaryFairness
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 1, 0, 1, 0, 1])
>>> groups = torch.tensor([0, 1, 0, 1, 0, 1])
>>> metric = BinaryFairness(2)
>>> metric(preds, target, groups)
{'DP_0_1': tensor(0.), 'EO_0_1': tensor(0.)}
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryFairness
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.84, 0.22, 0.73, 0.33, 0.92])
>>> groups = torch.tensor([0, 1, 0, 1, 0, 1])
>>> metric = BinaryFairness(2)
>>> metric(preds, target, groups)
{'DP_0_1': tensor(0.), 'EO_0_1': tensor(0.)}
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

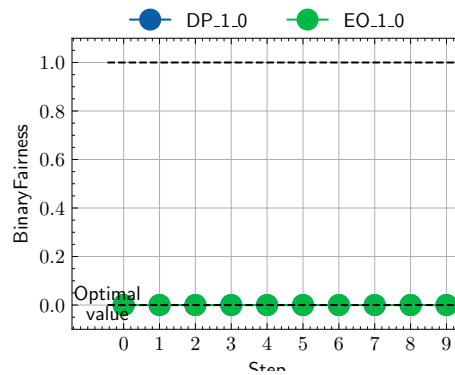
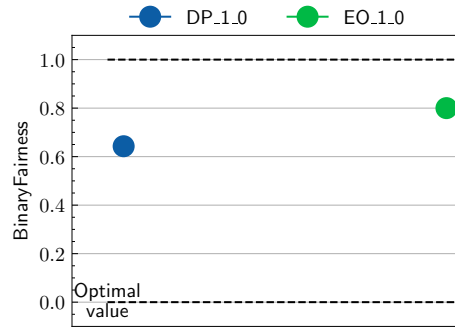
Figure object and Axes object

#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinaryFairness
>>> metric = BinaryFairness(2)
>>> metric.update(rand(20), randint(2,(20,)), randint(2,(20,)))
>>> fig_, ax_ = metric.plot()
```

```
>>> from torch import rand, randint, ones
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryFairness
>>> metric = BinaryFairness(2)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(20), randint(2,(20,)), ones(20).long()))
>>> fig_, ax_ = metric.plot(values)
```



## BinaryGroupStatRates

```
class torchmetrics.classification.BinaryGroupStatRates(num_groups, threshold=0.5,
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Computes the true/false positives and true/false negatives rates for binary classification by group.

Related to [Type I](#) and [Type II](#) errors.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...).
- **groups** (int tensor): (N, ...). The group identifiers should be 0, 1, ..., (num\_groups - 1).

The additional dimensions are flattened along the batch dimension.

### Parameters

- **num\_groups** (**int**) – The number of groups.
- **threshold** (**float**) – Threshold for transforming probability to binary {0,1} predictions.
- **ignore\_index** (**Optional[int]**) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Returns**

The metric returns a dict with a group identifier as key and a tensor with the tp, fp, tn and fn rates as value.

**Example (preds is int tensor):**

```
>>> from torchmetrics.classification import BinaryGroupStatRates
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 1, 0, 1, 0, 1])
>>> groups = torch.tensor([0, 1, 0, 1, 0, 1])
>>> metric = BinaryGroupStatRates(num_groups=2)
>>> metric(preds, target, groups)
{'group_0': tensor([0., 0., 1., 0.]), 'group_1': tensor([1., 0., 0., 0.]})
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinaryGroupStatRates
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.84, 0.22, 0.73, 0.33, 0.92])
>>> groups = torch.tensor([0, 1, 0, 1, 0, 1])
>>> metric = BinaryGroupStatRates(num_groups=2)
>>> metric(preds, target, groups)
{'group_0': tensor([0., 0., 1., 0.]), 'group_1': tensor([1., 0., 0., 0.]})
```

## 1.34.2 Functional Interface

### binary\_fairness

`torchmetrics.functional.classification.binary_fairness(preds, target, groups, task='all', threshold=0.5, ignore_index=None, validate_args=True)`

Compute either [Demographic parity](#) and [Equal opportunity](#) ratio for binary classification problems.

This is done by setting the task argument to either 'demographic\_parity', 'equal\_opportunity' or all. See the documentation of `_compute_binary_demographic_parity()` and `_compute_binary_equal_opportunity()` for the specific details of each argument influence and examples.

**Parameters**

- **preds** (*Tensor*) – Tensor with predictions.
- **target** (*Tensor*) – Tensor with true labels (not required for `demographic_parity`).
- **groups** (*Tensor*) – Tensor with group identifiers. The group identifiers should be 0, 1, ..., (num\_groups - 1).
- **task** (*Literal*['demographic\_parity', 'equal\_opportunity', 'all']) – The task to compute. Can be either `demographic_parity` or `equal_opportunity` or `all`.
- **threshold** (*float*) – Threshold for transforming probability to binary {0,1} predictions.
- **ignore\_index** (*Optional*[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation

- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type**`Dict[str, Tensor]`**demographic\_parity**

```
torchmetrics.functional.classification.demographic_parity(preds, groups, threshold=0.5,
                                                         ignore_index=None,
                                                         validate_args=True)
```

Demographic parity compares the positivity rates between all groups.

If more than two groups are present, the disparity between the lowest and highest group is reported. The lowest positivity rate is divided by the highest, so a lower value means more discrimination against the numerator. In the results this is also indicated as the key of dict is `DP_{identifier_low_group}_{identifier_high_group}`.

$$DP = \frac{\min_a PR_a}{\max_a PR_a}.$$

where PR represents the positivity rate for group a.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **groups** (int tensor): (N, ...). The group identifiers should be 0, 1, ..., (num\_groups - 1).
- **target** (int tensor): (N, ...).

The additional dimensions are flattened along the batch dimension.

**Parameters**

- **preds** (`Tensor`) – Tensor with predictions.
- **groups** (`Tensor`) – Tensor with group identifiers. The group identifiers should be 0, 1, ..., (num\_groups - 1).
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type**`Dict[str, Tensor]`**Returns**

The metric returns a dict where the key identifies the group with the lowest and highest positivity rates as follows: `DP_{identifier_low_group}_{identifier_high_group}`. The value is a tensor with the DP rate.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import demographic_parity
>>> preds = torch.tensor([0, 1, 0, 1, 0, 1])
>>> groups = torch.tensor([0, 1, 0, 1, 0, 1])
>>> demographic_parity(preds, groups)
{'DP_0_1': tensor(0.)}
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import demographic_parity
>>> preds = torch.tensor([0.11, 0.84, 0.22, 0.73, 0.33, 0.92])
>>> groups = torch.tensor([0, 1, 0, 1, 0, 1])
>>> demographic_parity(preds, groups)
{'DP_0_1': tensor(0.)}
```

## equal\_opportunity

`torchmetrics.functional.classification.equal_opportunity(preds, target, groups, threshold=0.5, ignore_index=None, validate_args=True)`

Equal opportunity compares the true positive rates between all groups.

If more than two groups are present, the disparity between the lowest and highest group is reported. The lowest true positive rate is divided by the highest, so a lower value means more discrimination against the numerator. In the results this is also indicated as the key of dict is `EO_{identifier_low_group}_{identifier_high_group}`.

$$DP = \frac{\min_a TPR_a}{\max_a TPR_a}.$$

where TPR represents the true positives rate for group a.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...).
- **groups** (int tensor): (N, ...). The group identifiers should be 0, 1, ..., (num\_groups - 1).

The additional dimensions are flattened along the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions.
- **target** ([Tensor](#)) – Tensor with true labels.
- **groups** ([Tensor](#)) – Tensor with group identifiers. The group identifiers should be 0, 1, ..., (num\_groups - 1).
- **threshold** ([float](#)) – Threshold for transforming probability to binary {0,1} predictions.
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Return type

[Dict](#)[[str](#), [Tensor](#)]

**Returns**

The metric returns a dict where the key identifies the group with the lowest and highest true positives rates as follows: `EO_{identifier_low_group}_{identifier_high_group}`. The value is a tensor with the EO rate.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import equal_opportunity
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 1, 0, 1, 0, 1])
>>> groups = torch.tensor([0, 1, 0, 1, 0, 1])
>>> equal_opportunity(preds, target, groups)
{'EO_0_1': tensor(0.)}
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import equal_opportunity
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.84, 0.22, 0.73, 0.33, 0.92])
>>> groups = torch.tensor([0, 1, 0, 1, 0, 1])
>>> equal_opportunity(preds, target, groups)
{'EO_0_1': tensor(0.)}
```

**binary\_groups\_stat\_rates**

`torchmetrics.functional.classification.binary_groups_stat_rates(preds, target, groups, num_groups, threshold=0.5, ignore_index=None, validate_args=True)`

Compute the true/false positives and true/false negatives rates for binary classification by group.

Related to [Type I and Type II errors](#).

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): (N, ...).
- **groups** (int tensor): (N, ...). The group identifiers should be 0, 1, ..., (num\_groups - 1).

The additional dimensions are flattened along the batch dimension.

**Parameters**

- **preds** ([Tensor](#)) – Tensor with predictions.
- **target** ([Tensor](#)) – Tensor with true labels.
- **groups** ([Tensor](#)) – Tensor with group identifiers. The group identifiers should be 0, 1, ..., (num\_groups - 1).
- **num\_groups** ([int](#)) – The number of groups.
- **threshold** ([float](#)) – Threshold for transforming probability to binary {0,1} predictions.

- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type**`Dict[str, Tensor]`**Returns**

The metric returns a dict with a group identifier as key and a tensor with the tp, fp, tn and fn rates as value.

**Example (preds is int tensor):**

```
>>> from torchmetrics.functional.classification import binary_groups_stat_rates
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0, 1, 0, 1, 0, 1])
>>> groups = torch.tensor([0, 1, 0, 1, 0, 1])
>>> binary_groups_stat_rates(preds, target, groups, 2)
{'group_0': tensor([0., 0., 1., 0.]), 'group_1': tensor([1., 0., 0., 0.]})
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_groups_stat_rates
>>> target = torch.tensor([0, 1, 0, 1, 0, 1])
>>> preds = torch.tensor([0.11, 0.84, 0.22, 0.73, 0.33, 0.92])
>>> groups = torch.tensor([0, 1, 0, 1, 0, 1])
>>> binary_groups_stat_rates(preds, target, groups, 2)
{'group_0': tensor([0., 0., 1., 0.]), 'group_1': tensor([1., 0., 0., 0.]})
```

## 1.35 Hamming Distance

### 1.35.1 Module Interface

#### HammingDistance

```
class torchmetrics.HammingDistance(task: typing_extensions.Literal[binary, multiclass, multilabel],
                                   threshold: float = 0.5, num_classes: int | None = None, num_labels:
                                   int | None = None, average: typing_extensions.Literal[micro, macro,
                                   weighted, none] | None = 'micro', multidim_average:
                                   typing_extensions.Literal[global, samplewise] | None = 'global', top_k:
                                   int | None = 1, ignore_index: int | None = None, validate_args: bool =
                                   True, **kwargs: ~typing.Any)
```

Compute the average *Hamming distance* (also known as Hamming loss).

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either 'binary', 'multiclass' or 'multilabel'. See the documentation of



BinaryHammingDistance, MulticlassHammingDistance and MultilabelHammingDistance for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([[0, 1], [1, 1]])
>>> preds = tensor([[0, 1], [0, 1]])
>>> hamming_distance = HammingDistance(task="multilabel", num_labels=2)
>>> hamming_distance(preds, target)
tensor(0.2500)
```

Initialize task metric.

### BinaryHammingDistance

```
class torchmetrics.classification.BinaryHammingDistance(threshold=0.5, multidim_average='global',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Compute the average *Hamming distance* (also known as Hamming loss) for binary tasks.

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): An int or float tensor of shape  $(N, \dots)$ . If preds is a floating point tensor with values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (**Tensor**): An int tensor of shape  $(N, \dots)$ .

As output to forward and compute the metric returns the following output:

- **bhd** (**Tensor**): A tensor whose returned shape depends on the **multidim\_average** arguments:
  - If **multidim\_average** is set to **global**, the metric returns a scalar value.
  - If **multidim\_average** is set to **samplewise**, the metric returns  $(N,)$  vector consisting of a scalar value per sample.

#### Parameters

- **threshold** (**float**) – Threshold for transforming probability to binary  $\{0,1\}$  predictions
- **multidim\_average** (**Literal** $['global', 'samplewise']$ ) – Defines how additionally dimensions  $\dots$  should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the  $N$  axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional** $[\text{int}]$ ) – Specifies a target value that is ignored and does not contribute to the metric calculation

- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryHammingDistance
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryHammingDistance()
>>> metric(preds, target)
tensor(0.3333)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryHammingDistance
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryHammingDistance()
>>> metric(preds, target)
tensor(0.3333)
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import BinaryHammingDistance
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...               [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = BinaryHammingDistance(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.6667, 0.8333])
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure object and Axes object

#### Raises

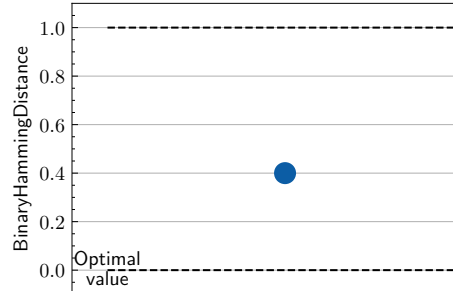
**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> from torch import rand, randint
>>> from torchmetrics.classification import BinaryHammingDistance
>>> metric = BinaryHammingDistance()
```

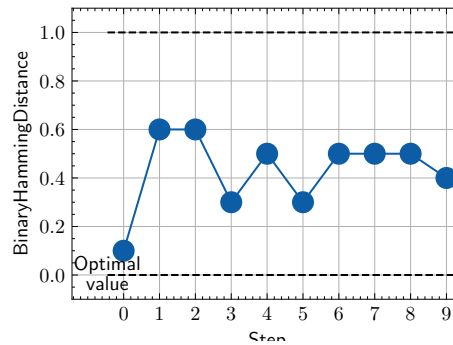
(continues on next page)

(continued from previous page)

```
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torch import rand, randint
>>> from torchmetrics.classification import BinaryHammingDistance
>>> metric = BinaryHammingDistance()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassHammingDistance

```
class torchmetrics.classification.MulticlassHammingDistance(num_classes, top_k=1,
                                                            average='macro',
                                                            multidim_average='global',
                                                            ignore_index=None,
                                                            validate_args=True, **kwargs)
```

Compute the average *Hamming distance* (also known as Hamming loss) for multiclass tasks.

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int tensor of shape (N, ...) or float tensor of shape (N, C, ...). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** ([Tensor](#)): An int tensor of shape (N, ...).

As output to forward and compute the metric returns the following output:

- **mchd** ([Tensor](#)): A tensor whose returned shape depends on the `average` and `multidim_average` arguments:
  - If `multidim_average` is set to `global`:
    - \* If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
    - \* If `average=None/'none'`, the shape will be (C,)
  - If `multidim_average` is set to `samplewise`:
    - \* If `average='micro'/'macro'/'weighted'`, the shape will be (N,)
    - \* If `average=None/'none'`, the shape will be (N, C)

### Parameters

- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassHammingDistance
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassHammingDistance(num_classes=3)
>>> metric(preds, target)
tensor(0.1667)
>>> mchd = MulticlassHammingDistance(num_classes=3, average=None)
>>> mchd(preds, target)
tensor([0.5000, 0.0000, 0.0000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MulticlassHammingDistance
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> metric = MulticlassHammingDistance(num_classes=3)
>>> metric(preds, target)
tensor(0.1667)
>>> mchd = MulticlassHammingDistance(num_classes=3, average=None)
>>> mchd(preds, target)
tensor([0.5000, 0.0000, 0.0000])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MulticlassHammingDistance
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassHammingDistance(num_classes=3, multidim_average=
↳ 'samplewise')
>>> metric(preds, target)
tensor([0.5000, 0.7222])
>>> mchd = MulticlassHammingDistance(num_classes=3, multidim_average='samplewise
↳ ', average=None)
>>> mchd(preds, target)
tensor([[0.0000, 1.0000, 0.5000],
        [1.0000, 0.6667, 0.5000]])
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

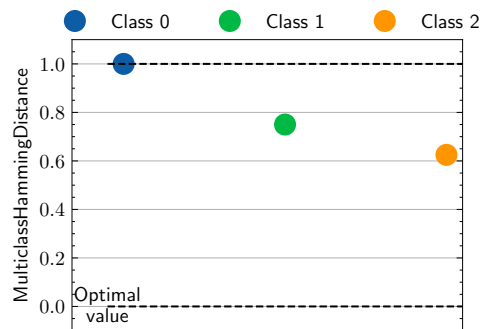
`Tuple[Figure, Union[Axes, ndarray]]`

**Returns**

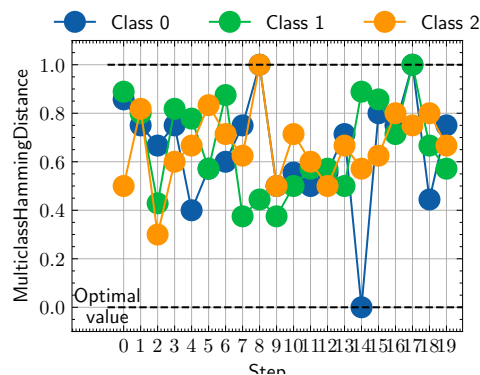
Figure object and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value per class
>>> from torch import randint
>>> from torchmetrics.classification import MulticlassHammingDistance
>>> metric = MulticlassHammingDistance(num_classes=3, average=None)
>>> metric.update(randint(3, (20,)), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting a multiple values per class
>>> from torch import randint
>>> from torchmetrics.classification import MulticlassHammingDistance
>>> metric = MulticlassHammingDistance(num_classes=3, average=None)
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randint(3, (20,)), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelHammingDistance

```
class torchmetrics.classification.MultilabelHammingDistance(num_labels, threshold=0.5,
                                                            average='macro',
                                                            multidim_average='global',
                                                            ignore_index=None,
                                                            validate_args=True, **kwargs)
```

Compute the average *Hamming distance* (also known as Hamming loss) for multilabel tasks.

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): An int tensor or float tensor of shape  $(N, C, \dots)$ . If preds is a floating point tensor with values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (**Tensor**): An int tensor of shape  $(N, C, \dots)$ .

As output to forward and compute the metric returns the following output:

- **mlhd** (**Tensor**): A tensor whose returned shape depends on the **average** and **multidim\_average** arguments:
  - If **multidim\_average** is set to **global**:
    - \* If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
    - \* If **average**=None/'none', the shape will be  $(C,)$
  - If **multidim\_average** is set to **samplewise**:
    - \* If **average**='micro'/'macro'/'weighted', the shape will be  $(N,)$
    - \* If **average**=None/'none', the shape will be  $(N, C)$

### Parameters

- **num\_labels** (**int**) – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **average** (**Optional**[**Literal**['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension

- **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelHammingDistance
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelHammingDistance(num_labels=3)
>>> metric(preds, target)
tensor(0.3333)
>>> mlhd = MultilabelHammingDistance(num_labels=3, average=None)
>>> mlhd(preds, target)
tensor([0.0000, 0.5000, 0.5000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelHammingDistance
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelHammingDistance(num_labels=3)
>>> metric(preds, target)
tensor(0.3333)
>>> mlhd = MultilabelHammingDistance(num_labels=3, average=None)
>>> mlhd(preds, target)
tensor([0.0000, 0.5000, 0.5000])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MultilabelHammingDistance
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = MultilabelHammingDistance(num_labels=3, multidim_average=
↳ 'samplewise')
>>> metric(preds, target)
tensor([0.6667, 0.8333])
>>> mlhd = MultilabelHammingDistance(num_labels=3, multidim_average='samplewise
↳ ', average=None)
>>> mlhd(preds, target)
tensor([[0.5000, 0.5000, 1.0000],
        [1.0000, 1.0000, 0.5000]])
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

**Parameters**



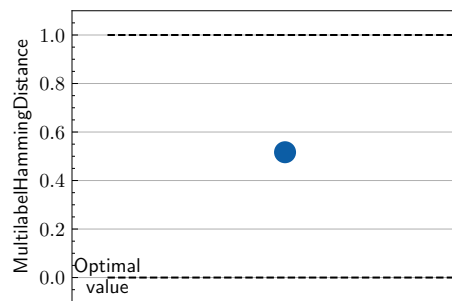
- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

Figure and Axes object

**Raises****ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> # Example plotting a single value
>>> from torch import rand, randint
>>> from torchmetrics.classification import MultilabelHammingDistance
>>> metric = MultilabelHammingDistance(num_labels=3)
>>> metric.update(randint(2, (20, 3)), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```

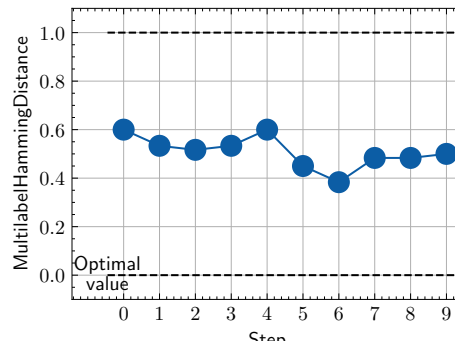


```
>>> # Example plotting multiple values
>>> from torch import rand, randint
>>> from torchmetrics.classification import MultilabelHammingDistance
>>> metric = MultilabelHammingDistance(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2, (20, 3)), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```

## 1.35.2 Functional Interface

### hamming\_distance

`torchmetrics.functional.hamming_distance(preds, target, task, threshold=0.5, num_classes=None, num_labels=None, average='micro', multidim_average='global', top_k=1, ignore_index=None, validate_args=True)`



Compute the average *Hamming distance* (also known as Hamming loss). :rtype: **Tensor**

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of `binary_hamming_distance()`, `multiclass_hamming_distance()` and `multilabel_hamming_distance()` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([[0, 1], [1, 1]])
>>> preds = tensor([[0, 1], [0, 1]])
>>> hamming_distance(preds, target, task="binary")
tensor(0.2500)
```

### binary\_hamming\_distance

`torchmetrics.functional.classification.binary_hamming_distance(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Compute the average *Hamming distance* (also known as Hamming loss) for binary tasks.

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

**Parameters**

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type**`Tensor`**Returns**

If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_hamming_distance
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> binary_hamming_distance(preds, target)
tensor(0.3333)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_hamming_distance
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_hamming_distance(preds, target)
tensor(0.3333)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import binary_hamming_distance
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...               [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> binary_hamming_distance(preds, target, multidim_average='samplewise')
tensor([0.6667, 0.8333])
```

## multiclass\_hamming\_distance

```
torchmetrics.functional.classification.multiclass_hamming_distance(preds, target, num_classes,
                                                                    average='macro', top_k=1,
                                                                    multidim_average='global',
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Compute the average *Hamming distance* (also known as Hamming loss) for multiclass tasks.

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor

- If average=None/'none', the shape will be (C,)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N,)
  - If average=None/'none', the shape will be (N, C)

**Return type**

The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_hamming_
↳ distance
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_hamming_distance(preds, target, num_classes=3)
tensor(0.1667)
>>> multiclass_hamming_distance(preds, target, num_classes=3, average=None)
tensor([0.5000, 0.0000, 0.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_hamming_
↳ distance
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> multiclass_hamming_distance(preds, target, num_classes=3)
tensor(0.1667)
>>> multiclass_hamming_distance(preds, target, num_classes=3, average=None)
tensor([0.5000, 0.0000, 0.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_hamming_
↳ distance
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_hamming_distance(preds, target, num_classes=3, multidim_average=
↳ 'samplewise')
tensor([0.5000, 0.7222])
>>> multiclass_hamming_distance(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.0000, 1.0000, 0.5000],
        [1.0000, 0.6667, 0.5000]])
```

## multilabel\_hamming\_distance

```
torchmetrics.functional.classification.multilabel_hamming_distance(preds, target, num_labels,
                                                                    threshold=0.5,
                                                                    average='macro',
                                                                    multidim_average='global',
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Compute the average *Hamming distance* (also known as Hamming loss) for multilabel tasks.

$$\text{Hamming distance} = \frac{1}{N \cdot L} \sum_i^N \sum_l^L 1(y_{il} \neq \hat{y}_{il})$$

Where  $y$  is a tensor of target values,  $\hat{y}$  is a tensor of predictions, and  $\bullet_{il}$  refers to the  $l$ -th label of the  $i$ -th sample of that tensor.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

- If **multidim\_average** is set to **global**:

- If average='micro'/'macro'/'weighted', the output will be a scalar tensor
- If average=None/'none', the shape will be (C,)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N,)
  - If average=None/'none', the shape will be (N, C)

**Return type**

The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_hamming_
↳ distance
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_hamming_distance(preds, target, num_labels=3)
tensor(0.3333)
>>> multilabel_hamming_distance(preds, target, num_labels=3, average=None)
tensor([0.0000, 0.5000, 0.5000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_hamming_
↳ distance
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_hamming_distance(preds, target, num_labels=3)
tensor(0.3333)
>>> multilabel_hamming_distance(preds, target, num_labels=3, average=None)
tensor([0.0000, 0.5000, 0.5000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multilabel_hamming_
↳ distance
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> multilabel_hamming_distance(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0.6667, 0.8333])
>>> multilabel_hamming_distance(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.5000, 0.5000, 1.0000],
        [1.0000, 1.0000, 0.5000]])
```

## 1.36 Hinge Loss

### 1.36.1 Module Interface

```
class torchmetrics.HingeLoss(task: typing_extensions.Literal[binary, multiclass], num_classes: int | None =
    None, squared: bool = False, multiclass_mode:
    typing_extensions.Literal[crammer - singer, one - vs - all] | None =
    'crammer-singer', ignore_index: int | None = None, validate_args: bool =
    True, **kwargs: Any)
```

Compute the mean *Hinge loss* typically used for Support Vector Machines (SVMs).

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary' or 'multiclass'. See the documentation of BinaryHingeLoss and MulticlassHingeLoss for the specific details of each argument influence and examples.

**Legacy Example:**

```
>>> from torch import tensor
>>> target = tensor([0, 1, 1])
>>> preds = tensor([0.5, 0.7, 0.1])
>>> hinge = HingeLoss(task="binary")
>>> hinge(preds, target)
tensor(0.9000)
```

```
>>> target = tensor([0, 1, 2])
>>> preds = tensor([[ -1.0,  0.9,  0.2], [ 0.5, -1.1,  0.8], [ 2.2, -0.5,  0.3]])
>>> hinge = HingeLoss(task="multiclass", num_classes=3)
>>> hinge(preds, target)
tensor(1.5551)
```

```
>>> target = tensor([0, 1, 2])
>>> preds = tensor([[ -1.0,  0.9,  0.2], [ 0.5, -1.1,  0.8], [ 2.2, -0.5,  0.3]])
>>> hinge = HingeLoss(task="multiclass", num_classes=3, multiclass_mode="one-vs-
↳ all")
>>> hinge(preds, target)
tensor([1.3743, 1.1945, 1.2359])
```

Initialize task metric.

### BinaryHingeLoss

```
class torchmetrics.classification.BinaryHingeLoss(squared=False, ignore_index=None,
    validate_args=True, **kwargs)
```

Compute the mean *Hinge loss* typically used for Support Vector Machines (SVMs) for binary tasks.

$$\text{Hinge loss} = \max(0, 1 - y \times \hat{y})$$

Where  $y \in -1, 1$  is the target, and  $\hat{y} \in \mathbb{R}$  is the prediction.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): A float tensor of shape (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.



- **target** (`Tensor`): An int tensor of shape (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **bhl** (`Tensor`): A tensor containing the hinge loss.

#### Parameters

- **squared** (`bool`) – If True, this will compute the squared hinge loss. Otherwise, computes the regular hinge loss.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torchmetrics.classification import BinaryHingeLoss
>>> preds = torch.tensor([0.25, 0.25, 0.55, 0.75, 0.75])
>>> target = torch.tensor([0, 0, 1, 1, 1])
>>> bhl = BinaryHingeLoss()
>>> bhl(preds, target)
tensor(0.6900)
>>> bhl = BinaryHingeLoss(squared=True)
>>> bhl(preds, target)
tensor(0.6905)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

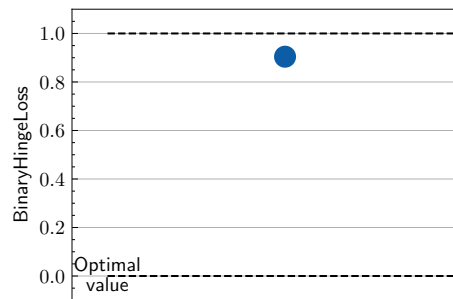
#### Returns

Figure object and Axes object

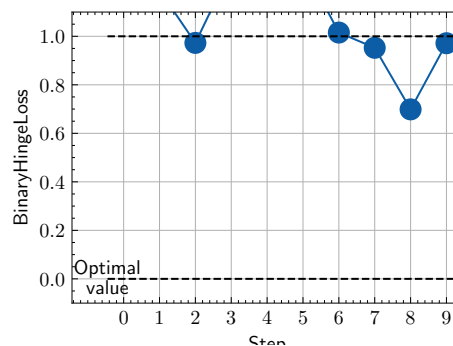
#### Raises

`ModuleNotFoundError` – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> from torch import rand, randint
>>> from torchmetrics.classification import BinaryHingeLoss
>>> metric = BinaryHingeLoss()
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torch import rand, randint
>>> from torchmetrics.classification import BinaryHingeLoss
>>> metric = BinaryHingeLoss()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassHingeLoss

```
class torchmetrics.classification.MulticlassHingeLoss(num_classes, squared=False,
                                                    multiclass_mode='crammer-singer',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Compute the mean *Hinge loss* typically used for Support Vector Machines (SVMs) for multiclass tasks.

The metric can be computed in two ways. Either, the definition by Crammer and Singer is used:

$$\text{Hinge loss} = \max \left( 0, 1 - \hat{y}_y + \max_{i \neq y}(\hat{y}_i) \right)$$

Where  $y \in 0, \dots, C$  is the target class (where  $C$  is the number of classes), and  $\hat{y} \in \mathbb{R}^C$  is the predicted output per class. Alternatively, the metric can also be computed in one-vs-all approach, where each class is valued against all other classes in a binary fashion.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`Tensor`): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** (`Tensor`): An int tensor of shape  $(N, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain values in the  $[0, n\_classes-1]$  range (except if `ignore_index` is specified).

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **mchl** (`Tensor`): A tensor containing the multi-class hinge loss.

#### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **squared** (`bool`) – If True, this will compute the squared hinge loss. Otherwise, computes the regular hinge loss.
- **multiclass\_mode** (`Literal['crammer-singer', 'one-vs-all']`) – Determines how to compute the metric
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torchmetrics.classification import MulticlassHingeLoss
>>> preds = torch.tensor([[0.25, 0.20, 0.55],
...                       [0.55, 0.05, 0.40],
...                       [0.10, 0.30, 0.60],
...                       [0.90, 0.05, 0.05]])
>>> target = torch.tensor([0, 1, 2, 0])
>>> mchl = MulticlassHingeLoss(num_classes=3)
>>> mchl(preds, target)
tensor(0.9125)
>>> mchl = MulticlassHingeLoss(num_classes=3, squared=True)
>>> mchl(preds, target)
tensor(1.1131)
>>> mchl = MulticlassHingeLoss(num_classes=3, multiclass_mode='one-vs-all')
```

(continues on next page)

(continued from previous page)

```
>>> mchl(preds, target)
tensor([0.8750, 1.1250, 1.1000])
```

**plot**(val=None, ax=None)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

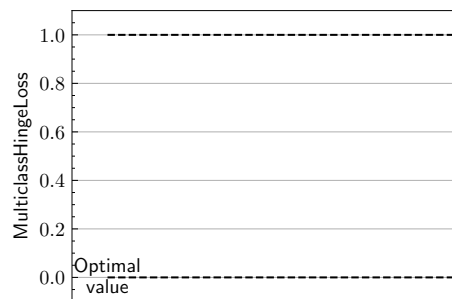
#### Returns

Figure object and Axes object

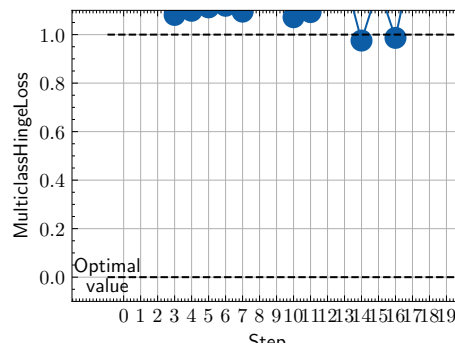
#### Raises

**`ModuleNotFoundError`** – If *matplotlib* is not installed

```
>>> # Example plotting a single value per class
>>> from torch import randint, randn
>>> from torchmetrics.classification import MulticlassHingeLoss
>>> metric = MulticlassHingeLoss(num_classes=3)
>>> metric.update(randn(20, 3), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting a multiple values per class
>>> from torch import randint, randn
>>> from torchmetrics.classification import MulticlassHingeLoss
>>> metric = MulticlassHingeLoss(num_classes=3)
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randn(20, 3), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.36.2 Functional Interface

`torchmetrics.functional.hinge_loss(preds, target, task, num_classes=None, squared=False, multiclass_mode='crammer-singer', ignore_index=None, validate_args=True)`

Compute the mean *Hinge loss* typically used for Support Vector Machines (SVMs).

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either 'binary' or 'multiclass'. See the documentation of `binary_hinge_loss()` and `multiclass_hinge_loss()` for the specific details of each argument influence and examples.

**Return type**  
Tensor

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([0, 1, 1])
>>> preds = tensor([0.5, 0.7, 0.1])
>>> hinge_loss(preds, target, task="binary")
tensor(0.9000)
```

```
>>> target = tensor([0, 1, 2])
>>> preds = tensor([[ -1.0,  0.9,  0.2], [ 0.5, -1.1,  0.8], [ 2.2, -0.5,  0.3]])
>>> hinge_loss(preds, target, task="multiclass", num_classes=3)
tensor(1.5551)
```

```
>>> target = tensor([0, 1, 2])
>>> preds = tensor([[ -1.0,  0.9,  0.2], [ 0.5, -1.1,  0.8], [ 2.2, -0.5,  0.3]])
>>> hinge_loss(preds, target, task="multiclass", num_classes=3, multiclass_mode=
↳ "one-vs-all")
tensor([1.3743, 1.1945, 1.2359])
```

## binary\_hinge\_loss

```
torchmetrics.functional.classification.binary_hinge_loss(preds, target, squared=False,
                                                         ignore_index=None,
                                                         validate_args=False)
```

Compute the mean *Hinge loss* typically used for Support Vector Machines (SVMs) for binary tasks.

$$\text{Hinge loss} = \max(0, 1 - y \times \hat{y})$$

Where  $y \in -1, 1$  is the target, and  $\hat{y} \in \mathbb{R}$  is the prediction.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **squared** (bool) – If True, this will compute the squared hinge loss. Otherwise, computes the regular hinge loss.
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

### Return type

Tensor

### Example

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_hinge_loss
>>> preds = tensor([0.25, 0.25, 0.55, 0.75, 0.75])
>>> target = tensor([0, 0, 1, 1, 1])
>>> binary_hinge_loss(preds, target)
tensor(0.6900)
>>> binary_hinge_loss(preds, target, squared=True)
tensor(0.6905)
```

## multiclass\_hinge\_loss

```
torchmetrics.functional.classification.multiclass_hinge_loss(preds, target, num_classes,
                                                             squared=False,
                                                             multiclass_mode='crammer-singer',
                                                             ignore_index=None,
                                                             validate_args=False)
```

Compute the mean *Hinge loss* typically used for Support Vector Machines (SVMs) for multiclass tasks.

The metric can be computed in two ways. Either, the definition by Crammer and Singer is used:

$$\text{Hinge loss} = \max \left( 0, 1 - \hat{y}_y + \max_{i \neq y}(\hat{y}_i) \right)$$

Where  $y \in 0, \dots, C$  is the target class (where  $C$  is the number of classes), and  $\hat{y} \in \mathbb{R}^C$  is the predicted output per class. Alternatively, the metric can also be computed in one-vs-all approach, where each class is valued against all other classes in a binary fashion.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **squared** ([bool](#)) – If True, this will compute the squared hinge loss. Otherwise, computes the regular hinge loss.
- **multiclass\_mode** ([Literal](#)['crammer-singer', 'one-vs-all']) – Determines how to compute the metric
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Return type

[Tensor](#)

### Example

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_hinge_loss
>>> preds = tensor([[0.25, 0.20, 0.55],
...                 [0.55, 0.05, 0.40],
...                 [0.10, 0.30, 0.60],
...                 [0.90, 0.05, 0.05]])
>>> target = tensor([0, 1, 2, 0])
>>> multiclass_hinge_loss(preds, target, num_classes=3)
tensor(0.9125)
>>> multiclass_hinge_loss(preds, target, num_classes=3, squared=True)
tensor(1.1131)
>>> multiclass_hinge_loss(preds, target, num_classes=3, multiclass_mode='one-vs-all
↪')
tensor([0.8750, 1.1250, 1.1000])
```

## 1.37 Jaccard Index

### 1.37.1 Module Interface

#### JaccardIndex

```
class torchmetrics.JaccardIndex(task: typing_extensions.Literal[binary, multiclass, multilabel], threshold:
    float = 0.5, num_classes: int | None = None, num_labels: int | None =
    None, average: typing_extensions.Literal[micro, macro, weighted, none] |
    None = 'macro', ignore_index: int | None = None, validate_args: bool =
    True, **kwargs: Any)
```

Calculate the Jaccard index for multilabel tasks.

The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of BinaryJaccardIndex, MulticlassJaccardIndex and MultilabelJaccardIndex for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import randint, tensor
>>> target = randint(0, 2, (10, 25, 25))
>>> pred = tensor(target)
>>> pred[2:5, 7:13, 9:15] = 1 - pred[2:5, 7:13, 9:15]
>>> jaccard = JaccardIndex(task="multiclass", num_classes=2)
>>> jaccard(pred, target)
tensor(0.9660)
```

Initialize task metric.



## BinaryJaccardIndex

```
class torchmetrics.classification.BinaryJaccardIndex(threshold=0.5, ignore_index=None,
                                                    validate_args=True, **kwargs)
```

Calculate the Jaccard index for binary tasks.

The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A int or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** ([Tensor](#)): An int tensor of shape (N, ...).

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **bji** ([Tensor](#)): A tensor containing the Binary Jaccard Index.

### Parameters

- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryJaccardIndex
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> metric = BinaryJaccardIndex()
>>> metric(preds, target)
tensor(0.5000)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinaryJaccardIndex
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0.35, 0.85, 0.48, 0.01])
>>> metric = BinaryJaccardIndex()
>>> metric(preds, target)
tensor(0.5000)
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

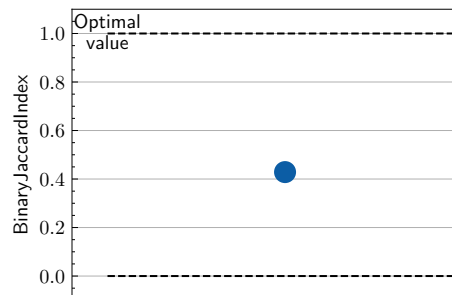
#### Returns

Figure object and Axes object

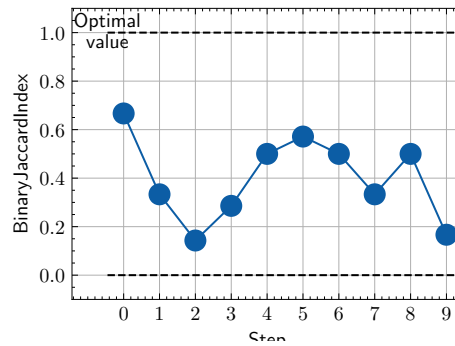
#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> # Example plotting a single value
>>> from torch import rand, randint
>>> from torchmetrics.classification import BinaryJaccardIndex
>>> metric = BinaryJaccardIndex()
>>> metric.update(rand(10), randint(2, (10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torch import rand, randint
>>> from torchmetrics.classification import BinaryJaccardIndex
>>> metric = BinaryJaccardIndex()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2, (10,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassJaccardIndex

```
class torchmetrics.classification.MulticlassJaccardIndex(num_classes, average='macro',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Calculate the Jaccard index for multiclass tasks.

The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): A int tensor of shape (N, ...) or float tensor of shape (N, C, ...). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (**Tensor**): An int tensor of shape (N, ...).

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to forward and compute the metric returns the following output:

- **mcji** (**Tensor**): A tensor containing the Multi-class Jaccard Index.

### Parameters

- **num\_classes** (**int**) – Integer specifying the number of classes
- **ignore\_index** (**Optional[int]**) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **average** (**Optional[Literal['micro', 'macro', 'weighted', 'none']]**) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support

- "none" or None: calculates statistic for each label and applies no reduction
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (pred is integer tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassJaccardIndex
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassJaccardIndex(num_classes=3)
>>> metric(preds, target)
tensor(0.6667)
```

Example (pred is float tensor):

```
>>> from torchmetrics.classification import MulticlassJaccardIndex
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> metric = MulticlassJaccardIndex(num_classes=3)
>>> metric(preds, target)
tensor(0.6667)
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

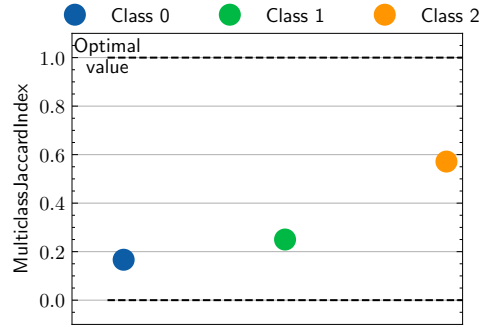
#### Returns

Figure object and Axes object

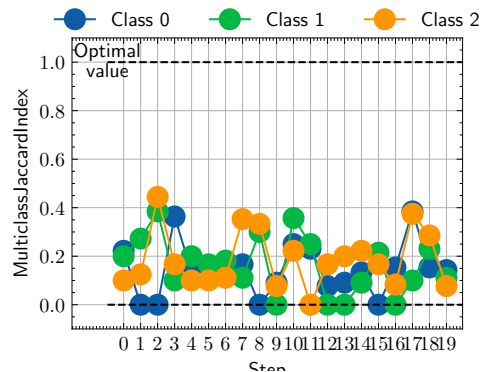
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value per class
>>> from torch import randint
>>> from torchmetrics.classification import MulticlassJaccardIndex
>>> metric = MulticlassJaccardIndex(num_classes=3, average=None)
>>> metric.update(randint(3, (20,)), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting a multiple values per class
>>> from torch import randint
>>> from torchmetrics.classification import MulticlassJaccardIndex
>>> metric = MulticlassJaccardIndex(num_classes=3, average=None)
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randint(3, (20,)), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelJaccardIndex

```
class torchmetrics.classification.MultilabelJaccardIndex(num_labels, threshold=0.5,
                                                         average='macro', ignore_index=None,
                                                         validate_args=True, **kwargs)
```

Calculate the Jaccard index for multilabel tasks.

The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): A int tensor or float tensor of shape (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.

- **target** ([Tensor](#)): An int tensor of shape (N, C, ...)

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to forward and compute the metric returns the following output:

- **mlji** ([Tensor](#)): A tensor containing the Multi-label Jaccard Index loss.

#### Parameters

- **num\_classes** – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **average** ([Optional\[Literall\['micro', 'macro', 'weighted', 'none'\]\]](#)) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelJaccardIndex
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelJaccardIndex(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MultilabelJaccardIndex
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelJaccardIndex(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

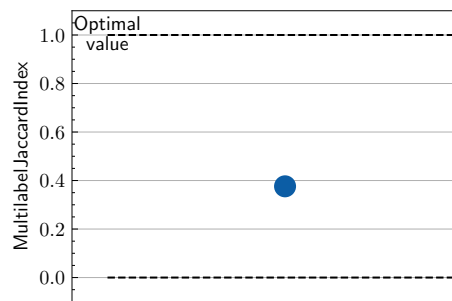
- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

Figure and Axes object

**Raises****`ModuleNotFoundError`** – If `matplotlib` is not installed

```
>>> # Example plotting a single value
>>> from torch import rand, randint
>>> from torchmetrics.classification import MultilabelJaccardIndex
>>> metric = MultilabelJaccardIndex(num_labels=3)
>>> metric.update(randint(2, (20, 3)), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```



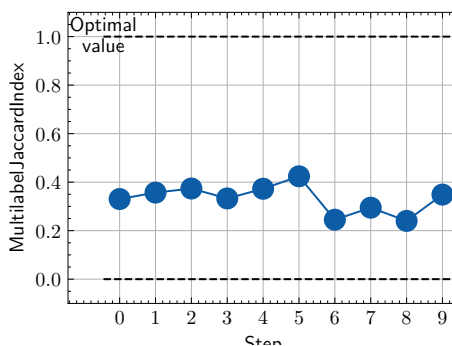
```
>>> # Example plotting multiple values
>>> from torch import rand, randint
>>> from torchmetrics.classification import MultilabelJaccardIndex
>>> metric = MultilabelJaccardIndex(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2, (20, 3)), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```

## 1.37.2 Functional Interface

### jaccard\_index

`torchmetrics.functional.jaccard_index(preds, target, task, threshold=0.5, num_classes=None, num_labels=None, average='macro', ignore_index=None, validate_args=True)`

Calculate the Jaccard index.



The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets: :rtype: [Tensor](#)

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of `binary_jaccard_index()`, `multiclass_jaccard_index()` and `multilabel_jaccard_index()` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import randint, tensor
>>> target = randint(0, 2, (10, 25, 25))
>>> pred = tensor(target)
>>> pred[2:5, 7:13, 9:15] = 1 - pred[2:5, 7:13, 9:15]
>>> jaccard_index(pred, target, task="multiclass", num_classes=2)
tensor(0.9660)
```

### binary\_jaccard\_index

`torchmetrics.functional.classification.binary_jaccard_index(preds, target, threshold=0.5, ignore_index=None, validate_args=True)`

Calculate the Jaccard index for binary tasks.

The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)



Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Return type

`Tensor`

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_jaccard_index
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> binary_jaccard_index(preds, target)
tensor(0.5000)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import binary_jaccard_index
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0.35, 0.85, 0.48, 0.01])
>>> binary_jaccard_index(preds, target)
tensor(0.5000)
```

### multiclass\_jaccard\_index

`torchmetrics.functional.classification.multiclass_jaccard_index(preds, target, num_classes, average='macro', ignore_index=None, validate_args=True)`

Calculate the Jaccard index for multiclass tasks.

The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Accepts the following input tensors:

- **preds**: (`N, ...`) (int tensor) or (`N, C, ...`) (float tensor). If preds is a floating point we apply `torch.argmax` along the `C` dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (`N, ...`)

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Return type

`Tensor`

**Example (pred is integer tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_jaccard_index
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_jaccard_index(preds, target, num_classes=3)
tensor(0.6667)
```

**Example (pred is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_jaccard_index
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                 [0.22, 0.61, 0.17],
...                 [0.71, 0.09, 0.20],
...                 [0.05, 0.82, 0.13]])
>>> multiclass_jaccard_index(preds, target, num_classes=3)
tensor(0.6667)
```

## multilabel\_jaccard\_index

```
torchmetrics.functional.classification.multilabel_jaccard_index(preds, target, num_labels,
                                                                threshold=0.5, average='macro',
                                                                ignore_index=None,
                                                                validate_args=True)
```

Calculate the Jaccard index for multilabel tasks.

The *Jaccard index* (also known as the intersection over union or jaccard similarity coefficient) is an statistic that can be used to determine the similarity and diversity of a sample set. It is defined as the size of the intersection divided by the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** (**Tensor**) – Tensor with predictions
- **target** (**Tensor**) – Tensor with true labels
- **num\_labels** (**int**) – Integer specifying the number of labels
- **threshold** (**float**) – Threshold for transforming probability to binary (0,1) predictions
- **average** (**Optional[Literal['micro', 'macro', 'weighted', 'none']]**) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **ignore\_index** (**Optional[int]**) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.
- **kwargs** – Additional keyword arguments, see *Advanced metric settings* for more info.

### Return type

**Tensor**

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_jaccard_index
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_jaccard_index(preds, target, num_labels=3)
tensor(0.5000)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multilabel_jaccard_index
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_jaccard_index(preds, target, num_labels=3)
tensor(0.5000)
```

## 1.38 Label Ranking Average Precision

### 1.38.1 Module Interface

```
class torchmetrics.classification.MultilabelRankingAveragePrecision(num_labels,
                                                                    ignore_index=None,
                                                                    validate_args=True,
                                                                    **kwargs)
```

Compute label ranking average precision score for multilabel data [1].

The score is the average over each ground truth label assigned to each sample of the ratio of true vs. total labels with lower score. Best score is 1.

As input to forward and update the metric accepts the following input:

- **preds** (`Tensor`): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (`Tensor`): An int tensor of shape  $(N, C, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if *ignore\_index* is specified).

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to forward and compute the metric returns the following output:

- **mlrap** (`Tensor`): A tensor containing the multilabel ranking average precision.

#### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

## Example

```
>>> from torchmetrics.classification import MultilabelRankingAveragePrecision
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> mlapr = MultilabelRankingAveragePrecision(num_labels=5)
>>> mlapr(preds, target)
tensor(0.7744)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

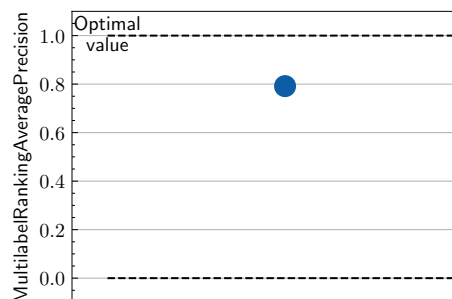
### Returns

Figure object and Axes object

### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelRankingAveragePrecision
>>> metric = MultilabelRankingAveragePrecision(num_labels=3)
>>> metric.update(rand(20, 3), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```

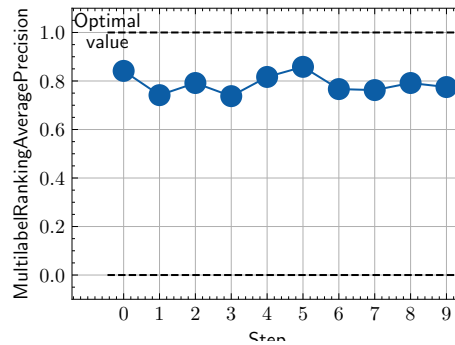


```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import MultilabelRankingAveragePrecision
>>> metric = MultilabelRankingAveragePrecision(num_labels=3)
>>> values = [ ]
```

(continues on next page)

(continued from previous page)

```
>>> for _ in range(10):
...     values.append(metric(rand(20, 3), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```



## 1.38.2 Functional Interface

`torchmetrics.functional.classification.multilabel_ranking_average_precision`(*preds*, *target*, *num\_labels*, *ignore\_index=None*, *validate\_args=True*)

Compute label ranking average precision score for multilabel data [1].

The score is the average over each ground truth label assigned to each sample of the ratio of true vs. total labels with lower score. Best score is 1.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_labels** (int) – Integer specifying the number of labels
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

### Return type

Tensor

### Example

```
>>> from torchmetrics.functional.classification import multilabel_ranking_average_
    precision
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> multilabel_ranking_average_precision(preds, target, num_labels=5)
tensor(0.7744)
```

### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

## 1.39 Label Ranking Loss

### 1.39.1 Module Interface

```
class torchmetrics.classification.MultilabelRankingLoss(num_labels, ignore_index=None,
                                                         validate_args=True, **kwargs)
```

Compute the label ranking loss for multilabel data [1].

The score corresponds to the average number of label pairs that are incorrectly ordered given some predictions weighted by the size of the label set and the number of labels not in the label set. The best score is 0.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (**Tensor**): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (**Tensor**): An int tensor of shape  $(N, C, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if `ignore_index` is specified).

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **mlr1** (**Tensor**): A tensor containing the multilabel ranking loss.

#### Parameters

- **preds** – Tensor with predictions
- **target** – Tensor with true labels
- **num\_labels** (**int**) – Integer specifying the number of labels
- **ignore\_index** (**Optional[int]**) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

## Example

```
>>> from torchmetrics.classification import MultilabelRankingLoss
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> mrlr = MultilabelRankingLoss(num_labels=5)
>>> mrlr(preds, target)
tensor(0.4167)
```

**plot**(*val=None*, *ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

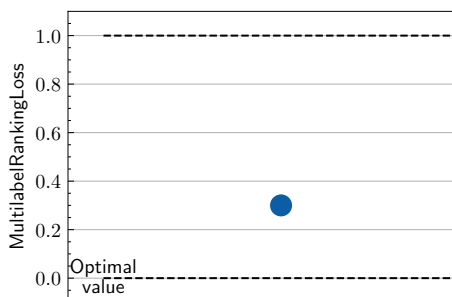
### Returns

Figure object and Axes object

### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelRankingLoss
>>> metric = MultilabelRankingLoss(num_labels=3)
>>> metric.update(rand(20, 3), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```



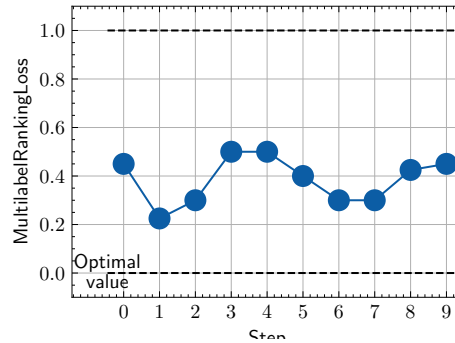
```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import MultilabelRankingLoss
>>> metric = MultilabelRankingLoss(num_labels=3)
>>> values = [ ]
```

(continues on next page)



(continued from previous page)

```
>>> for _ in range(10):
...     values.append(metric(rand(20, 3), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.39.2 Functional Interface

`torchmetrics.functional.classification.multilabel_ranking_loss(preds, target, num_labels, ignore_index=None, validate_args=True)`

Compute the label ranking loss for multilabel data [1].

The score corresponds to the average number of label pairs that are incorrectly ordered given some predictions weighted by the size of the label set and the number of labels not in the label set. The best score is 0.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_labels** (int) – Integer specifying the number of labels
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

#### Return type

Tensor

### Example

```
>>> from torchmetrics.functional.classification import multilabel_ranking_loss
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(10, 5)
>>> target = torch.randint(2, (10, 5))
>>> multilabel_ranking_loss(preds, target, num_labels=5)
tensor(0.4167)
```

### References

[1] Tsoumakas, G., Katakis, I., & Vlahavas, I. (2010). Mining multi-label data. In Data mining and knowledge discovery handbook (pp. 667-685). Springer US.

## 1.40 Matthews Correlation Coefficient

### 1.40.1 Module Interface

#### MatthewsCorrCoef

```
class torchmetrics.MatthewsCorrCoef(task: typing_extensions.Literal[binary, multiclass, multilabel],
                                     threshold: float = 0.5, num_classes: int | None = None, num_labels:
                                     int | None = None, ignore_index: int | None = None, validate_args:
                                     bool = True, **kwargs: Any)
```

Calculate [Matthews correlation coefficient](#) .

This metric measures the general correlation or quality of a classification.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of [BinaryMatthewsCorrCoef](#), [MulticlassMatthewsCorrCoef](#) and [MultilabelMatthewsCorrCoef](#) for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> matthews_corrcoef = MatthewsCorrCoef(task='binary')
>>> matthews_corrcoef(preds, target)
tensor(0.5774)
```

Initialize task metric.

## BinaryMatthewsCorrCoef

```
class torchmetrics.classification.BinaryMatthewsCorrCoef(threshold=0.5, ignore_index=None,
                                                         validate_args=True, **kwargs)
```

Calculate [Matthews correlation coefficient](#) for binary tasks.

This metric measures the general correlation or quality of a classification.

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A int tensor or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** ([Tensor](#)): An int tensor of shape (N, ...)

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **bmcc** ([Tensor](#)): A tensor containing the Binary Matthews Correlation Coefficient.

### Parameters

- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryMatthewsCorrCoef
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> metric = BinaryMatthewsCorrCoef()
>>> metric(preds, target)
tensor(0.5774)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinaryMatthewsCorrCoef
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0.35, 0.85, 0.48, 0.01])
>>> metric = BinaryMatthewsCorrCoef()
>>> metric(preds, target)
tensor(0.5774)
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

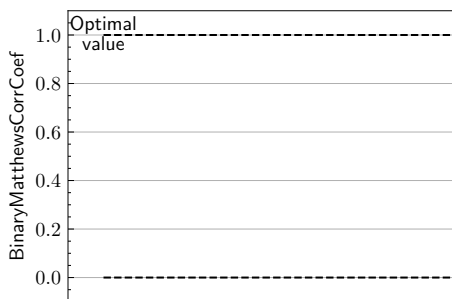
- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

Figure object and Axes object

**Raises****`ModuleNotFoundError`** – If `matplotlib` is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinaryMatthewsCorrCoef
>>> metric = BinaryMatthewsCorrCoef()
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryMatthewsCorrCoef
>>> metric = BinaryMatthewsCorrCoef()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```

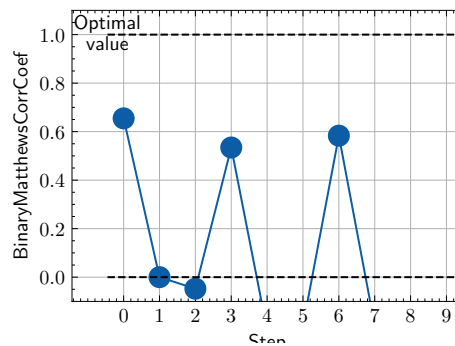
**MulticlassMatthewsCorrCoef**

```
class torchmetrics.classification.MulticlassMatthewsCorrCoef(num_classes, ignore_index=None,
                                                             validate_args=True, **kwargs)
```

Calculate [Matthews correlation coefficient](#) for multiclass tasks.

This metric measures the general correlation or quality of a classification.

As input to forward and update the metric accepts the following input:



- **preds** ([Tensor](#)): A int tensor of shape (N, ...) or float tensor of shape (N, C, ...). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** ([Tensor](#)): An int tensor of shape (N, ...)

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to forward and compute the metric returns the following output:

- **mcmcc** ([Tensor](#)): A tensor containing the Multi-class Matthews Correlation Coefficient.

#### Parameters

- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (pred is integer tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassMatthewsCorrCoef
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassMatthewsCorrCoef(num_classes=3)
>>> metric(preds, target)
tensor(0.7000)
```

**Example (pred is float tensor):**

```
>>> from torchmetrics.classification import MulticlassMatthewsCorrCoef
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                 [0.22, 0.61, 0.17],
...                 [0.71, 0.09, 0.20],
...                 [0.05, 0.82, 0.13]])
```

(continues on next page)

(continued from previous page)

```
>>> metric = MulticlassMatthewsCorrCoef(num_classes=3)
>>> metric(preds, target)
tensor(0.7000)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

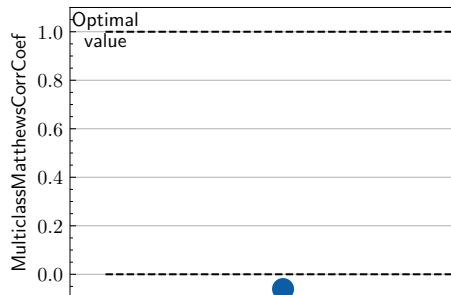
#### Returns

Figure object and Axes object

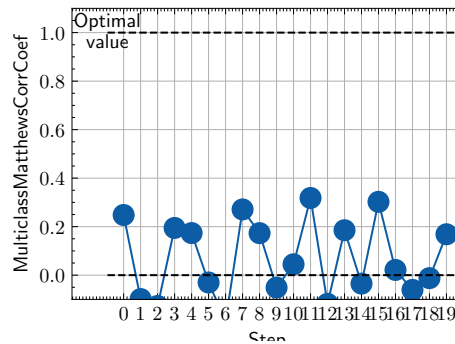
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randint
>>> # Example plotting a single value per class
>>> from torchmetrics.classification import MulticlassMatthewsCorrCoef
>>> metric = MulticlassMatthewsCorrCoef(num_classes=3)
>>> metric.update(randint(3, (20,)), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randint
>>> # Example plotting a multiple values per class
>>> from torchmetrics.classification import MulticlassMatthewsCorrCoef
>>> metric = MulticlassMatthewsCorrCoef(num_classes=3)
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randint(3, (20,)), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelMatthewsCorrCoef

```
class torchmetrics.classification.MultilabelMatthewsCorrCoef(num_labels, threshold=0.5,
                                                             ignore_index=None,
                                                             validate_args=True, **kwargs)
```

Calculate [Matthews correlation coefficient](#) for multilabel tasks.

This metric measures the general correlation or quality of a classification.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int or float tensor of shape  $(N, C, \dots)$ . If preds is a floating point tensor with values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** ([Tensor](#)): An int tensor of shape  $(N, C, \dots)$

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to forward and compute the metric returns the following output:

- **mlmcc** ([Tensor](#)): A tensor containing the Multi-label Matthews Correlation Coefficient.

### Parameters

- **num\_classes** – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelMatthewsCorrCoef
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
```

(continues on next page)

(continued from previous page)

```
>>> metric = MultilabelMatthewsCorrCoef(num_labels=3)
>>> metric(preds, target)
tensor(0.3333)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelMatthewsCorrCoef
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelMatthewsCorrCoef(num_labels=3)
>>> metric(preds, target)
tensor(0.3333)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

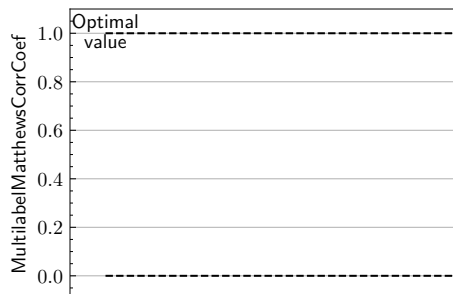
#### Returns

Figure object and Axes object

#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelMatthewsCorrCoef
>>> metric = MultilabelMatthewsCorrCoef(num_labels=3)
>>> metric.update(randint(2, (20, 3)), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```



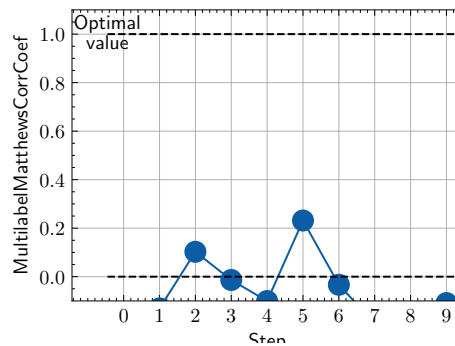
```
>>> from torch import rand, randint
>>> # Example plotting multiple values
```

(continues on next page)



(continued from previous page)

```
>>> from torchmetrics.classification import MultilabelMatthewsCorrCoef
>>> metric = MultilabelMatthewsCorrCoef(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2, (20, 3)), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```



## 1.40.2 Functional Interface

### matthews\_corrcoef

`torchmetrics.functional.matthews_corrcoef(preds, target, task, threshold=0.5, num_classes=None, num_labels=None, ignore_index=None, validate_args=True)`

Calculate [Matthews correlation coefficient](#).

This metric measures the general correlation or quality of a classification.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either `'binary'`, `'multiclass'` or `'multilabel'`. See the documentation of `binary_matthews_corrcoef()`, `multiclass_matthews_corrcoef()` and `multilabel_matthews_corrcoef()` for the specific details of each argument influence and examples.

#### Return type

[Tensor](#)

#### Legacy Example:

```
>>> from torch import tensor
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> matthews_corrcoef(preds, target, task="multiclass", num_classes=2)
tensor(0.5774)
```

## binary\_matthews\_corrcoef

```
torchmetrics.functional.classification.binary_matthews_corrcoef(preds, target, threshold=0.5,
                                                                ignore_index=None,
                                                                validate_args=True)
```

Calculate [Matthews correlation coefficient](#) for binary tasks.

This metric measures the general correlation or quality of a classification.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Return type

[Tensor](#)

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_matthews_corrcoef
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0, 1, 0, 0])
>>> binary_matthews_corrcoef(preds, target)
tensor(0.5774)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import binary_matthews_corrcoef
>>> target = tensor([1, 1, 0, 0])
>>> preds = tensor([0.35, 0.85, 0.48, 0.01])
>>> binary_matthews_corrcoef(preds, target)
tensor(0.5774)
```

## multiclass\_matthews\_corrcoef

```
torchmetrics.functional.classification.multiclass_matthews_corrcoef(preds, target, num_classes,
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Calculate [Matthews correlation coefficient](#) for multiclass tasks.

This metric measures the general correlation or quality of a classification.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Return type

[Tensor](#)

**Example (pred is integer tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_matthews_
    ↪corrcoef
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_matthews_corrcoef(preds, target, num_classes=3)
tensor(0.7000)
```

**Example (pred is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_matthews_
    ↪corrcoef
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                 [0.22, 0.61, 0.17],
...                 [0.71, 0.09, 0.20],
...                 [0.05, 0.82, 0.13]])
>>> multiclass_matthews_corrcoef(preds, target, num_classes=3)
tensor(0.7000)
```

## multilabel\_matthews\_corrcoef

```
torchmetrics.functional.classification.multilabel_matthews_corrcoef(preds, target, num_labels,
                                                                    threshold=0.5,
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Calculate [Matthews correlation coefficient](#) for multilabel tasks.

This metric measures the general correlation or quality of a classification.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

Additional dimension ... will be flattened into the batch dimension.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Return type

[Tensor](#)

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_matthews_
    ↪ corrcoef
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_matthews_corrcoef(preds, target, num_labels=3)
tensor(0.3333)
```

Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multilabel_matthews_
    ↪ corrcoef
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_matthews_corrcoef(preds, target, num_labels=3)
tensor(0.3333)
```

## 1.41 Precision

### 1.41.1 Module Interface

```
class torchmetrics.Precision(task: typing_extensions.Literal[binary, multiclass, multilabel], threshold: float
                             = 0.5, num_classes: int | None = None, num_labels: int | None = None,
                             average: typing_extensions.Literal[micro, macro, weighted, none] | None =
                             'micro', multidim_average: typing_extensions.Literal[global, samplewise] |
                             None = 'global', top_k: int | None = 1, ignore_index: int | None = None,
                             validate_args: bool = True, **kwargs: ~typing.Any)
```

Compute [Precision](#).

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively. The metric is only proper defined when  $\text{TP} + \text{FP} \neq 0$ . If this case is encountered for any class/label, the metric for that class/label will be set to 0 and the overall metric may therefore be affected in turn.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either 'binary', 'multiclass' or 'multilabel'. See the documentation of `BinaryPrecision`, `MulticlassPrecision()` and `MultilabelPrecision()` for the specific details of each argument influence and examples.

**Legacy Example:**

```
>>> from torch import tensor
>>> preds = tensor([2, 0, 2, 1])
>>> target = tensor([1, 1, 2, 0])
>>> precision = Precision(task="multiclass", average='macro', num_classes=3)
>>> precision(preds, target)
tensor(0.1667)
>>> precision = Precision(task="multiclass", average='micro', num_classes=3)
>>> precision(preds, target)
tensor(0.2500)
```

Initialize task metric.

### BinaryPrecision

```
class torchmetrics.classification.BinaryPrecision(threshold=0.5, multidim_average='global',
                                                  ignore_index=None, validate_args=True,
                                                  **kwargs)
```

Compute [Precision](#) for binary tasks.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively. The metric is only proper defined when  $\text{TP} + \text{FP} \neq 0$ . If this case is encountered a score of 0 is returned.

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A int or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.

- **target** (`Tensor`): An int tensor of shape (N, ...).

As output to forward and compute the metric returns the following output:

- **bp** (`Tensor`): If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

#### Parameters

- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryPrecision
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryPrecision()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinaryPrecision
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryPrecision()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import BinaryPrecision
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = BinaryPrecision(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.4000, 0.0000])
```

**plot** (`val=None, ax=None`)

Plot a single or multiple values from the metric.

#### Parameters

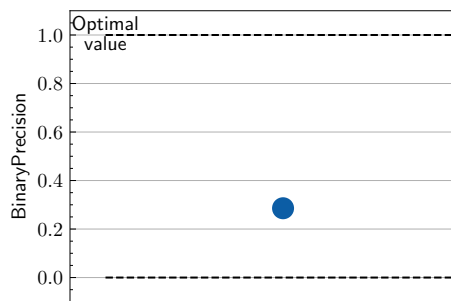
- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

Figure object and Axes object

**Raises****`ModuleNotFoundError`** – If `matplotlib` is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinaryPrecision
>>> metric = BinaryPrecision()
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot()
```



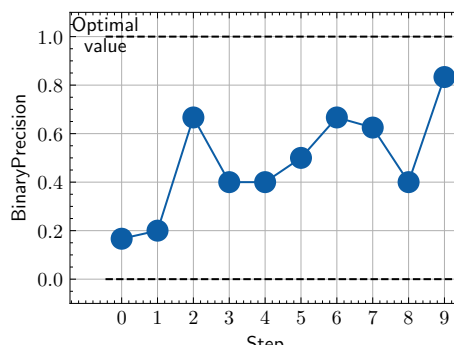
```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryPrecision
>>> metric = BinaryPrecision()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```

**MulticlassPrecision**

```
class torchmetrics.classification.MulticlassPrecision(num_classes, top_k=1, average='macro',
                                                    multidim_average='global',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Compute [Precision](#) for multiclass tasks.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$



Where TP and FP represent the number of true positives and false positives respectively. The metric is only proper defined when  $TP + FP \neq 0$ . If this case is encountered for any class, the metric for that class will be set to 0 and the overall metric may therefore be affected in turn.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): An int tensor of shape (N, ...) or float tensor of shape (N, C, ...). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (**Tensor**): An int tensor of shape (N, ...).

As output to forward and compute the metric returns the following output:

- **mcp** (**Tensor**): The returned shape depends on the `average` and `multidim_average` arguments:
  - If `multidim_average` is set to `global`:
    - \* If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
    - \* If `average=None/'none'`, the shape will be (C,)
  - If `multidim_average` is set to `samplewise`:
    - \* If `average='micro'/'macro'/'weighted'`, the shape will be (N,)
    - \* If `average=None/'none'`, the shape will be (N, C)

### Parameters

- **num\_classes** (**int**) – Integer specifying the number of classes
- **average** (**Optional** **Literal**['micro', 'macro', 'weighted', 'none']) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **top\_k** (**int**) – Number of highest probability or logit score predictions considered to find the correct label. Only works when `preds` contain probabilities/logits.
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:



- **global**: Additional dimensions are flattened along the batch dimension
- **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassPrecision
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassPrecision(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> mcp = MulticlassPrecision(num_classes=3, average=None)
>>> mcp(preds, target)
tensor([1.0000, 0.5000, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import MulticlassPrecision
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                 [0.22, 0.61, 0.17],
...                 [0.71, 0.09, 0.20],
...                 [0.05, 0.82, 0.13]])
>>> metric = MulticlassPrecision(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> mcp = MulticlassPrecision(num_classes=3, average=None)
>>> mcp(preds, target)
tensor([1.0000, 0.5000, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import MulticlassPrecision
>>> target = tensor([[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]])
>>> preds = tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> metric = MulticlassPrecision(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.3889, 0.2778])
>>> mcp = MulticlassPrecision(num_classes=3, multidim_average='samplewise',
↪ average=None)
>>> mcp(preds, target)
tensor([[0.6667, 0.0000, 0.5000],
        [0.0000, 0.5000, 0.3333]])
```

**plot** (`val=None, ax=None`)

Plot a single or multiple values from the metric.

**Parameters**

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

**Returns**

Figure object and Axes object

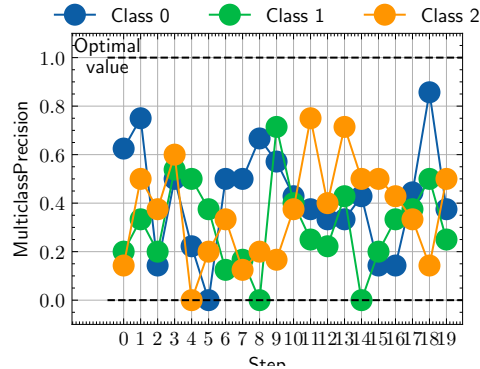
**Raises**

**`ModuleNotFoundError`** – If *matplotlib* is not installed

```
>>> from torch import randint
>>> # Example plotting a single value per class
>>> from torchmetrics.classification import MulticlassPrecision
>>> metric = MulticlassPrecision(num_classes=3, average=None)
>>> metric.update(randint(3, (20,)), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randint
>>> # Example plotting a multiple values per class
>>> from torchmetrics.classification import MulticlassPrecision
>>> metric = MulticlassPrecision(num_classes=3, average=None)
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randint(3, (20,)), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelPrecision

```
class torchmetrics.classification.MultilabelPrecision(num_labels, threshold=0.5, average='macro',
    multidim_average='global',
    ignore_index=None, validate_args=True,
    **kwargs)
```

Compute [Precision](#) for multilabel tasks.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively. The metric is only proper defined when  $\text{TP} + \text{FP} \neq 0$ . If this case is encountered for any label, the metric for that label will be set to 0 and the overall metric may therefore be affected in turn.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int tensor or float tensor of shape  $(N, C, \dots)$ . If preds is a floating point tensor with values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** ([Tensor](#)): An int tensor of shape  $(N, C, \dots)$ .

As output to forward and compute the metric returns the following output:

- **mlp** ([Tensor](#)): The returned shape depends on the **average** and **multidim\_average** arguments:
  - If **multidim\_average** is set to **global**:
    - \* If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
    - \* If **average**=None/'none', the shape will be  $(C,)$
  - If **multidim\_average** is set to **samplewise**:
    - \* If **average**='micro'/'macro'/'weighted', the shape will be  $(N,)$
    - \* If **average**=None/'none', the shape will be  $(N, C)$

### Parameters

- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:

- micro: Sum statistics over all labels
- macro: Calculate statistics for each label and average them
- weighted: calculates statistics for each label and computes weighted average using their support
- "none" or None: calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelPrecision
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelPrecision(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
>>> mlp = MultilabelPrecision(num_labels=3, average=None)
>>> mlp(preds, target)
tensor([1.0000, 0.0000, 0.5000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelPrecision
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelPrecision(num_labels=3)
>>> metric(preds, target)
tensor(0.5000)
>>> mlp = MultilabelPrecision(num_labels=3, average=None)
>>> mlp(preds, target)
tensor([1.0000, 0.0000, 0.5000])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MultilabelPrecision
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = MultilabelPrecision(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.3333, 0.0000])
>>> mlp = MultilabelPrecision(num_labels=3, multidim_average='samplewise',
```

(continues on next page)

(continued from previous page)

```

average=None)
>>> mlp(preds, target)
tensor([[0.5000, 0.5000, 0.0000],
        [0.0000, 0.0000, 0.0000]])

```

**plot**(val=None, ax=None)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure object and Axes object

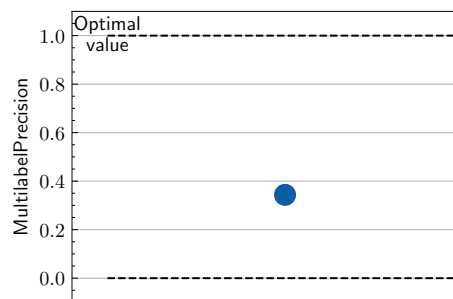
#### Raises

`ModuleNotFoundError` – If *matplotlib* is not installed

```

>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelPrecision
>>> metric = MultilabelPrecision(num_labels=3)
>>> metric.update(randint(2, (20, 3)), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()

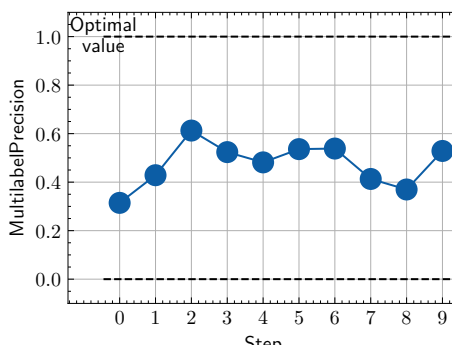
```



```

>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import MultilabelPrecision
>>> metric = MultilabelPrecision(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2, (20, 3)), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)

```



### 1.41.2 Functional Interface

`torchmetrics.functional.precision(preds, target, task, threshold=0.5, num_classes=None, num_labels=None, average='micro', multidim_average='global', top_k=1, ignore_index=None, validate_args=True)`

Compute Precision. :rtype: `Tensor`

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either 'binary', 'multiclass' or 'multilabel'. See the documentation of `binary_precision()`, `multiclass_precision()` and `multilabel_precision()` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> preds = tensor([2, 0, 2, 1])
>>> target = tensor([1, 1, 2, 0])
>>> precision(preds, target, task="multiclass", average='macro', num_classes=3)
tensor(0.1667)
>>> precision(preds, target, task="multiclass", average='micro', num_classes=3)
tensor(0.2500)
```

### binary\_precision

`torchmetrics.functional.classification.binary_precision(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Compute Precision for binary tasks.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

#### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

#### Return type

[Tensor](#)

#### Returns

If **multidim\_average** is set to **global**, the metric returns a scalar value. If **multidim\_average** is set to **samplewise**, the metric returns (N,) vector consisting of a scalar value per sample.

#### Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_precision
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> binary_precision(preds, target)
tensor(0.6667)
```

#### Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import binary_precision
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_precision(preds, target)
tensor(0.6667)
```

#### Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import binary_precision
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                 [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
```

(continues on next page)

(continued from previous page)

```
>>> binary_precision(preds, target, multidim_average='samplewise')
tensor([0.4000, 0.0000])
```

## multiclass\_precision

```
torchmetrics.functional.classification.multiclass_precision(preds, target, num_classes,
                                                            average='macro', top_k=1,
                                                            multidim_average='global',
                                                            ignore_index=None,
                                                            validate_args=True)
```

Compute [Precision](#) for multiclass tasks.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when `preds` contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.



**Returns**

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type**

The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_precision
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_precision(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_precision(preds, target, num_classes=3, average=None)
tensor([1.0000, 0.5000, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_precision
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> multiclass_precision(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_precision(preds, target, num_classes=3, average=None)
tensor([1.0000, 0.5000, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_precision
>>> target = tensor([[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]])
>>> preds = tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> multiclass_precision(preds, target, num_classes=3, multidim_average=
↳ 'samplewise')
tensor([0.3889, 0.2778])
>>> multiclass_precision(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.6667, 0.0000, 0.5000],
        [0.0000, 0.5000, 0.3333]])
```

## multilabel\_precision

```
torchmetrics.functional.classification.multilabel_precision(preds, target, num_labels,
                                                            threshold=0.5, average='macro',
                                                            multidim_average='global',
                                                            ignore_index=None,
                                                            validate_args=True)
```

Compute [Precision](#) for multilabel tasks.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Where TP and FP represent the number of true positives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - micro: Sum statistics over all labels
  - macro: Calculate statistics for each label and average them
  - weighted: calculates statistics for each label and computes weighted average using their support
  - "none" or None: calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

### Returns

- If **multidim\_average** is set to global:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If **average**=None/'none', the shape will be (C,)

- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type**

The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_precision
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_precision(preds, target, num_labels=3)
tensor(0.5000)
>>> multilabel_precision(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.5000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_precision
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_precision(preds, target, num_labels=3)
tensor(0.5000)
>>> multilabel_precision(preds, target, num_labels=3, average=None)
tensor([1.0000, 0.0000, 0.5000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multilabel_precision
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> multilabel_precision(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0.3333, 0.0000])
>>> multilabel_precision(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.5000, 0.5000, 0.0000],
        [0.0000, 0.0000, 0.0000]])
```

## 1.42 Precision At Fixed Recall

### 1.42.1 Module Interface

#### BinaryPrecisionAtFixedRecall

```
class torchmetrics.classification.BinaryPrecisionAtFixedRecall(min_recall, thresholds=None,
                                                              ignore_index=None,
                                                              validate_args=True, **kwargs)
```

Compute the highest possible precision value given the minimum recall thresholds provided.

This is done by first calculating the precision-recall curve for different thresholds and then find the precision for a given recall level.

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A float tensor of shape  $(N, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** ([Tensor](#)): An int tensor of shape  $(N, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if `ignore_index` is specified). The value 1 always encodes the positive class.

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **precision** ([Tensor](#)): A scalar tensor with the maximum precision for the given recall level
- **threshold** ([Tensor](#)): A scalar tensor with the corresponding threshold level

---

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the `thresholds` argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the `thresholds` argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

---

### Parameters

- **min\_recall** ([float](#)) – float value specifying minimum recall threshold.
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d [Tensor](#) of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

## Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryPrecisionAtFixedRecall
>>> preds = tensor([0, 0.5, 0.7, 0.8])
>>> target = tensor([0, 1, 1, 0])
>>> metric = BinaryPrecisionAtFixedRecall(min_recall=0.5, thresholds=None)
>>> metric(preds, target)
(tensor(0.6667), tensor(0.5000))
>>> metric = BinaryPrecisionAtFixedRecall(min_recall=0.5, thresholds=5)
>>> metric(preds, target)
(tensor(0.6667), tensor(0.5000))
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

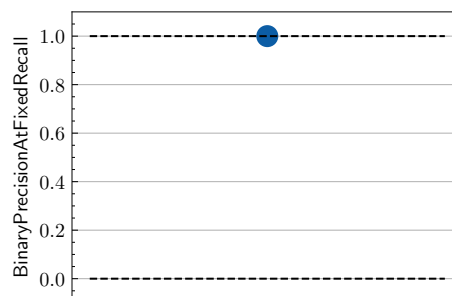
### Returns

Figure object and Axes object

### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

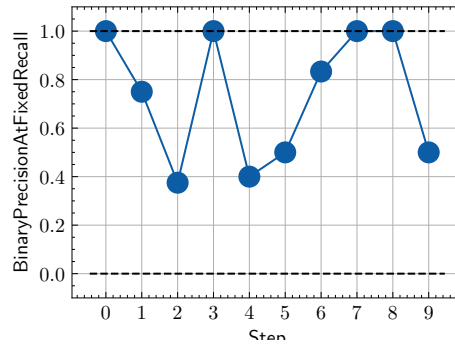
```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinaryPrecisionAtFixedRecall
>>> metric = BinaryPrecisionAtFixedRecall(min_recall=0.5)
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot() # the returned plot only shows the maximum_
↪ recall value by default
```



```

>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryPrecisionAtFixedRecall
>>> metric = BinaryPrecisionAtFixedRecall(min_recall=0.5)
>>> values = [ ]
>>> for _ in range(10):
...     # we index by 0 such that only the maximum recall value is plotted
...     values.append(metric(rand(10), randint(2,(10,)))[0])
>>> fig_, ax_ = metric.plot(values)

```



## MulticlassPrecisionAtFixedRecall

```

class torchmetrics.classification.MulticlassPrecisionAtFixedRecall(num_classes, min_recall,
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True,
                                                                    **kwargs)

```

Compute the highest possible precision value given the minimum recall thresholds provided.

This is done by first calculating the precision-recall curve for different thresholds and then find the precision for a given recall level.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** (**Tensor**): An int tensor of shape  $(N, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain values in the  $[0, n\_classes-1]$  range (except if *ignore\_index* is specified).

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to forward and compute the metric returns a tuple of either 2 tensors or 2 lists containing:

- **precision** (**Tensor**): A 1d tensor of size  $(n\_classes, )$  with the maximum precision for the given recall level per class
- **threshold** (**Tensor**): A 1d tensor of size  $(n\_classes, )$  with the corresponding threshold level per class

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{\text{samples}})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{\text{thresholds}} \times n_{\text{classes}})$  (constant memory).

### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **min\_recall** (`float`) – float value specifying minimum recall threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `Tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassPrecisionAtFixedRecall
>>> preds = tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                 [0.05, 0.75, 0.05, 0.05, 0.05],
...                 [0.05, 0.05, 0.75, 0.05, 0.05],
...                 [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = tensor([0, 1, 3, 2])
>>> metric = MulticlassPrecisionAtFixedRecall(num_classes=5, min_recall=0.5,
...     ↪ thresholds=None)
>>> metric(preds, target)
(tensor([1.0000, 1.0000, 0.2500, 0.2500, 0.0000]),
 tensor([7.5000e-01, 7.5000e-01, 5.0000e-02, 5.0000e-02, 1.0000e+06]))
>>> mcrafp = MulticlassPrecisionAtFixedRecall(num_classes=5, min_recall=0.5,
...     ↪ thresholds=5)
>>> mcrafp(preds, target)
(tensor([1.0000, 1.0000, 0.2500, 0.2500, 0.0000]),
 tensor([7.5000e-01, 7.5000e-01, 0.0000e+00, 0.0000e+00, 1.0000e+06]))
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

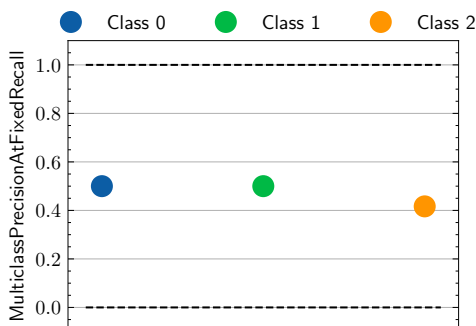
- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

Figure object and Axes object

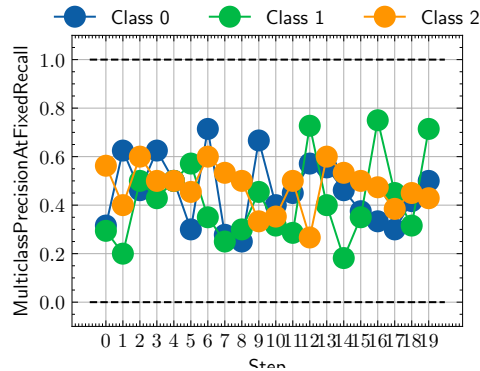
**Raises****`ModuleNotFoundError`** – If `matplotlib` is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value per class
>>> from torchmetrics.classification import MulticlassPrecisionAtFixedRecall
>>> metric = MulticlassPrecisionAtFixedRecall(num_classes=3, min_recall=0.5)
>>> metric.update(rand(20, 3).softmax(dim=-1), randint(3, (20,)))
>>> fig_, ax_ = metric.plot() # the returned plot only shows the maximum_
↪ recall value by default
```



```
>>> from torch import rand, randint
>>> # Example plotting a multiple values per class
>>> from torchmetrics.classification import MulticlassPrecisionAtFixedRecall
>>> metric = MulticlassPrecisionAtFixedRecall(num_classes=3, min_recall=0.5)
>>> values = []
>>> for _ in range(20):
...     # we index by 0 such that only the maximum recall value is plotted
...     values.append(metric(rand(20, 3).softmax(dim=-1), randint(3, (20,))))[0])
>>> fig_, ax_ = metric.plot(values)
```





## MultilabelPrecisionAtFixedRecall

```
class torchmetrics.classification.MultilabelPrecisionAtFixedRecall(num_labels, min_recall,
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True,
                                                                    **kwargs)
```

Compute the highest possible precision value given the minimum recall thresholds provided.

This is done by first calculating the precision-recall curve for different thresholds and then find the precision for a given recall level.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** ([Tensor](#)): An int tensor of shape  $(N, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to forward and compute the metric returns a tuple of either 2 tensors or 2 lists containing:

- **precision** ([Tensor](#)): A 1d tensor of size  $(n\_classes, )$  with the maximum precision for the given recall level per class
- **threshold** ([Tensor](#)): A 1d tensor of size  $(n\_classes, )$  with the corresponding threshold level per class

---

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

---

### Parameters

- **num\_labels** ([int](#)) – Integer specifying the number of labels

- **min\_recall** (`float`) – float value specifying minimum recall threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `Tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelPrecisionAtFixedRecall
>>> preds = tensor([[0.75, 0.05, 0.35],
...                 [0.45, 0.75, 0.05],
...                 [0.05, 0.55, 0.75],
...                 [0.05, 0.65, 0.05]])
>>> target = tensor([[1, 0, 1],
...                  [0, 0, 0],
...                  [0, 1, 1],
...                  [1, 1, 1]])
>>> metric = MultilabelPrecisionAtFixedRecall(num_labels=3, min_recall=0.5,
↳ thresholds=None)
>>> metric(preds, target)
(tensor([1.0000, 0.6667, 1.0000]), tensor([0.7500, 0.5500, 0.3500]))
>>> mlrafp = MultilabelPrecisionAtFixedRecall(num_labels=3, min_recall=0.5,
↳ thresholds=5)
>>> mlrafp(preds, target)
(tensor([1.0000, 0.6667, 1.0000]), tensor([0.7500, 0.5000, 0.2500]))
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

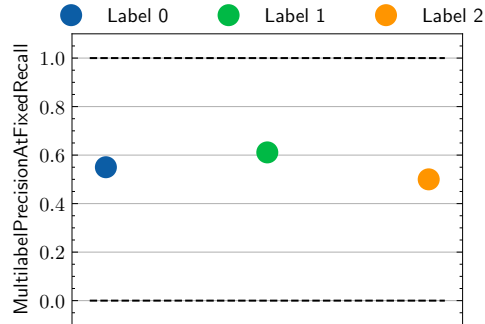
`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

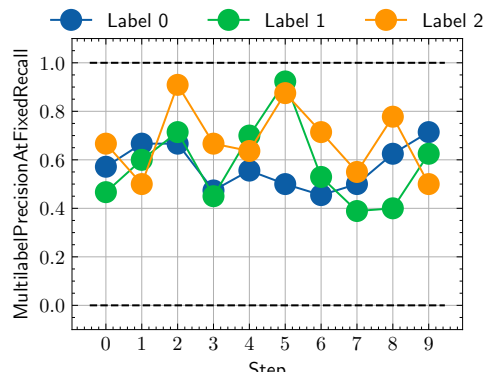
Figure object and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelPrecisionAtFixedRecall
>>> metric = MultilabelPrecisionAtFixedRecall(num_labels=3, min_recall=0.5)
>>> metric.update(rand(20, 3), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot() # the returned plot only shows the maximum_
↪ recall value by default
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import MultilabelPrecisionAtFixedRecall
>>> metric = MultilabelPrecisionAtFixedRecall(num_labels=3, min_recall=0.5)
>>> values = [ ]
>>> for _ in range(10):
...     # we index by 0 such that only the maximum recall value is plotted
...     values.append(metric(rand(20, 3), randint(2, (20, 3)))[0])
>>> fig_, ax_ = metric.plot(values)
```



## 1.42.2 Functional Interface

### binary\_precision\_at\_fixed\_recall

```
torchmetrics.functional.classification.binary_precision_at_fixed_recall(preds, target,
                                                                    min_recall,
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True)
```

Compute the highest possible precision value given the minimum recall thresholds provided for binary tasks.

This is done by first calculating the precision-recall curve for different thresholds and then find the precision for a given recall level.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

#### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **min\_recall** ([float](#)) – float value specifying minimum recall threshold.
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d [Tensor](#) of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** ([Optional\[int\]](#)) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

#### Returns

a tuple of 2 tensors containing:

- **precision**: an scalar tensor with the maximum precision for the given precision level
- **threshold**: an scalar tensor with the corresponding threshold level

**Return type**  
(tuple)

### Example

```
>>> from torchmetrics.functional.classification import binary_precision_at_fixed_
    recall
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_precision_at_fixed_recall(preds, target, min_recall=0.5, thresholds=None)
(tensor(0.6667), tensor(0.5000))
>>> binary_precision_at_fixed_recall(preds, target, min_recall=0.5, thresholds=5)
(tensor(0.6667), tensor(0.5000))
```

### multiclass\_precision\_at\_fixed\_recall

```
torchmetrics.functional.classification.multiclass_precision_at_fixed_recall(preds, target,
                                                                            num_classes,
                                                                            min_recall,
                                                                            thresh-
                                                                            olds=None,
                                                                            ig-
                                                                            nore_index=None,
                                                                            vali-
                                                                            date_args=True)
```

Compute the highest possible precision value given the minimum recall thresholds provided for multiclass tasks.

This is done by first calculating the precision-recall curve for different thresholds and then find the precision for a given recall level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels

- **num\_classes** (`int`) – Integer specifying the number of classes
- **min\_recall** (`float`) – float value specifying minimum recall threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `Tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

a tuple of either 2 tensors or 2 lists containing

- precision: an 1d tensor of size (`n_classes,`) with the maximum precision for the given recall level per class
- thresholds: an 1d tensor of size (`n_classes,`) with the corresponding threshold level per class

### Return type

(`tuple`)

### Example

```
>>> from torchmetrics.functional.classification import multiclass_precision_at_
↳fixed_recall
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> multiclass_precision_at_fixed_recall(
...     preds, target, num_classes=5, min_recall=0.5, thresholds=None)
(tensor([1.0000, 1.0000, 0.2500, 0.2500, 0.0000]),
 tensor([7.5000e-01, 7.5000e-01, 5.0000e-02, 5.0000e-02, 1.0000e+06]))
>>> multiclass_precision_at_fixed_recall(
...     preds, target, num_classes=5, min_recall=0.5, thresholds=5)
(tensor([1.0000, 1.0000, 0.2500, 0.2500, 0.0000]),
 tensor([7.5000e-01, 7.5000e-01, 0.0000e+00, 0.0000e+00, 1.0000e+06]))
```

## multilabel\_precision\_at\_fixed\_recall

```
torchmetrics.functional.classification.multilabel_precision_at_fixed_recall(preds, target,
                                                                              num_labels,
                                                                              min_recall,
                                                                              thresh-
                                                                              olds=None,
                                                                              ig-
                                                                              nore_index=None,
                                                                              vali-
                                                                              date_args=True)
```

Compute the highest possible precision value given the minimum recall thresholds provided for multilabel tasks.

This is done by first calculating the precision-recall curve for different thresholds and then find the precision for a given recall level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

### Parameters

- **preds** (*Tensor*) – Tensor with predictions
- **target** (*Tensor*) – Tensor with true labels
- **num\_labels** (*int*) – Integer specifying the number of labels
- **min\_recall** (*float*) – float value specifying minimum recall threshold.
- **thresholds** (*Union[int, List[float], Tensor, None]*) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *Tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (*Optional[int]*) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

**Returns**

a tuple of either 2 tensors or 2 lists containing

- precision: an 1d tensor of size (n\_classes, ) with the maximum precision for the given recall level per class
- thresholds: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

**Return type**

(tuple)

**Example**

```
>>> from torchmetrics.functional.classification import multilabel_precision_at_
    ↪ fixed_recall
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                        [0.45, 0.75, 0.05],
...                        [0.05, 0.55, 0.75],
...                        [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                         [0, 0, 0],
...                         [0, 1, 1],
...                         [1, 1, 1]])
>>> multilabel_precision_at_fixed_recall(preds, target, num_labels=3, min_recall=0.
    ↪ 5, thresholds=None)
(tensor([1.0000, 0.6667, 1.0000]), tensor([0.7500, 0.5500, 0.3500]))
>>> multilabel_precision_at_fixed_recall(preds, target, num_labels=3, min_recall=0.
    ↪ 5, thresholds=5)
(tensor([1.0000, 0.6667, 1.0000]), tensor([0.7500, 0.5000, 0.2500]))
```

## 1.43 Precision Recall Curve

### 1.43.1 Module Interface

```
class torchmetrics.PrecisionRecallCurve(task: typing_extensions.Literal[binary, multiclass, multilabel],
    thresholds: int | List[float] | Tensor | None = None, num_classes:
    int | None = None, num_labels: int | None = None, ignore_index:
    int | None = None, validate_args: bool = True, **kwargs: Any)
```

Compute the precision-recall curve.

The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either 'binary', 'multiclass' or multilabel. See the documentation of `BinaryPrecisionRecallCurve`, `MulticlassPrecisionRecallCurve` and `MultilabelPrecisionRecallCurve` for the specific details of each argument influence and examples.

**Legacy Example:**



```

>>> pred = torch.tensor([0, 0.1, 0.8, 0.4])
>>> target = torch.tensor([0, 1, 1, 0])
>>> pr_curve = PrecisionRecallCurve(task="binary")
>>> precision, recall, thresholds = pr_curve(pred, target)
>>> precision
tensor([0.5000, 0.6667, 0.5000, 1.0000, 1.0000])
>>> recall
tensor([1.0000, 1.0000, 0.5000, 0.5000, 0.0000])
>>> thresholds
tensor([0.0000, 0.1000, 0.4000, 0.8000])

>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> pr_curve = PrecisionRecallCurve(task="multiclass", num_classes=5)
>>> precision, recall, thresholds = pr_curve(pred, target)
>>> precision
[tensor([0.2500, 1.0000, 1.0000]), tensor([0.2500, 1.0000, 1.0000]), tensor([0.
↳ 2500, 0.0000, 1.0000]),
 tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.])]
>>> recall
[tensor([1., 1., 0.]), tensor([1., 1., 0.]), tensor([1., 0., 0.]), tensor([1.,
↳ 0., 0.]), tensor([nan, 0.])]
>>> thresholds
[tensor([0.0500, 0.7500]), tensor([0.0500, 0.7500]), tensor([0.0500, 0.7500]),
↳ tensor([0.0500, 0.7500]),
 tensor(0.0500)]

```

Initialize task metric.

## BinaryPrecisionRecallCurve

```

class torchmetrics.classification.BinaryPrecisionRecallCurve(thresholds=None,
                                                            ignore_index=None,
                                                            validate_args=True, **kwargs)

```

Compute the precision-recall curve for binary tasks.

The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): A float tensor of shape (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (**Tensor**): An int tensor of shape (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **precision** ([Tensor](#)): if *thresholds*=*None* a list for each class is returned with an 1d tensor of size  $(n\_thresholds+1, )$  with precision values (length may differ between classes). If *thresholds* is set to something else, then a single 2d tensor of size  $(n\_classes, n\_thresholds+1)$  with precision values is returned.
- **recall** ([Tensor](#)): if *thresholds*=*None* a list for each class is returned with an 1d tensor of size  $(n\_thresholds+1, )$  with recall values (length may differ between classes). If *thresholds* is set to something else, then a single 2d tensor of size  $(n\_classes, n\_thresholds+1)$  with recall values is returned.
- **thresholds** ([Tensor](#)): if *thresholds*=*None* a list for each class is returned with an 1d tensor of size  $(n\_thresholds, )$  with increasing threshold values (length may differ between classes). If *threshold* is set to something else, then a single 1d tensor of size  $(n\_thresholds, )$  is returned with shared threshold values for all classes.

---

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

---

### Parameters

- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torchmetrics.classification import BinaryPrecisionRecallCurve
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> bprc = BinaryPrecisionRecallCurve(thresholds=None)
>>> bprc(preds, target)
(tensor([0.5000, 0.6667, 0.5000, 0.0000, 1.0000]),
 tensor([1.0000, 1.0000, 0.5000, 0.0000, 0.0000]),
 tensor([0.0000, 0.5000, 0.7000, 0.8000]))
>>> bprc = BinaryPrecisionRecallCurve(thresholds=5)
>>> bprc(preds, target)
(tensor([0.5000, 0.6667, 0.6667, 0.0000, 0.0000, 1.0000]),
```

(continues on next page)

(continued from previous page)

```
tensor([1., 1., 1., 0., 0., 0.]),
tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))
```

**plot**(*curve=None, score=None, ax=None*)

Plot a single curve from the metric.

#### Parameters

- **curve** (`Optional[Tuple[Tensor, Tensor, Tensor]]`) – the output of either *metric.compute* or *metric.forward*. If no value is provided, will automatically call *metric.compute* and plot that result.
- **score** (`Union[Tensor, bool, None]`) – Provide a area-under-the-curve score to be displayed on the plot. If *True* and no curve is provided, will automatically compute the score.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

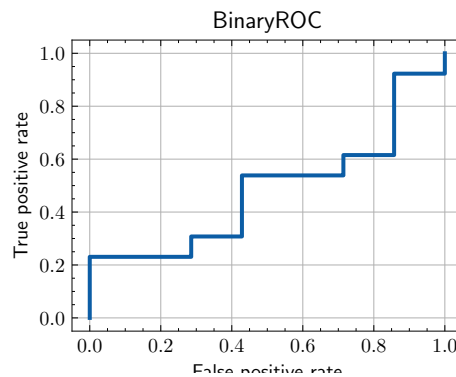
#### Returns

Figure and Axes object

#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randn, randint
>>> import torch.nn.functional as F
>>> from torchmetrics.classification import BinaryROC
>>> preds = F.softmax(randn(20, 2), dim=1)
>>> target = randint(2, (20,))
>>> metric = BinaryROC()
>>> metric.update(preds[:, 1], target)
>>> fig_, ax_ = metric.plot()
```



## MulticlassPrecisionRecallCurve

```
class torchmetrics.classification.MulticlassPrecisionRecallCurve(num_classes, thresholds=None,
                                                                ignore_index=None,
                                                                validate_args=True, **kwargs)
```

Compute the precision-recall curve for multiclass tasks.

The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** ([Tensor](#)): An int tensor of shape  $(N, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain values in the  $[0, n\_classes-1]$  range (except if `ignore_index` is specified).

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **precision** ([Tensor](#)): A 1d tensor of size  $(n\_thresholds+1, )$  with precision values
- **recall** ([Tensor](#)): A 1d tensor of size  $(n\_thresholds+1, )$  with recall values
- **thresholds** ([Tensor](#)): A 1d tensor of size  $(n\_thresholds, )$  with increasing threshold values

---

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the `thresholds` argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the `thresholds` argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

---

### Parameters

- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **thresholds** ([Union](#)[[int](#), [List](#)[[float](#)], [Tensor](#), `None`]) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to a 1D `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torchmetrics.classification import MulticlassPrecisionRecallCurve
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> mcprc = MulticlassPrecisionRecallCurve(num_classes=5, thresholds=None)
>>> precision, recall, thresholds = mcprc(preds, target)
>>> precision
(tensor([0.2500, 1.0000, 1.0000]), tensor([0.2500, 1.0000, 1.0000]), tensor([0.2500,
↪ 0.0000, 1.0000]),
tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.]])
>>> recall
(tensor([1., 1., 0.]), tensor([1., 1., 0.]), tensor([1., 0., 0.]), tensor([1., 0.,
↪ 0.]), tensor([nan, 0.]])
>>> thresholds
(tensor([0.0500, 0.7500]), tensor([0.0500, 0.7500]), tensor([0.0500, 0.7500]),
↪ tensor([0.0500, 0.7500]),
tensor(0.0500))
>>> mcprc = MulticlassPrecisionRecallCurve(num_classes=5, thresholds=5)
>>> mcprc(preds, target)
(tensor([[0.2500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000],
        [0.2500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000],
        [0.2500, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.2500, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
tensor([[1., 1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 0., 0.],
        [1., 0., 0., 0., 0., 0.],
        [1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0.]])
tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))
```

**plot**(*curve=None, score=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **curve** (`Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]], None`) – the output of either `metric.compute` or `metric.forward`. If no value is provided, will automatically call `metric.compute` and plot that result.
- **score** (`Union[Tensor, bool, None]`) – Provide a area-under-the-curve score to be displayed on the plot. If `True` and no curve is provided, will automatically compute the score.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure and Axes object

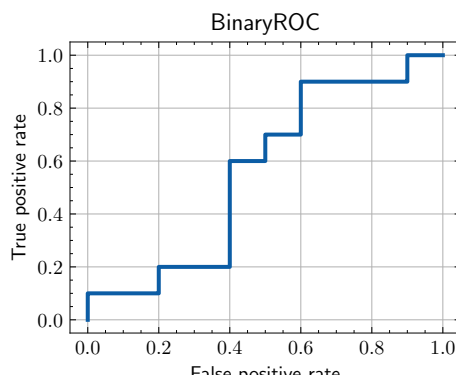
#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```

>>> from torch import randn, randint
>>> import torch.nn.functional as F
>>> from torchmetrics.classification import BinaryROC
>>> preds = F.softmax(randn(20, 2), dim=1)
>>> target = randint(2, (20,))
>>> metric = BinaryROC()
>>> metric.update(preds[:, 1], target)
>>> fig_, ax_ = metric.plot()

```



## MultilabelPrecisionRecallCurve

```

class torchmetrics.classification.MultilabelPrecisionRecallCurve(num_labels, thresholds=None,
                                                                ignore_index=None,
                                                                validate_args=True, **kwargs)

```

Compute the precision-recall curve for multilabel tasks.

The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** ([Tensor](#)): An int tensor of shape  $(N, C, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if `ignore_index` is specified).

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following a tuple of either 3 tensors or 3 lists containing:

- **precision** ([Tensor](#) or [List](#)): if `thresholds=None` a list for each label is returned with an 1d tensor of size  $(n\_thresholds+1, )$  with precision values (length may differ between labels). If `thresholds` is set to something else, then a single 2d tensor of size  $(n\_labels, n\_thresholds+1)$  with precision values is returned.
- **recall** ([Tensor](#) or [List](#)): if `thresholds=None` a list for each label is returned with an 1d tensor of size  $(n\_thresholds+1, )$  with recall values (length may differ between labels). If `thresholds` is set to something else, then a single 2d tensor of size  $(n\_labels, n\_thresholds+1)$  with recall values is returned.

- **thresholds** (`Tensor` or `List`): if *thresholds*=*None* a list for each label is returned with an 1d tensor of size (*n\_thresholds*, ) with increasing threshold values (length may differ between labels). If *threshold* is set to something else, then a single 1d tensor of size (*n\_thresholds*, ) is returned with shared threshold values for all labels.

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

### Parameters

- **preds** – Tensor with predictions
- **target** – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Example

```
>>> from torchmetrics.classification import MultilabelPrecisionRecallCurve
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> mlprc = MultilabelPrecisionRecallCurve(num_labels=3, thresholds=None)
>>> precision, recall, thresholds = mlprc(preds, target)
>>> precision
[tensor([0.5000, 0.5000, 1.0000, 1.0000]), tensor([0.5000, 0.6667, 0.5000, 0.0000,
↪1.0000]),
 tensor([0.7500, 1.0000, 1.0000, 1.0000])]
>>> recall
[tensor([1.0000, 0.5000, 0.5000, 0.0000]), tensor([1.0000, 1.0000, 0.5000, 0.0000,
```

(continues on next page)

(continued from previous page)

```

    ↪0.0000]),
    tensor([1.0000, 0.6667, 0.3333, 0.0000]))
>>> thresholds
[tensor([0.0500, 0.4500, 0.7500]), tensor([0.0500, 0.5500, 0.6500, 0.7500]), ↪
    ↪tensor([0.0500, 0.3500, 0.7500])]
>>> mlprc = MultilabelPrecisionRecallCurve(num_labels=3, thresholds=5)
>>> mlprc(preds, target)
(tensor([[0.5000, 0.5000, 1.0000, 1.0000, 0.0000, 1.0000],
        [0.5000, 0.6667, 0.6667, 0.0000, 0.0000, 1.0000],
        [0.7500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000]]),
 tensor([[1.0000, 0.5000, 0.5000, 0.5000, 0.0000, 0.0000],
        [1.0000, 1.0000, 1.0000, 0.0000, 0.0000, 0.0000],
        [1.0000, 0.6667, 0.3333, 0.3333, 0.0000, 0.0000]]),
 tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))

```

**plot**(*curve=None, score=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **curve** (`Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]], None`) – the output of either *metric.compute* or *metric.forward*. If no value is provided, will automatically call *metric.compute* and plot that result.
- **score** (`Union[Tensor, bool, None]`) – Provide a area-under-the-curve score to be displayed on the plot. If *True* and no curve is provided, will automatically compute the score.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure and Axes object

#### Raises

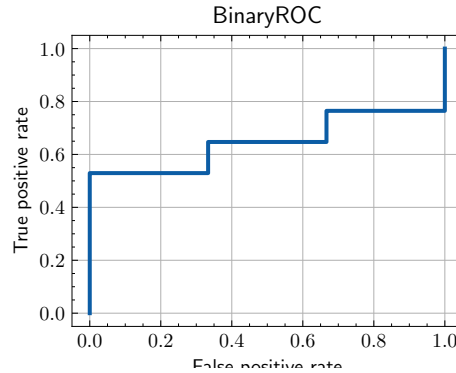
**ModuleNotFoundError** – If *matplotlib* is not installed

```

>>> from torch import randn, randint
>>> import torch.nn.functional as F
>>> from torchmetrics.classification import BinaryROC
>>> preds = F.softmax(randn(20, 2), dim=1)
>>> target = randint(2, (20,))
>>> metric = BinaryROC()
>>> metric.update(preds[:, 1], target)
>>> fig_, ax_ = metric.plot()

```





### 1.43.2 Functional Interface

`torchmetrics.functional.precision_recall_curve(preds, target, task, thresholds=None, num_classes=None, num_labels=None, ignore_index=None, validate_args=True)`

Compute the precision-recall curve.

The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either `'binary'`, `'multiclass'` or `'multilabel'`. See the documentation of `binary_precision_recall_curve()`, `multiclass_precision_recall_curve()` and `multilabel_precision_recall_curve()` for the specific details of each argument influence and examples.

#### Return type

`Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]`

#### Legacy Example:

```
>>> pred = torch.tensor([0, 0.1, 0.8, 0.4])
>>> target = torch.tensor([0, 1, 1, 0])
>>> precision, recall, thresholds = precision_recall_curve(pred, target, task=
↳ 'binary')
>>> precision
tensor([0.5000, 0.6667, 0.5000, 1.0000, 1.0000])
>>> recall
tensor([1.0000, 1.0000, 0.5000, 0.5000, 0.0000])
>>> thresholds
tensor([0.0000, 0.1000, 0.4000, 0.8000])
```

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                      [0.05, 0.75, 0.05, 0.05, 0.05],
...                      [0.05, 0.05, 0.75, 0.05, 0.05],
...                      [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> precision, recall, thresholds = precision_recall_curve(pred, target, task=
↳ 'multiclass', num_classes=5)
>>> precision
[tensor([0.2500, 1.0000, 1.0000]), tensor([0.2500, 1.0000, 1.0000]), tensor([0.
```

(continues on next page)

(continued from previous page)

```

↳2500, 0.0000, 1.0000]],
  tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.]))
>>> recall
[tensor([1., 1., 0.]), tensor([1., 1., 0.]), tensor([1., 0., 0.]), tensor([1.,
↳0., 0.]), tensor([nan, 0.])]
>>> thresholds
[tensor([0.0500, 0.7500]), tensor([0.0500, 0.7500]), tensor([0.0500, 0.7500]),
↳tensor([0.0500, 0.7500]),
  tensor([0.0500])]

```

## binary\_precision\_recall\_curve

`torchmetrics.functional.classification.binary_precision_recall_curve(preds, target, thresholds=None, ignore_index=None, validate_args=True)`

Compute the precision-recall curve for binary tasks.

The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.

- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

a tuple of 3 tensors containing:

- precision: an 1d tensor of size  $(n\_thresholds+1, )$  with precision values
- recall: an 1d tensor of size  $(n\_thresholds+1, )$  with recall values
- thresholds: an 1d tensor of size  $(n\_thresholds, )$  with increasing threshold values

### Return type

(tuple)

### Example

```
>>> from torchmetrics.functional.classification import binary_precision_recall_curve
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_precision_recall_curve(preds, target, thresholds=None)
(tensor([0.5000, 0.6667, 0.5000, 0.0000, 1.0000]),
 tensor([1.0000, 1.0000, 0.5000, 0.0000, 0.0000]),
 tensor([0.0000, 0.5000, 0.7000, 0.8000]))
>>> binary_precision_recall_curve(preds, target, thresholds=5)
(tensor([0.5000, 0.6667, 0.6667, 0.0000, 0.0000, 1.0000]),
 tensor([1., 1., 1., 0., 0., 0.]),
 tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))
```

## multiclass\_precision\_recall\_curve

```
torchmetrics.functional.classification.multiclass_precision_recall_curve(preds, target,
                                                                           num_classes,
                                                                           thresholds=None,
                                                                           ignore_index=None,
                                                                           validate_args=True)
```

Compute the precision-recall curve for multiclass tasks.

The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor):  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor):  $(N, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain values in the  $[0, n\_classes-1]$  range (except if `ignore_index` is specified).

Additional dimension  $\dots$  will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the `thresholds` argument to `None` will activate the

non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_classes** (`int`) – Integer specifying the number of classes
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Returns

a tuple of either 3 tensors or 3 lists containing

- **precision**: if *thresholds=None* a list for each class is returned with an 1d tensor of size (`n_thresholds+1`, ) with precision values (length may differ between classes). If *thresholds* is set to something else, then a single 2d tensor of size (`n_classes`, `n_thresholds+1`) with precision values is returned.
- **recall**: if *thresholds=None* a list for each class is returned with an 1d tensor of size (`n_thresholds+1`, ) with recall values (length may differ between classes). If *thresholds* is set to something else, then a single 2d tensor of size (`n_classes`, `n_thresholds+1`) with recall values is returned.
- **thresholds**: if *thresholds=None* a list for each class is returned with an 1d tensor of size (`n_thresholds`, ) with increasing threshold values (length may differ between classes). If *threshold* is set to something else, then a single 1d tensor of size (`n_thresholds`, ) is returned with shared threshold values for all classes.

### Return type

(`tuple`)

### Example

```
>>> from torchmetrics.functional.classification import multiclass_precision_recall_
↳ curve
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                        [0.05, 0.75, 0.05, 0.05, 0.05],
...                        [0.05, 0.05, 0.75, 0.05, 0.05],
...                        [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> precision, recall, thresholds = multiclass_precision_recall_curve(
...     preds, target, num_classes=5, thresholds=None
... )
>>> precision
[tensor([0.2500, 1.0000, 1.0000]), tensor([0.2500, 1.0000, 1.0000]), tensor([0.2500,
↳ 0.0000, 1.0000]),
 tensor([0.2500, 0.0000, 1.0000]), tensor([0., 1.])]
>>> recall
[tensor([1., 1., 0.]), tensor([1., 1., 0.]), tensor([1., 0., 0.]), tensor([1., 0.,
↳ 0.]), tensor([nan, 0.])]
>>> thresholds
[tensor([0.0500, 0.7500]), tensor([0.0500, 0.7500]), tensor([0.0500, 0.7500]),
↳ tensor([0.0500, 0.7500]),
 tensor([0.0500])]
>>> multiclass_precision_recall_curve(
...     preds, target, num_classes=5, thresholds=5
... )
(tensor([[0.2500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000],
        [0.2500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000],
        [0.2500, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.2500, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
 tensor([[1., 1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 0., 0.],
        [1., 0., 0., 0., 0., 0.],
        [1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0.]]),
 tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))
```

### multilabel\_precision\_recall\_curve

`torchmetrics.functional.classification.multilabel_precision_recall_curve(preds, target, num_labels, thresholds=None, ignore_index=None, validate_args=True)`

Compute the precision-recall curve for multilabel tasks.

The curve consist of multiple pairs of precision and recall values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply

sigmoid per element.

- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_labels** (int) – Integer specifying the number of labels
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

#### Returns

a tuple of either 3 tensors or 3 lists containing

- **precision**: if *thresholds=None* a list for each label is returned with an 1d tensor of size (n\_thresholds+1, ) with precision values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size (n\_labels, n\_thresholds+1) with precision values is returned.
- **recall**: if *thresholds=None* a list for each label is returned with an 1d tensor of size (n\_thresholds+1, ) with recall values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size (n\_labels, n\_thresholds+1) with recall values is returned.
- **thresholds**: if *thresholds=None* a list for each label is returned with an 1d tensor of size (n\_thresholds, ) with increasing threshold values (length may differ between labels). If *threshold* is set to something else, then a single 1d tensor of size (n\_thresholds, ) is returned with shared threshold values for all labels.

#### Return type

(tuple)

## Example

```

>>> from torchmetrics.functional.classification import multilabel_precision_recall_
    ↪ curve
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                        [0.45, 0.75, 0.05],
...                        [0.05, 0.55, 0.75],
...                        [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                         [0, 0, 0],
...                         [0, 1, 1],
...                         [1, 1, 1]])
>>> precision, recall, thresholds = multilabel_precision_recall_curve(
...     preds, target, num_labels=3, thresholds=None
... )
>>> precision
(tensor([0.5000, 0.5000, 1.0000, 1.0000]), tensor([0.5000, 0.6667, 0.5000, 0.0000, ↪
    ↪ 1.0000]),
 tensor([0.7500, 1.0000, 1.0000, 1.0000]))
>>> recall
(tensor([1.0000, 0.5000, 0.5000, 0.0000]), tensor([1.0000, 1.0000, 0.5000, 0.0000, ↪
    ↪ 0.0000]),
 tensor([1.0000, 0.6667, 0.3333, 0.0000]))
>>> thresholds
(tensor([0.0500, 0.4500, 0.7500]), tensor([0.0500, 0.5500, 0.6500, 0.7500]), ↪
    ↪ tensor([0.0500, 0.3500, 0.7500]))
>>> multilabel_precision_recall_curve(
...     preds, target, num_labels=3, thresholds=5
... )
(tensor([[0.5000, 0.5000, 1.0000, 1.0000, 0.0000, 1.0000],
        [0.5000, 0.6667, 0.6667, 0.0000, 0.0000, 1.0000],
        [0.7500, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000]]),
 tensor([[1.0000, 0.5000, 0.5000, 0.5000, 0.0000, 0.0000],
        [1.0000, 1.0000, 1.0000, 0.0000, 0.0000, 0.0000],
        [1.0000, 0.6667, 0.3333, 0.3333, 0.0000, 0.0000]]),
 tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000]))

```

## 1.44 Recall

### 1.44.1 Module Interface

**class** torchmetrics.Recall(task: typing\_extensions.Literal[binary, multiclass, multilabel], threshold: float = 0.5, num\_classes: int | None = None, num\_labels: int | None = None, average: typing\_extensions.Literal[micro, macro, weighted, none] | None = 'micro', multidim\_average: typing\_extensions.Literal[global, samplewise] | None = 'global', top\_k: int | None = 1, ignore\_index: int | None = None, validate\_args: bool = True, \*\*kwargs: ~typing.Any)

Compute *Recall*.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively. The metric is only proper defined when  $TP + FN \neq 0$ . If this case is encountered for any class/label, the metric for that class/label will be set to 0 and the overall metric may therefore be affected in turn.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of BinaryRecall, MulticlassRecall and MultilabelRecall for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> preds = tensor([2, 0, 2, 1])
>>> target = tensor([1, 1, 2, 0])
>>> recall = Recall(task="multiclass", average='macro', num_classes=3)
>>> recall(preds, target)
tensor(0.3333)
>>> recall = Recall(task="multiclass", average='micro', num_classes=3)
>>> recall(preds, target)
tensor(0.2500)
```

Initialize task metric.

### BinaryRecall

```
class torchmetrics.classification.BinaryRecall(threshold=0.5, multidim_average='global',
                                              ignore_index=None, validate_args=True, **kwargs)
```

Compute *Recall* for binary tasks.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively. The metric is only proper defined when  $TP + FN \neq 0$ . If this case is encountered a score of 0 is returned.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int tensor or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** ([Tensor](#)): An int tensor of shape (N, ...)

As output to forward and compute the metric returns the following output:

- **br** ([Tensor](#)): If **multidim\_average** is set to **global**, the metric returns a scalar value. If **multidim\_average** is set to **samplewise**, the metric returns (N,) vector consisting of a scalar value per sample.

#### Parameters

- **threshold** ([float](#)) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.



- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryRecall
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryRecall()
>>> metric(preds, target)
tensor(0.6667)
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryRecall
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryRecall()
>>> metric(preds, target)
tensor(0.6667)
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import BinaryRecall
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = BinaryRecall(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.6667, 0.0000])
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure object and Axes object

#### Raises

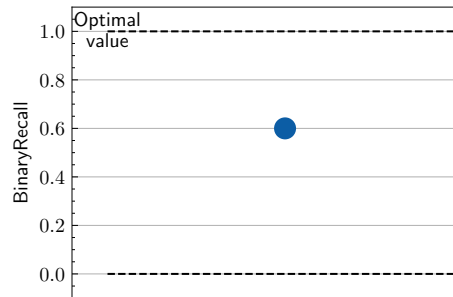
**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
```

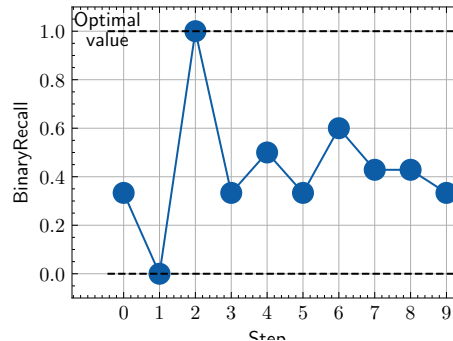
(continues on next page)

(continued from previous page)

```
>>> from torchmetrics.classification import BinaryRecall
>>> metric = BinaryRecall()
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryRecall
>>> metric = BinaryRecall()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassRecall

```
class torchmetrics.classification.MulticlassRecall(num_classes, top_k=1, average='macro',
                                                    multidim_average='global', ignore_index=None,
                                                    validate_args=True, **kwargs)
```

Compute *Recall* for multiclass tasks.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively. The metric is only

proper defined when  $TP + FN \neq 0$ . If this case is encountered for any class, the metric for that class will be set to 0 and the overall metric may therefore be affected in turn.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): An int tensor of shape (N, ...) or float tensor of shape (N, C, ...) If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (**Tensor**): An int tensor of shape (N, ...)

As output to forward and compute the metric returns the following output:

- **mcr** (**Tensor**): The returned shape depends on the `average` and `multidim_average` arguments:
  - If `multidim_average` is set to `global`:
    - \* If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
    - \* If `average=None/'none'`, the shape will be (C,)
  - If `multidim_average` is set to `samplewise`:
    - \* If `average='micro'/'macro'/'weighted'`, the shape will be (N,)
    - \* If `average=None/'none'`, the shape will be (N, C)

### Parameters

- **num\_classes** (**int**) – Integer specifying the number of classes
- **average** (**Optional**[**Literal**['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **top\_k** (**int**) – Number of highest probability or logit score predictions considered to find the correct label. Only works when `preds` contain probabilities/logits.
- **multidim\_average** (**Literal**['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (**Optional**[**int**]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (**bool**) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassRecall
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassRecall(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> mcr = MulticlassRecall(num_classes=3, average=None)
>>> mcr(preds, target)
tensor([0.5000, 1.0000, 1.0000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MulticlassRecall
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> metric = MulticlassRecall(num_classes=3)
>>> metric(preds, target)
tensor(0.8333)
>>> mcr = MulticlassRecall(num_classes=3, average=None)
>>> mcr(preds, target)
tensor([0.5000, 1.0000, 1.0000])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MulticlassRecall
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassRecall(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.5000, 0.2778])
>>> mcr = MulticlassRecall(num_classes=3, multidim_average='samplewise',
↪ average=None)
>>> mcr(preds, target)
tensor([[1.0000, 0.0000, 0.5000],
        [0.0000, 0.3333, 0.5000]])
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

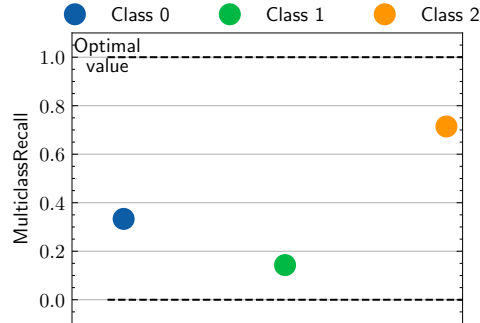
`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

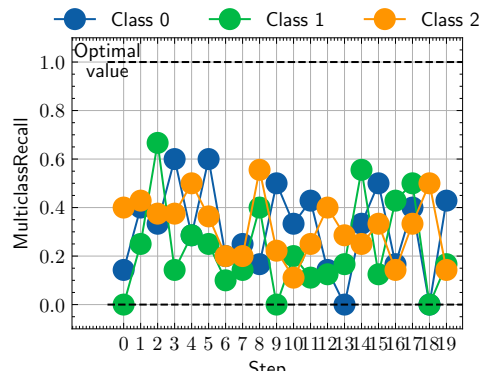
Figure object and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randint
>>> # Example plotting a single value per class
>>> from torchmetrics.classification import MulticlassRecall
>>> metric = MulticlassRecall(num_classes=3, average=None)
>>> metric.update(randint(3, (20,)), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randint
>>> # Example plotting a multiple values per class
>>> from torchmetrics.classification import MulticlassRecall
>>> metric = MulticlassRecall(num_classes=3, average=None)
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randint(3, (20,)), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelRecall

```
class torchmetrics.classification.MultilabelRecall(num_labels, threshold=0.5, average='macro',
                                                    multidim_average='global', ignore_index=None,
                                                    validate_args=True, **kwargs)
```

Compute *Recall* for multilabel tasks.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively. The metric is only proper defined when  $\text{TP} + \text{FN} \neq 0$ . If this case is encountered for any label, the metric for that label will be set to 0 and the overall metric may therefore be affected in turn.

As input to forward and update the metric accepts the following input:

- **preds** (*Tensor*): An int or float tensor of shape (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (*Tensor*): An int tensor of shape (N, C, ...)

As output to forward and compute the metric returns the following output:

- **mlr** (*Tensor*): The returned shape depends on the average and multidim\_average arguments:
  - If **multidim\_average** is set to **global**:
    - \* If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
    - \* If **average**=None/'none', the shape will be (C,)
  - If **multidim\_average** is set to **samplewise**:
    - \* If **average**='micro'/'macro'/'weighted', the shape will be (N,)
    - \* If **average**=None/'none', the shape will be (N, C)

### Parameters

- **num\_labels** (*int*) – Integer specifying the number of labels
- **threshold** (*float*) – Threshold for transforming probability to binary (0,1) predictions
- **average** (*Optional[Literal['micro', 'macro', 'weighted', 'none']]*) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **multidim\_average** (*Literal['global', 'samplewise']*) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (*Optional[int]*) – Specifies a target value that is ignored and does not contribute to the metric calculation

- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelRecall
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelRecall(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> mlr = MultilabelRecall(num_labels=3, average=None)
>>> mlr(preds, target)
tensor([1., 0., 1.])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelRecall
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelRecall(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> mlr = MultilabelRecall(num_labels=3, average=None)
>>> mlr(preds, target)
tensor([1., 0., 1.])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MultilabelRecall
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = MultilabelRecall(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.6667, 0.0000])
>>> mlr = MultilabelRecall(num_labels=3, multidim_average='samplewise',
↪ average=None)
>>> mlr(preds, target)
tensor([[1., 1., 0.],
        [0., 0., 0.]])
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

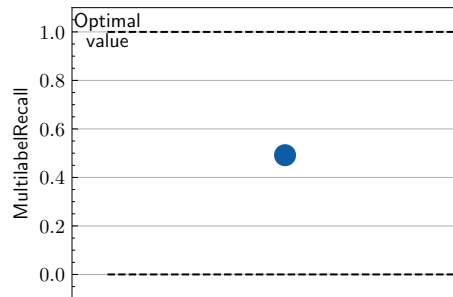
`Tuple[Figure, Union[Axes, ndarray]]`

**Returns**

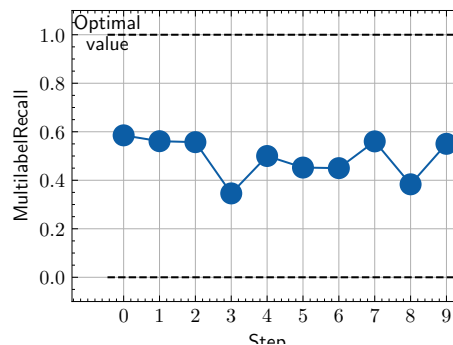
Figure object and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelRecall
>>> metric = MultilabelRecall(num_labels=3)
>>> metric.update(randint(2, (20, 3)), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import MultilabelRecall
>>> metric = MultilabelRecall(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2, (20, 3)), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```





### 1.44.2 Functional Interface

`torchmetrics.functional.recall(preds, target, task, threshold=0.5, num_classes=None, num_labels=None, average='micro', multidim_average='global', top_k=1, ignore_index=None, validate_args=True)`

Compute *Recall*. :rtype: `Tensor`

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either 'binary', 'multiclass' or 'multilabel'. See the documentation of `binary_recall()`, `multiclass_recall()` and `multilabel_recall()` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> preds = tensor([2, 0, 2, 1])
>>> target = tensor([1, 1, 2, 0])
>>> recall(preds, target, task="multiclass", average='macro', num_classes=3)
tensor(0.3333)
>>> recall(preds, target, task="multiclass", average='micro', num_classes=3)
tensor(0.2500)
```

### binary\_recall

`torchmetrics.functional.classification.binary_recall(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Compute *Recall* for binary tasks.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): (N, ...)

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension

- **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.

- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type**`Tensor`**Returns**

If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_recall
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> binary_recall(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_recall
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_recall(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import binary_recall
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> binary_recall(preds, target, multidim_average='samplewise')
tensor([0.6667, 0.0000])
```

**multiclass\_recall**

`torchmetrics.functional.classification.multiclass_recall(preds, target, num_classes, average='macro', top_k=1, multidim_average='global', ignore_index=None, validate_args=True)`

Compute *Recall* for multiclass tasks.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_classes** (`int`) – Integer specifying the number of classes
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **top\_k** (`int`) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be (C,)
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be (N,)
  - If `average=None/'none'`, the shape will be (N, C)

### Return type

The returned shape depends on the `average` and `multidim_average` arguments

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_recall
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
```

(continues on next page)

(continued from previous page)

```
>>> multiclass_recall(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_recall(preds, target, num_classes=3, average=None)
tensor([0.5000, 1.0000, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_recall
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                 [0.22, 0.61, 0.17],
...                 [0.71, 0.09, 0.20],
...                 [0.05, 0.82, 0.13]])
>>> multiclass_recall(preds, target, num_classes=3)
tensor(0.8333)
>>> multiclass_recall(preds, target, num_classes=3, average=None)
tensor([0.5000, 1.0000, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_recall
>>> target = tensor([[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]])
>>> preds = tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> multiclass_recall(preds, target, num_classes=3, multidim_average='samplewise
↳')
tensor([0.5000, 0.2778])
>>> multiclass_recall(preds, target, num_classes=3, multidim_average='samplewise
↳', average=None)
tensor([[1.0000, 0.0000, 0.5000],
        [0.0000, 0.3333, 0.5000]])
```

## multilabel\_recall

`torchmetrics.functional.classification.multilabel_recall(preds, target, num_labels, threshold=0.5, average='macro', multidim_average='global', ignore_index=None, validate_args=True)`

Compute *Recall* for multilabel tasks.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where TP and FN represent the number of true positives and false negatives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): (N, C, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions

- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **threshold** (`float`) – Threshold for transforming probability to binary (0,1) predictions
- **average** (`Optional[Literal['micro', 'macro', 'weighted', 'none']]`) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

#### Returns

- If `multidim_average` is set to `global`:
  - If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
  - If `average=None/'none'`, the shape will be `(C,)`
- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

#### Return type

The returned shape depends on the `average` and `multidim_average` arguments

#### Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_recall
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_recall(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_recall(preds, target, num_labels=3, average=None)
tensor([1., 0., 1.])
```

#### Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import multilabel_recall
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_recall(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_recall(preds, target, num_labels=3, average=None)
tensor([1., 0., 1.])
```

Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import multilabel_recall
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...               [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> multilabel_recall(preds, target, num_labels=3, multidim_average='samplewise
↳')
tensor([0.6667, 0.0000])
>>> multilabel_recall(preds, target, num_labels=3, multidim_average='samplewise
↳', average=None)
tensor([[1., 1., 0.],
        [0., 0., 0.]])
```

## 1.45 Recall At Fixed Precision

### 1.45.1 Module Interface

#### BinaryRecallAtFixedPrecision

```
class torchmetrics.classification.BinaryRecallAtFixedPrecision(min_precision, thresholds=None,
                                                                ignore_index=None,
                                                                validate_args=True, **kwargs)
```

Compute the highest possible recall value given the minimum precision thresholds provided.

This is done by first calculating the precision-recall curve for different thresholds and then find the recall for a given precision level.

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A float tensor of shape (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** ([Tensor](#)): An int tensor of shape (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns the following output:

- **recall** ([Tensor](#)): A scalar tensor with the maximum recall for the given precision level
- **threshold** ([Tensor](#)): A scalar tensor with the corresponding threshold level

---

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

---

### Parameters

- **min\_precision** (`float`) – float value specifying minimum precision threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `Tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryRecallAtFixedPrecision
>>> preds = tensor([0, 0.5, 0.7, 0.8])
>>> target = tensor([0, 1, 1, 0])
>>> metric = BinaryRecallAtFixedPrecision(min_precision=0.5, thresholds=None)
>>> metric(preds, target)
(tensor(1.), tensor(0.5000))
>>> metric = BinaryRecallAtFixedPrecision(min_precision=0.5, thresholds=5)
>>> metric(preds, target)
(tensor(1.), tensor(0.5000))
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

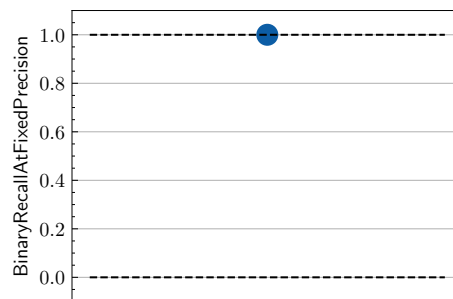
`Tuple[Figure, Union[Axes, ndarray]]`

**Returns**

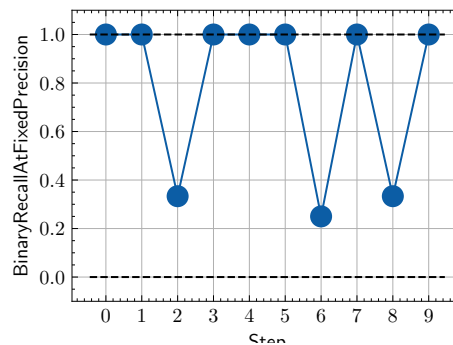
Figure object and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinaryRecallAtFixedPrecision
>>> metric = BinaryRecallAtFixedPrecision(min_precision=0.5)
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot() # the returned plot only shows the maximum_
↪ recall value by default
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinaryRecallAtFixedPrecision
>>> metric = BinaryRecallAtFixedPrecision(min_precision=0.5)
>>> values = [ ]
>>> for _ in range(10):
...     # we index by 0 such that only the maximum recall value is plotted
...     values.append(metric(rand(10), randint(2,(10,)))[0])
>>> fig_, ax_ = metric.plot(values)
```





## MulticlassRecallAtFixedPrecision

```
class torchmetrics.classification.MulticlassRecallAtFixedPrecision(num_classes, min_precision,
                                                                thresholds=None,
                                                                ignore_index=None,
                                                                validate_args=True,
                                                                **kwargs)
```

Compute the highest possible recall value given the minimum precision thresholds provided.

This is done by first calculating the precision-recall curve for different thresholds and then find the recall for a given precision level.

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** ([Tensor](#)): An int tensor of shape  $(N, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain values in the  $[0, n\_classes-1]$  range (except if `ignore_index` is specified).

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns a tuple of either 2 tensors or 2 lists containing:

- **recall** ([Tensor](#)): A 1d tensor of size  $(n\_classes, )$  with the maximum recall for the given precision level per class
- **threshold** ([Tensor](#)): A 1d tensor of size  $(n\_classes, )$  with the corresponding threshold level per class

---

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the `thresholds` argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the `thresholds` argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

---

### Parameters

- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **min\_precision** ([float](#)) – float value specifying minimum precision threshold.
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d [Tensor](#) of floats, will use the indicated thresholds in the tensor as bins for the calculation.

- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassRecallAtFixedPrecision
>>> preds = tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                 [0.05, 0.75, 0.05, 0.05, 0.05],
...                 [0.05, 0.05, 0.75, 0.05, 0.05],
...                 [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = tensor([0, 1, 3, 2])
>>> metric = MulticlassRecallAtFixedPrecision(num_classes=5, min_precision=0.5,
↳ thresholds=None)
>>> metric(preds, target)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 1.0000e+06, 1.
↳ 0000e+06, 1.0000e+06]))
>>> mcrafp = MulticlassRecallAtFixedPrecision(num_classes=5, min_precision=0.5,
↳ thresholds=5)
>>> mcrafp(preds, target)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 1.0000e+06, 1.
↳ 0000e+06, 1.0000e+06]))
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure object and Axes object

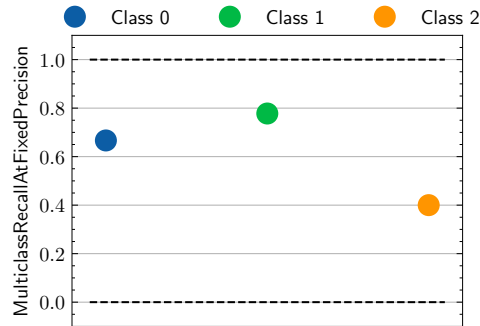
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value per class
>>> from torchmetrics.classification import MulticlassRecallAtFixedPrecision
>>> metric = MulticlassRecallAtFixedPrecision(num_classes=3, min_precision=0.5)
>>> metric.update(rand(20, 3).softmax(dim=-1), randint(3, (20,)))
>>> fig_, ax_ = metric.plot() # the returned plot only shows the maximum
↳ recall value by default
```

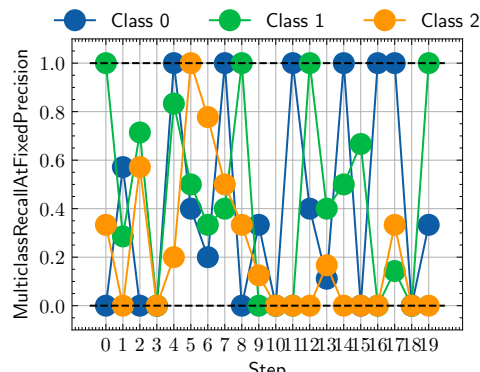
```
>>> from torch import rand, randint
>>> # Example plotting a multiple values per class
```

(continues on next page)



(continued from previous page)

```
>>> from torchmetrics.classification import MulticlassRecallAtFixedPrecision
>>> metric = MulticlassRecallAtFixedPrecision(num_classes=3, min_precision=0.5)
>>> values = []
>>> for _ in range(20):
...     # we index by 0 such that only the maximum recall value is plotted
...     values.append(metric(rand(20, 3).softmax(dim=-1), randint(3, (20,)))[0])
>>> fig_, ax_ = metric.plot(values)
```



## MultilabelRecallAtFixedPrecision

```
class torchmetrics.classification.MultilabelRecallAtFixedPrecision(num_labels, min_precision,
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True,
                                                                    **kwargs)
```

Compute the highest possible recall value given the minimum precision thresholds provided.

This is done by first calculating the precision-recall curve for different thresholds and the find the recall for a given precision level.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.

- **target** ([Tensor](#)): An int tensor of shape (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns a tuple of either 2 tensors or 2 lists containing:

- **recall** ([Tensor](#)): A 1d tensor of size (n\_classes, ) with the maximum recall for the given precision level per class
- **threshold** ([Tensor](#)): A 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

---

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

---

### Parameters

- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **min\_precision** ([float](#)) – float value specifying minimum precision threshold.
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d [Tensor](#) of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelRecallAtFixedPrecision
>>> preds = tensor([[0.75, 0.05, 0.35],
...                [0.45, 0.75, 0.05],
...                [0.05, 0.55, 0.75],
...                [0.05, 0.65, 0.05]])
>>> target = tensor([[1, 0, 1],
...                 [0, 0, 0],
```

(continues on next page)

(continued from previous page)

```

...             [0, 1, 1],
...             [1, 1, 1]])
>>> metric = MultilabelRecallAtFixedPrecision(num_labels=3, min_precision=0.5,
↳ thresholds=None)
>>> metric(preds, target)
(tensor([1., 1., 1.]), tensor([0.0500, 0.5500, 0.0500]))
>>> mlrafp = MultilabelRecallAtFixedPrecision(num_labels=3, min_precision=0.5,
↳ thresholds=5)
>>> mlrafp(preds, target)
(tensor([1., 1., 1.]), tensor([0.0000, 0.5000, 0.0000]))

```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure object and Axes object

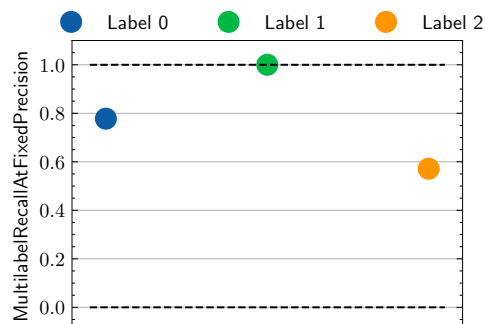
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```

>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelRecallAtFixedPrecision
>>> metric = MultilabelRecallAtFixedPrecision(num_labels=3, min_precision=0.5)
>>> metric.update(rand(20, 3), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot() # the returned plot only shows the maximum
↳ recall value by default

```



```

>>> from torch import rand, randint
>>> # Example plotting multiple values

```

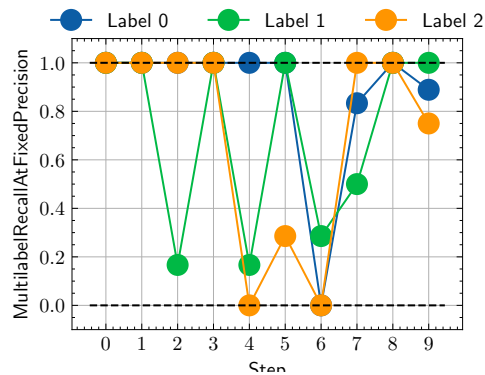
(continues on next page)

(continued from previous page)

```

>>> from torchmetrics.classification import MultilabelRecallAtFixedPrecision
>>> metric = MultilabelRecallAtFixedPrecision(num_labels=3, min_precision=0.5)
>>> values = [ ]
>>> for _ in range(10):
...     # we index by 0 such that only the maximum recall value is plotted
...     values.append(metric(rand(20, 3), randint(2, (20, 3)))[0])
>>> fig_, ax_ = metric.plot(values)

```



## 1.45.2 Functional Interface

### binary\_recall\_at\_fixed\_precision

`torchmetrics.functional.classification.binary_recall_at_fixed_precision(preds, target, min_precision, thresholds=None, ignore_index=None, validate_args=True)`

Compute the highest possible recall value given the minimum precision thresholds provided for binary tasks.

This is done by first calculating the precision-recall curve for different thresholds and the find the recall for a given precision level.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions

- **target** (`Tensor`) – Tensor with true labels
- **min\_precision** (`float`) – float value specifying minimum precision threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `Tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

a tuple of 2 tensors containing:

- recall: an scalar tensor with the maximum recall for the given precision level
- threshold: an scalar tensor with the corresponding threshold level

### Return type

(tuple)

### Example

```
>>> from torchmetrics.functional.classification import binary_recall_at_fixed_
    ↳ precision
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_recall_at_fixed_precision(preds, target, min_precision=0.5,
    ↳ thresholds=None)
(tensor(1.), tensor(0.5000))
>>> binary_recall_at_fixed_precision(preds, target, min_precision=0.5, thresholds=5)
(tensor(1.), tensor(0.5000))
```

### multiclass\_recall\_at\_fixed\_precision

```
torchmetrics.functional.classification.multiclass_recall_at_fixed_precision(preds, target,
                                                                              num_classes,
                                                                              min_precision,
                                                                              thresh-
                                                                              olds=None,
                                                                              ig-
                                                                              nore_index=None,
                                                                              vali-
                                                                              date_args=True)
```

Compute the highest possible recall value given the minimum precision thresholds provided for multiclass tasks.

This is done by first calculating the precision-recall curve for different thresholds and then find the recall for a given precision level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

#### Parameters

- **preds** (*Tensor*) – Tensor with predictions
- **target** (*Tensor*) – Tensor with true labels
- **num\_classes** (*int*) – Integer specifying the number of classes
- **min\_precision** (*float*) – float value specifying minimum precision threshold.
- **thresholds** (*Union[int, List[float], Tensor, None]*) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *Tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (*Optional[int]*) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

#### Returns

a tuple of either 2 tensors or 2 lists containing

- **recall**: an 1d tensor of size (n\_classes, ) with the maximum recall for the given precision level per class
- **thresholds**: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

#### Return type

(*tuple*)



### Example

```
>>> from torchmetrics.functional.classification import multiclass_recall_at_fixed_
    ↪ precision
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                        [0.05, 0.75, 0.05, 0.05, 0.05],
...                        [0.05, 0.05, 0.75, 0.05, 0.05],
...                        [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> multiclass_recall_at_fixed_precision(preds, target, num_classes=5, min_
    ↪ precision=0.5, thresholds=None)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 1.0000e+06, 1.
    ↪ 0000e+06, 1.0000e+06]))
>>> multiclass_recall_at_fixed_precision(preds, target, num_classes=5, min_
    ↪ precision=0.5, thresholds=5)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 1.0000e+06, 1.
    ↪ 0000e+06, 1.0000e+06]))
```

### multilabel\_recall\_at\_fixed\_precision

```
torchmetrics.functional.classification.multilabel_recall_at_fixed_precision(preds, target,
                                                                              num_labels,
                                                                              min_precision,
                                                                              thresh-
                                                                              olds=None,
                                                                              ig-
                                                                              nore_index=None,
                                                                              vali-
                                                                              date_args=True)
```

Compute the highest possible recall value given the minimum precision thresholds provided for multilabel tasks.

This is done by first calculating the precision-recall curve for different thresholds and then find the recall for a given precision level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels

- **num\_labels** (`int`) – Integer specifying the number of labels
- **min\_precision** (`float`) – float value specifying minimum precision threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `Tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

a tuple of either 2 tensors or 2 lists containing

- recall: an 1d tensor of size (n\_classes, ) with the maximum recall for the given precision level per class
- thresholds: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

### Return type

(tuple)

### Example

```
>>> from torchmetrics.functional.classification import multilabel_recall_at_fixed_
↳precision
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                        [0.45, 0.75, 0.05],
...                        [0.05, 0.55, 0.75],
...                        [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                         [0, 0, 0],
...                         [0, 1, 1],
...                         [1, 1, 1]])
>>> multilabel_recall_at_fixed_precision(preds, target, num_labels=3, min_
↳precision=0.5, thresholds=None)
(tensor([1., 1., 1.]), tensor([0.0500, 0.5500, 0.0500]))
>>> multilabel_recall_at_fixed_precision(preds, target, num_labels=3, min_
↳precision=0.5, thresholds=5)
(tensor([1., 1., 1.]), tensor([0.0000, 0.5000, 0.0000]))
```

## 1.46 ROC

### 1.46.1 Module Interface

```
class torchmetrics.ROC(task: typing_extensions.Literal[binary, multiclass, multilabel], thresholds: int |
    List[float] | Tensor | None = None, num_classes: int | None = None, num_labels: int |
    None = None, ignore_index: int | None = None, validate_args: bool = True, **kwargs:
    Any)
```

Compute the Receiver Operating Characteristic (ROC).

The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of BinaryROC, MulticlassROC and MultilabelROC for the specific details of each argument influence and examples.

**Legacy Example:**

```
>>> from torch import tensor
>>> pred = tensor([0.0, 1.0, 2.0, 3.0])
>>> target = tensor([0, 1, 1, 1])
>>> roc = ROC(task="binary")
>>> fpr, tpr, thresholds = roc(pred, target)
>>> fpr
tensor([0., 0., 0., 0., 1.])
>>> tpr
tensor([0.0000, 0.3333, 0.6667, 1.0000, 1.0000])
>>> thresholds
tensor([1.0000, 0.9526, 0.8808, 0.7311, 0.5000])
```

```
>>> pred = tensor([[0.75, 0.05, 0.05, 0.05],
...               [0.05, 0.75, 0.05, 0.05],
...               [0.05, 0.05, 0.75, 0.05],
...               [0.05, 0.05, 0.05, 0.75]])
>>> target = tensor([0, 1, 3, 2])
>>> roc = ROC(task="multiclass", num_classes=4)
>>> fpr, tpr, thresholds = roc(pred, target)
>>> fpr
[tensor([0., 0., 1.]), tensor([0., 0., 1.]), tensor([0.0000, 0.3333, 1.0000]),
 tensor([0.0000, 0.3333, 1.0000])]
>>> tpr
[tensor([0., 1., 1.]), tensor([0., 1., 1.]), tensor([0., 0., 1.]), tensor([0.,
 0., 1.])]
>>> thresholds
[tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500])]
```

```
>>> pred = tensor([[0.8191, 0.3680, 0.1138],
...               [0.3584, 0.7576, 0.1183],
...               [0.2286, 0.3468, 0.1338],
```

(continues on next page)

(continued from previous page)

```

...          [0.8603, 0.0745, 0.1837]])
>>> target = tensor([[1, 1, 0], [0, 1, 0], [0, 0, 0], [0, 1, 1]])
>>> roc = ROC(task='multilabel', num_labels=3)
>>> fpr, tpr, thresholds = roc(pred, target)
>>> fpr
[tensor([0.0000, 0.3333, 0.3333, 0.6667, 1.0000]),
 tensor([0., 0., 0., 1., 1.]),
 tensor([0.0000, 0.0000, 0.3333, 0.6667, 1.0000])]
>>> tpr
[tensor([0., 0., 1., 1., 1.]),
 tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000]),
 tensor([0., 1., 1., 1., 1.])]
>>> thresholds
[tensor([1.0000, 0.8603, 0.8191, 0.3584, 0.2286]),
 tensor([1.0000, 0.7576, 0.3680, 0.3468, 0.0745]),
 tensor([1.0000, 0.1837, 0.1338, 0.1183, 0.1138])]

```

Initialize task metric.

## BinaryROC

**class** torchmetrics.classification.**BinaryROC**(*thresholds=None, ignore\_index=None, validate\_args=True, \*\*kwargs*)

Compute the Receiver Operating Characteristic (ROC) for binary tasks.

The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (**Tensor**): A float tensor of shape (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (**Tensor**): An int tensor of shape (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns a tuple of 3 tensors containing:

- **fpr** (**Tensor**): A 1d tensor of size (n\_thresholds+1, ) with false positive rate values
- **tpr** (**Tensor**): A 1d tensor of size (n\_thresholds+1, ) with true positive rate values
- **thresholds** (**Tensor**): A 1d tensor of size (n\_thresholds, ) with decreasing threshold values

---

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

---

---

**Note:** The outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotome increasing.

---

### Parameters

- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryROC
>>> preds = tensor([0, 0.5, 0.7, 0.8])
>>> target = tensor([0, 1, 1, 0])
>>> metric = BinaryROC(thresholds=None)
>>> metric(preds, target)
(tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0.0000, 0.0000, 0.5000, 1.0000, 1.0000]),
 tensor([1.0000, 0.8000, 0.7000, 0.5000, 0.0000]))
>>> broc = BinaryROC(thresholds=5)
>>> broc(preds, target)
(tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0., 0., 1., 1., 1.]),
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))
```

**plot**(*curve=None, score=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **curve** (`Optional[Tuple[Tensor, Tensor, Tensor]]`) – the output of either `metric.compute` or `metric.forward`. If no value is provided, will automatically call `metric.compute` and plot that result.
- **score** (`Union[Tensor, bool, None]`) – Provide a area-under-the-curve score to be displayed on the plot. If `True` and no curve is provided, will automatically compute the score.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

**Returns**

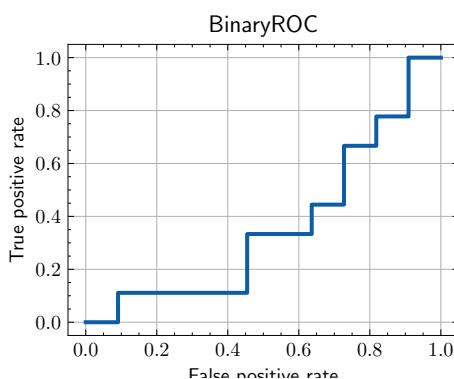
Figure and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```

>>> from torch import randn, randint
>>> import torch.nn.functional as F
>>> from torchmetrics.classification import BinaryROC
>>> preds = F.softmax(randn(20, 2), dim=1)
>>> target = randint(2, (20,))
>>> metric = BinaryROC()
>>> metric.update(preds[:, 1], target)
>>> fig_, ax_ = metric.plot()

```

**MulticlassROC**

**class** torchmetrics.classification.**MulticlassROC**(*num\_classes*, *thresholds=None*, *ignore\_index=None*, *validate\_args=True*, *\*\*kwargs*)

Compute the Receiver Operating Characteristic (ROC) for binary tasks.

The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): A float tensor of shape (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (**Tensor**): An int tensor of shape (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

---

**Note:** Additional dimension ... will be flattened into the batch dimension.

---

As output to forward and compute the metric returns a tuple of either 3 tensors or 3 lists containing

- **fpr** (**Tensor**): if *thresholds=None* a list for each class is returned with an 1d tensor of size (n\_thresholds+1, ) with false positive rate values (length may differ between classes). If *thresholds* is set to something else, then a single 2d tensor of size (n\_classes, n\_thresholds+1) with false positive rate values is returned.

- **tpr** ([Tensor](#)): if *thresholds=None* a list for each class is returned with an 1d tensor of size  $(n\_thresholds+1, )$  with true positive rate values (length may differ between classes). If *thresholds* is set to something else, then a single 2d tensor of size  $(n\_classes, n\_thresholds+1)$  with true positive rate values is returned.
- **thresholds** ([Tensor](#)): if *thresholds=None* a list for each class is returned with an 1d tensor of size  $(n\_thresholds, )$  with decreasing threshold values (length may differ between classes). If *threshold* is set to something else, then a single 1d tensor of size  $(n\_thresholds, )$  is returned with shared threshold values for all classes.

---

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

---



---

**Note:** Note that outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotone increasing.

---

### Parameters

- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassROC
>>> preds = tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                [0.05, 0.75, 0.05, 0.05, 0.05],
...                [0.05, 0.05, 0.75, 0.05, 0.05],
...                [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = tensor([0, 1, 3, 2])
>>> metric = MulticlassROC(num_classes=5, thresholds=None)
>>> fpr, tpr, thresholds = metric(preds, target)
>>> fpr
```

(continues on next page)

(continued from previous page)

```
[tensor([0., 0., 1.]), tensor([0., 0., 1.]), tensor([0.0000, 0.3333, 1.0000]),
 tensor([0.0000, 0.3333, 1.0000]), tensor([0., 1.])]
>>> tpr
[tensor([0., 1., 1.]), tensor([0., 1., 1.]), tensor([0., 0., 1.]), tensor([0., 0., 1.]),
 tensor([0., 0.]), tensor([0., 0.])]
>>> thresholds
[tensor([1.0000, 0.7500, 0.0500]), tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500]), tensor([1.0000, 0.7500, 0.0500]), tensor([1.0000,
 0.0500])]
>>> mcroc = MulticlassROC(num_classes=5, thresholds=5)
>>> mcroc(preds, target)
(tensor([[0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
 [0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
 [0.0000, 0.3333, 0.3333, 0.3333, 1.0000],
 [0.0000, 0.3333, 0.3333, 0.3333, 1.0000],
 [0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
 tensor([[0., 1., 1., 1., 1.],
 [0., 1., 1., 1., 1.],
 [0., 0., 0., 0., 1.],
 [0., 0., 0., 0., 1.],
 [0., 0., 0., 0., 0.]]),
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))
```

**plot**(*curve=None, score=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **curve** (`Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]], None`) – the output of either `metric.compute` or `metric.forward`. If no value is provided, will automatically call `metric.compute` and plot that result.
- **score** (`Union[Tensor, bool, None]`) – Provide a area-under-the-curve score to be displayed on the plot. If `True` and no curve is provided, will automatically compute the score.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

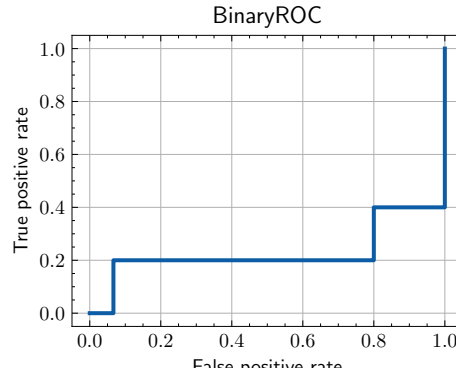
Figure and Axes object

#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> from torch import randn, randint
>>> import torch.nn.functional as F
>>> from torchmetrics.classification import BinaryROC
>>> preds = F.softmax(randn(20, 2), dim=1)
>>> target = randint(2, (20,))
>>> metric = BinaryROC()
>>> metric.update(preds[:, 1], target)
>>> fig_, ax_ = metric.plot()
```





## MultilabelROC

**class** torchmetrics.classification.**MultilabelROC**(*num\_labels*, *thresholds=None*, *ignore\_index=None*, *validate\_args=True*, *\*\*kwargs*)

Compute the Receiver Operating Characteristic (ROC) for binary tasks.

The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (**Tensor**): A float tensor of shape  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (**Tensor**): An int tensor of shape  $(N, C, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain  $\{0,1\}$  values (except if *ignore\_index* is specified).

---

**Note:** Additional dimension  $\dots$  will be flattened into the batch dimension.

---

As output to `forward` and `compute` the metric returns a tuple of either 3 tensors or 3 lists containing

- **fpr** (**Tensor**): if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n\_thresholds+1, )$  with false positive rate values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size  $(n\_labels, n\_thresholds+1)$  with false positive rate values is returned.
- **tpr** (**Tensor**): if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n\_thresholds+1, )$  with true positive rate values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size  $(n\_labels, n\_thresholds+1)$  with true positive rate values is returned.
- **thresholds** (**Tensor**): if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n\_thresholds, )$  with decreasing threshold values (length may differ between labels). If *threshold* is set to something else, then a single 1d tensor of size  $(n\_thresholds, )$  is returned with shared threshold values for all labels.

---

**Note:** The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$

(constant memory).

---

**Note:** The outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotome increasing.

---

### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelROC
>>> preds = tensor([[0.75, 0.05, 0.35],
...                 [0.45, 0.75, 0.05],
...                 [0.05, 0.55, 0.75],
...                 [0.05, 0.65, 0.05]])
>>> target = tensor([[1, 0, 1],
...                  [0, 0, 0],
...                  [0, 1, 1],
...                  [1, 1, 1]])
>>> metric = MultilabelROC(num_labels=3, thresholds=None)
>>> fpr, tpr, thresholds = metric(preds, target)
>>> fpr
[tensor([0.0000, 0.0000, 0.5000, 1.0000]),
 tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0., 0., 0., 1.])]
>>> tpr
[tensor([0.0000, 0.5000, 0.5000, 1.0000]),
 tensor([0.0000, 0.0000, 0.5000, 1.0000, 1.0000]),
 tensor([0.0000, 0.3333, 0.6667, 1.0000])]
>>> thresholds
[tensor([1.0000, 0.7500, 0.4500, 0.0500]),
 tensor([1.0000, 0.7500, 0.6500, 0.5500, 0.0500]),
 tensor([1.0000, 0.7500, 0.3500, 0.0500])]
```

(continues on next page)

(continued from previous page)

```
>>> mloc = MultilabelROC(num_labels=3, thresholds=5)
>>> mloc(preds, target)
(tensor([[0.0000, 0.0000, 0.0000, 0.5000, 1.0000],
        [0.0000, 0.5000, 0.5000, 0.5000, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
 tensor([[0.0000, 0.5000, 0.5000, 0.5000, 1.0000],
        [0.0000, 0.0000, 1.0000, 1.0000, 1.0000],
        [0.0000, 0.3333, 0.3333, 0.6667, 1.0000]]),
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))
```

**plot**(*curve=None, score=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **curve** (`Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]], None`) – the output of either `metric.compute` or `metric.forward`. If no value is provided, will automatically call `metric.compute` and plot that result.
- **score** (`Union[Tensor, bool, None]`) – Provide a area-under-the-curve score to be displayed on the plot. If `True` and no curve is provided, will automatically compute the score.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

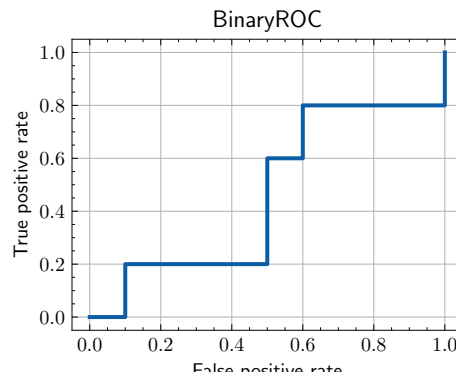
#### Returns

Figure and Axes object

#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> from torch import randn, randint
>>> import torch.nn.functional as F
>>> from torchmetrics.classification import BinaryROC
>>> preds = F.softmax(randn(20, 2), dim=1)
>>> target = randint(2, (20,))
>>> metric = BinaryROC()
>>> metric.update(preds[:, 1], target)
>>> fig_, ax_ = metric.plot()
```



## 1.46.2 Functional Interface

`torchmetrics.functional.roc(preds, target, task, thresholds=None, num_classes=None, num_labels=None, ignore_index=None, validate_args=True)`

Compute the Receiver Operating Characteristic (ROC).

The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either `'binary'`, `'multiclass'` or `'multilabel'`. See the documentation of `binary_roc()`, `multiclass_roc()` and `multilabel_roc()` for the specific details of each argument influence and examples.

### Return type

`Union[Tuple[Tensor, Tensor, Tensor], Tuple[List[Tensor], List[Tensor], List[Tensor]]]`

### Legacy Example:

```
>>> pred = torch.tensor([0.0, 1.0, 2.0, 3.0])
>>> target = torch.tensor([0, 1, 1, 1])
>>> fpr, tpr, thresholds = roc(pred, target, task='binary')
>>> fpr
tensor([0., 0., 0., 0., 1.])
>>> tpr
tensor([0.0000, 0.3333, 0.6667, 1.0000, 1.0000])
>>> thresholds
tensor([1.0000, 0.9526, 0.8808, 0.7311, 0.5000])
```

```
>>> pred = torch.tensor([[0.75, 0.05, 0.05, 0.05],
...                       [0.05, 0.75, 0.05, 0.05],
...                       [0.05, 0.05, 0.75, 0.05],
...                       [0.05, 0.05, 0.05, 0.75]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> fpr, tpr, thresholds = roc(pred, target, task='multiclass', num_classes=4)
>>> fpr
[tensor([0., 0., 1.]), tensor([0., 0., 1.]), tensor([0.0000, 0.3333, 1.0000]),
 tensor([0.0000, 0.3333, 1.0000])]
>>> tpr
[tensor([0., 1., 1.]), tensor([0., 1., 1.]), tensor([0., 0., 1.]), tensor([0.,
 0., 1.])]
>>> thresholds
[tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500])]
```

```
>>> pred = torch.tensor([[0.8191, 0.3680, 0.1138],
...                       [0.3584, 0.7576, 0.1183],
...                       [0.2286, 0.3468, 0.1338],
...                       [0.8603, 0.0745, 0.1837]])
>>> target = torch.tensor([[1, 1, 0], [0, 1, 0], [0, 0, 0], [0, 1, 1]])
>>> fpr, tpr, thresholds = roc(pred, target, task='multilabel', num_labels=3)
>>> fpr
```

(continues on next page)

(continued from previous page)

```
[tensor([0.0000, 0.3333, 0.3333, 0.6667, 1.0000]),
 tensor([0., 0., 0., 1., 1.]),
 tensor([0.0000, 0.0000, 0.3333, 0.6667, 1.0000])]
>>> tpr
[tensor([0., 0., 1., 1., 1.]), tensor([0.0000, 0.3333, 0.6667, 0.6667, 1.0000]),
 ↪ tensor([0., 1., 1., 1., 1.])]
>>> thresholds
[tensor([1.0000, 0.8603, 0.8191, 0.3584, 0.2286]),
 tensor([1.0000, 0.7576, 0.3680, 0.3468, 0.0745]),
 tensor([1.0000, 0.1837, 0.1338, 0.1183, 0.1138])]
```

## binary\_roc

`torchmetrics.functional.classification.binary_roc(preds, target, thresholds=None, ignore_index=None, validate_args=True)`

Compute the Receiver Operating Characteristic (ROC) for binary tasks.

The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified). The value 1 always encodes the positive class.

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

Note that outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotome increasing.

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.

- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

a tuple of 3 tensors containing:

- fpr: an 1d tensor of size  $(n\_thresholds+1, )$  with false positive rate values
- tpr: an 1d tensor of size  $(n\_thresholds+1, )$  with true positive rate values
- thresholds: an 1d tensor of size  $(n\_thresholds, )$  with decreasing threshold values

### Return type

(tuple)

### Example

```
>>> from torchmetrics.functional.classification import binary_roc
>>> preds = torch.tensor([0, 0.5, 0.7, 0.8])
>>> target = torch.tensor([0, 1, 1, 0])
>>> binary_roc(preds, target, thresholds=None)
(tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0.0000, 0.0000, 0.5000, 1.0000, 1.0000]),
 tensor([1.0000, 0.8000, 0.7000, 0.5000, 0.0000]))
>>> binary_roc(preds, target, thresholds=5)
(tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0., 0., 1., 1., 1.]),
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))
```

## multiclass\_roc

`torchmetrics.functional.classification.multiclass_roc(preds, target, num_classes, thresholds=None, ignore_index=None, validate_args=True)`

Compute the Receiver Operating Characteristic (ROC) for multiclass tasks.

The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor):  $(N, C, \dots)$ . Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside  $[0,1]$  range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor):  $(N, \dots)$ . Target should be a tensor containing ground truth labels, and therefore only contain values in the  $[0, n\_classes-1]$  range (except if `ignore_index` is specified).

Additional dimension  $\dots$  will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the `thresholds` argument to `None` will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the `thresholds` argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

Note that outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotone increasing.

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_classes** (`int`) – Integer specifying the number of classes
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to `None`, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an `int` (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an `list` of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d `tensor` of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Returns

a tuple of either 3 tensors or 3 lists containing

- fpr: if `thresholds=None` a list for each class is returned with an 1d tensor of size `(n_thresholds+1, )` with false positive rate values (length may differ between classes). If `thresholds` is set to something else, then a single 2d tensor of size `(n_classes, n_thresholds+1)` with false positive rate values is returned.
- tpr: if `thresholds=None` a list for each class is returned with an 1d tensor of size `(n_thresholds+1, )` with true positive rate values (length may differ between classes). If `thresholds` is set to something else, then a single 2d tensor of size `(n_classes, n_thresholds+1)` with true positive rate values is returned.
- thresholds: if `thresholds=None` a list for each class is returned with an 1d tensor of size `(n_thresholds, )` with decreasing threshold values (length may differ between classes). If `threshold` is set to something else, then a single 1d tensor of size `(n_thresholds, )` is returned with shared threshold values for all classes.

### Return type

(`tuple`)

### Example

```

>>> from torchmetrics.functional.classification import multiclass_roc
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                        [0.05, 0.75, 0.05, 0.05, 0.05],
...                        [0.05, 0.05, 0.75, 0.05, 0.05],
...                        [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> fpr, tpr, thresholds = multiclass_roc(
...     preds, target, num_classes=5, thresholds=None
... )
>>> fpr
[tensor([0., 0., 1.]), tensor([0., 0., 1.]), tensor([0.0000, 0.3333, 1.0000]),
 tensor([0.0000, 0.3333, 1.0000]), tensor([0., 1.])]
>>> tpr
[tensor([0., 1., 1.]), tensor([0., 1., 1.]), tensor([0., 0., 1.]), tensor([0., 0., 1.]),
 tensor([0., 0., 1.]), tensor([0., 0.])]
>>> thresholds
[tensor([1.0000, 0.7500, 0.0500]), tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500]), tensor([1.0000, 0.7500, 0.0500]),
 tensor([1.0000, 0.7500, 0.0500])]
>>> multiclass_roc(
...     preds, target, num_classes=5, thresholds=5
... )
(tensor([[0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 1.0000],
        [0.0000, 0.3333, 0.3333, 0.3333, 1.0000],
        [0.0000, 0.3333, 0.3333, 0.3333, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
 tensor([[0., 1., 1., 1., 1.],
        [0., 1., 1., 1., 1.],
        [0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0.]]),
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))

```

### multilabel\_roc

`torchmetrics.functional.classification.multilabel_roc(preds, target, num_labels, thresholds=None, ignore_index=None, validate_args=True)`

Compute the Receiver Operating Characteristic (ROC) for multilabel tasks.

The curve consist of multiple pairs of true positive rate (TPR) and false positive rate (FPR) values evaluated at different thresholds, such that the tradeoff between the two values can be seen.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.



The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

Note that outputted thresholds will be in reversed order to ensure that they corresponds to both fpr and tpr which are sorted in reversed order during their calculation, such that they are monotome increasing.

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **num\_labels** (`int`) – Integer specifying the number of labels
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

#### Returns

a tuple of either 3 tensors or 3 lists containing

- **fpr**: if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n_{thresholds}+1, )$  with false positive rate values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size  $(n_{labels}, n_{thresholds}+1)$  with false positive rate values is returned.
- **tpr**: if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n_{thresholds}+1, )$  with true positive rate values (length may differ between labels). If *thresholds* is set to something else, then a single 2d tensor of size  $(n_{labels}, n_{thresholds}+1)$  with true positive rate values is returned.
- **thresholds**: if *thresholds=None* a list for each label is returned with an 1d tensor of size  $(n_{thresholds}, )$  with decreasing threshold values (length may differ between labels). If *threshold* is set to something else, then a single 1d tensor of size  $(n_{thresholds}, )$  is returned with shared threshold values for all labels.

#### Return type

(`tuple`)

### Example

```

>>> from torchmetrics.functional.classification import multilabel_roc
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                       [0.45, 0.75, 0.05],
...                       [0.05, 0.55, 0.75],
...                       [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                        [0, 0, 0],
...                        [0, 1, 1],
...                        [1, 1, 1]])
>>> fpr, tpr, thresholds = multilabel_roc(
...     preds, target, num_labels=3, thresholds=None
... )
>>> fpr
[tensor([0.0000, 0.0000, 0.5000, 1.0000]),
 tensor([0.0000, 0.5000, 0.5000, 0.5000, 1.0000]),
 tensor([0., 0., 0., 1.])]
>>> tpr
[tensor([0.0000, 0.5000, 0.5000, 1.0000]),
 tensor([0.0000, 0.0000, 0.5000, 1.0000, 1.0000]),
 tensor([0.0000, 0.3333, 0.6667, 1.0000])]
>>> thresholds
[tensor([1.0000, 0.7500, 0.4500, 0.0500]),
 tensor([1.0000, 0.7500, 0.6500, 0.5500, 0.0500]),
 tensor([1.0000, 0.7500, 0.3500, 0.0500])]
>>> multilabel_roc(
...     preds, target, num_labels=3, thresholds=5
... )
(tensor([[0.0000, 0.0000, 0.0000, 0.5000, 1.0000],
        [0.0000, 0.5000, 0.5000, 0.5000, 1.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 1.0000]]),
 tensor([[0.0000, 0.5000, 0.5000, 0.5000, 1.0000],
        [0.0000, 0.0000, 1.0000, 1.0000, 1.0000],
        [0.0000, 0.3333, 0.3333, 0.6667, 1.0000]]),
 tensor([1.0000, 0.7500, 0.5000, 0.2500, 0.0000]))

```

## 1.47 Specificity

### 1.47.1 Module Interface

**class** torchmetrics.Specificity(task: typing\_extensions.Literal[binary, multiclass, multilabel], threshold: float = 0.5, num\_classes: int | None = None, num\_labels: int | None = None, average: typing\_extensions.Literal[micro, macro, weighted, none] | None = 'micro', multidim\_average: typing\_extensions.Literal[global, samplewise] | None = 'global', top\_k: int | None = 1, ignore\_index: int | None = None, validate\_args: bool = True, \*\*kwargs: ~typing.Any)

Compute *Specificity*.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively. The metric is only proper defined when  $TP + FP \neq 0$ . If this case is encountered for any class/label, the metric for that class/label will be set to 0 and the overall metric may therefore be affected in turn.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or 'multilabel'. See the documentation of `BinarySpecificity`, `MulticlassSpecificity` and `MultilabelSpecificity` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> preds = tensor([2, 0, 2, 1])
>>> target = tensor([1, 1, 2, 0])
>>> specificity = Specificity(task="multiclass", average='macro', num_classes=3)
>>> specificity(preds, target)
tensor(0.6111)
>>> specificity = Specificity(task="multiclass", average='micro', num_classes=3)
>>> specificity(preds, target)
tensor(0.6250)
```

Initialize task metric.

### BinarySpecificity

```
class torchmetrics.classification.BinarySpecificity(threshold=0.5, multidim_average='global',
                                                    ignore_index=None, validate_args=True,
                                                    **kwargs)
```

Compute *Specificity* for binary tasks.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively. The metric is only proper defined when  $\text{TN} + \text{FP} \neq 0$ . If this case is encountered a score of 0 is returned.

As input to forward and update the metric accepts the following input:

- **preds** (`Tensor`): An int or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (`Tensor`): An int tensor of shape (N, ...)

As output to forward and compute the metric returns the following output:

- **bs** (`Tensor`): If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns (N,) vector consisting of a scalar value per sample.

#### Parameters

- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension

- **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinarySpecificity
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinarySpecificity()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.classification import BinarySpecificity
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinarySpecificity()
>>> metric(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.classification import BinarySpecificity
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...               [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = BinarySpecificity(multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.0000, 0.3333])
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

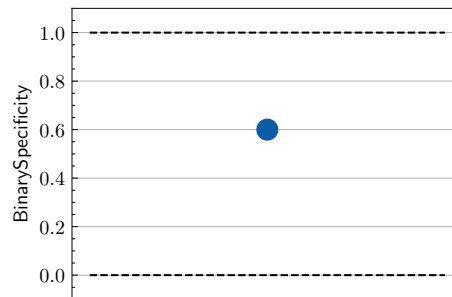
#### Returns

Figure object and Axes object

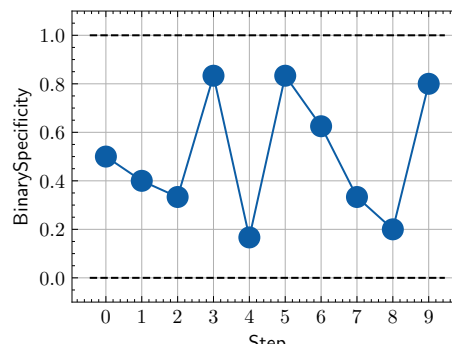
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import BinarySpecificity
>>> metric = BinarySpecificity()
>>> metric.update(rand(10), randint(2,(10,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import BinarySpecificity
>>> metric = BinarySpecificity()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(rand(10), randint(2,(10,))))
>>> fig_, ax_ = metric.plot(values)
```



## MulticlassSpecificity

```
class torchmetrics.classification.MulticlassSpecificity(num_classes, top_k=1, average='macro',
                                                         multidim_average='global',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Compute *Specificity* for multiclass tasks.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively. The metric is only proper defined when  $TN + FP \neq 0$ . If this case is encountered for any class, the metric for that class will be set to 0 and the overall metric may therefore be affected in turn.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int tensor of shape (N, ...) or float tensor of shape (N, C, ...). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** ([Tensor](#)): An int tensor of shape (N, ...)

As output to forward and compute the metric returns the following output:

- **mcs** ([Tensor](#)): The returned shape depends on the `average` and `multidim_average` arguments:
  - If `multidim_average` is set to `global`:
    - \* If `average='micro'/'macro'/'weighted'`, the output will be a scalar tensor
    - \* If `average=None/'none'`, the shape will be (C,)
  - If `multidim_average` is set to `samplewise`:
    - \* If `average='micro'/'macro'/'weighted'`, the shape will be (N,)
    - \* If `average=None/'none'`, the shape will be (N, C)

### Parameters

- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - `micro`: Sum statistics over all labels
  - `macro`: Calculate statistics for each label and average them
  - `weighted`: calculates statistics for each label and computes weighted average using their support
  - `"none"` or `None`: calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassSpecificity
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassSpecificity(num_classes=3)
>>> metric(preds, target)
tensor(0.8889)
>>> mcs = MulticlassSpecificity(num_classes=3, average=None)
>>> mcs(preds, target)
tensor([1.0000, 0.6667, 1.0000])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MulticlassSpecificity
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> metric = MulticlassSpecificity(num_classes=3)
>>> metric(preds, target)
tensor(0.8889)
>>> mcs = MulticlassSpecificity(num_classes=3, average=None)
>>> mcs(preds, target)
tensor([1.0000, 0.6667, 1.0000])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MulticlassSpecificity
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassSpecificity(num_classes=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.7500, 0.6556])
>>> mcs = MulticlassSpecificity(num_classes=3, multidim_average='samplewise',
↪ average=None)
>>> mcs(preds, target)
tensor([[0.7500, 0.7500, 0.7500],
        [0.8000, 0.6667, 0.5000]])
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

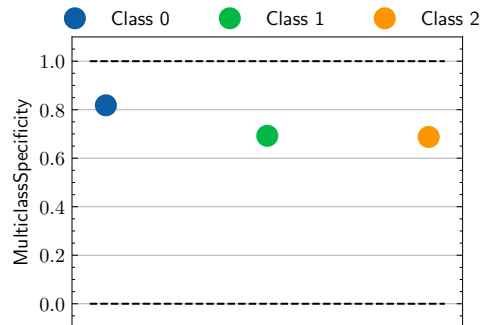
Figure object and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```

>>> from torch import randint
>>> # Example plotting a single value per class
>>> from torchmetrics.classification import MulticlassSpecificity
>>> metric = MulticlassSpecificity(num_classes=3, average=None)
>>> metric.update(randint(3, (20,)), randint(3, (20,)))
>>> fig_, ax_ = metric.plot()

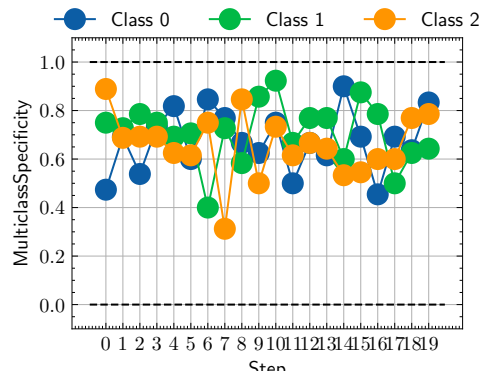
```



```

>>> from torch import randint
>>> # Example plotting a multiple values per class
>>> from torchmetrics.classification import MulticlassSpecificity
>>> metric = MulticlassSpecificity(num_classes=3, average=None)
>>> values = []
>>> for _ in range(20):
...     values.append(metric(randint(3, (20,)), randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)

```





## MultilabelSpecificity

```
class torchmetrics.classification.MultilabelSpecificity(num_labels, threshold=0.5,
                                                         average='macro',
                                                         multidim_average='global',
                                                         ignore_index=None, validate_args=True,
                                                         **kwargs)
```

Compute *Specificity* for multilabel tasks.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively. The metric is only proper defined when  $\text{TN} + \text{FP} \neq 0$ . If this case is encountered for any label, the metric for that label will be set to 0 and the overall metric may therefore be affected in turn.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (*Tensor*): An int or float tensor of shape (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (*Tensor*): An int tensor of shape (N, C, ...)

As output to `forward` and `compute` the metric returns the following output:

- **mls** (*Tensor*): The returned shape depends on the **average** and **multidim\_average** arguments:
  - If **multidim\_average** is set to **global**
    - \* If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
    - \* If **average**=None/'none', the shape will be (C,)
  - If **multidim\_average** is set to **samplewise**
    - \* If **average**='micro'/'macro'/'weighted', the shape will be (N,)
    - \* If **average**=None/'none', the shape will be (N, C)

### Parameters

- **num\_labels** (*int*) – Integer specifying the number of labels
- **threshold** (*float*) – Threshold for transforming probability to binary (0,1) predictions
- **average** (*Optional[Literal['micro', 'macro', 'weighted', 'none']]*) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **multidim\_average** (*Literal['global', 'samplewise']*) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.

- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelSpecificity
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelSpecificity(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> mls = MultilabelSpecificity(num_labels=3, average=None)
>>> mls(preds, target)
tensor([1., 1., 0.])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelSpecificity
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelSpecificity(num_labels=3)
>>> metric(preds, target)
tensor(0.6667)
>>> mls = MultilabelSpecificity(num_labels=3, average=None)
>>> mls(preds, target)
tensor([1., 1., 0.])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MultilabelSpecificity
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = MultilabelSpecificity(num_labels=3, multidim_average='samplewise')
>>> metric(preds, target)
tensor([0.0000, 0.3333])
>>> mls = MultilabelSpecificity(num_labels=3, multidim_average='samplewise',
↪ average=None)
>>> mls(preds, target)
tensor([[0., 0., 0.],
        [0., 0., 1.]])
```

**plot** (*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

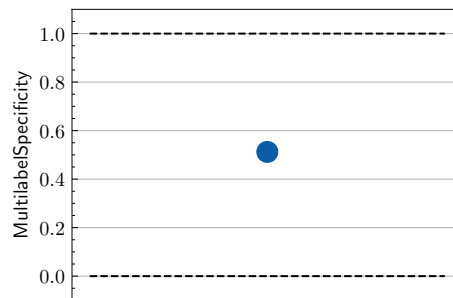
**Returns**

Figure object and Axes object

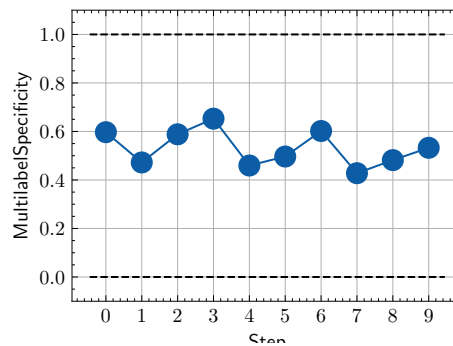
**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import rand, randint
>>> # Example plotting a single value
>>> from torchmetrics.classification import MultilabelSpecificity
>>> metric = MultilabelSpecificity(num_labels=3)
>>> metric.update(randint(2, (20, 3)), randint(2, (20, 3)))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import rand, randint
>>> # Example plotting multiple values
>>> from torchmetrics.classification import MultilabelSpecificity
>>> metric = MultilabelSpecificity(num_labels=3)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(randint(2, (20, 3)), randint(2, (20, 3))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.47.2 Functional Interface

`torchmetrics.functional.specificity(preds, target, task, threshold=0.5, num_classes=None, num_labels=None, average='micro', multidim_average='global', top_k=1, ignore_index=None, validate_args=True)`

Compute *Specificity*. :rtype: `Tensor`

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either 'binary', 'multiclass' or multilabel. See the documentation of `binary_specificity()`, `multiclass_specificity()` and `multilabel_specificity()` for the specific details of each argument influence and examples.

#### LegacyExample:

```
>>> from torch import tensor
>>> preds = tensor([2, 0, 2, 1])
>>> target = tensor([1, 1, 2, 0])
>>> specificity(preds, target, task="multiclass", average='macro', num_
↳ classes=3)
tensor(0.6111)
>>> specificity(preds, target, task="multiclass", average='micro', num_
↳ classes=3)
tensor(0.6250)
```

### binary\_specificity

`torchmetrics.functional.classification.binary_specificity(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Compute *Specificity* for binary tasks.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** (int tensor): (N, ...)

#### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions

- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the `N` axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Return type**`Tensor`**Returns**

If `multidim_average` is set to `global`, the metric returns a scalar value. If `multidim_average` is set to `samplewise`, the metric returns `(N,)` vector consisting of a scalar value per sample.

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_specificity
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> binary_specificity(preds, target)
tensor(0.6667)
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import binary_specificity
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_specificity(preds, target)
tensor(0.6667)
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import binary_specificity
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> binary_specificity(preds, target, multidim_average='samplewise')
tensor([0.0000, 0.3333])
```

## multiclass\_specificity

```
torchmetrics.functional.classification.multiclass_specificity(preds, target, num_classes,
                                                             average='macro', top_k=1,
                                                             multidim_average='global',
                                                             ignore_index=None,
                                                             validate_args=True)
```

Compute *Specificity* for multiclass tasks.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively.

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Returns

- If **multidim\_average** is set to **global**:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If **average**=None/'none', the shape will be (C,)

- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N,)
  - If average=None/'none', the shape will be (N, C)

**Return type**

The returned shape depends on the average and multidim\_average arguments

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_specificity
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_specificity(preds, target, num_classes=3)
tensor(0.8889)
>>> multiclass_specificity(preds, target, num_classes=3, average=None)
tensor([1.0000, 0.6667, 1.0000])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_specificity
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> multiclass_specificity(preds, target, num_classes=3)
tensor(0.8889)
>>> multiclass_specificity(preds, target, num_classes=3, average=None)
tensor([1.0000, 0.6667, 1.0000])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_specificity
>>> target = tensor([[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]])
>>> preds = tensor([[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]])
>>> multiclass_specificity(preds, target, num_classes=3, multidim_average=
↳ 'samplewise')
tensor([0.7500, 0.6556])
>>> multiclass_specificity(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0.7500, 0.7500, 0.7500],
        [0.8000, 0.6667, 0.5000]])
```

## multilabel\_specificity

```
torchmetrics.functional.classification.multilabel_specificity(preds, target, num_labels,
                                                             threshold=0.5, average='macro',
                                                             multidim_average='global',
                                                             ignore_index=None,
                                                             validate_args=True)
```

Compute *Specificity* for multilabel tasks.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Where TN and FP represent the number of true negatives and false positives respectively.

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - micro: Sum statistics over all labels
  - macro: Calculate statistics for each label and average them
  - weighted: calculates statistics for each label and computes weighted average using their support
  - "none" or None: calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to False for faster computations.

### Returns

- If **multidim\_average** is set to global:
  - If **average**='micro'/'macro'/'weighted', the output will be a scalar tensor
  - If **average**=None/'none', the shape will be (C,)



- If `multidim_average` is set to `samplewise`:
  - If `average='micro'/'macro'/'weighted'`, the shape will be `(N,)`
  - If `average=None/'none'`, the shape will be `(N, C)`

**Return type**

The returned shape depends on the `average` and `multidim_average` arguments

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_specificity
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_specificity(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_specificity(preds, target, num_labels=3, average=None)
tensor([1., 1., 0.])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_specificity
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_specificity(preds, target, num_labels=3)
tensor(0.6667)
>>> multilabel_specificity(preds, target, num_labels=3, average=None)
tensor([1., 1., 0.])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multilabel_specificity
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> multilabel_specificity(preds, target, num_labels=3, multidim_average=
↳ 'samplewise')
tensor([0.0000, 0.3333])
>>> multilabel_specificity(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[0., 0., 0.],
        [0., 0., 1.]])
```

## 1.48 Specificity At Sensitivity

### 1.48.1 Module Interface

#### BinarySpecificityAtSensitivity

```
class torchmetrics.classification.BinarySpecificityAtSensitivity(min_sensitivity,
                                                                thresholds=None,
                                                                ignore_index=None,
                                                                validate_args=True, **kwargs)
```

Compute the highest possible specificity value given the minimum sensitivity thresholds provided.

This is done by first calculating the Receiver Operating Characteristic (ROC) curve for different thresholds and then find the specificity for a given sensitivity level.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

#### Parameters

- **min\_sensitivity** (float) – float value specifying minimum sensitivity threshold.
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** (Any) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Returns

a tuple of 2 tensors containing:

- **specificity**: an scalar tensor with the maximum specificity for the given sensitivity level
- **threshold**: an scalar tensor with the corresponding threshold level

#### Return type

(tuple)

### Example

```
>>> from torchmetrics.classification import BinarySpecificityAtSensitivity
>>> from torch import tensor
>>> preds = tensor([0, 0.5, 0.4, 0.1])
>>> target = tensor([0, 1, 1, 1])
>>> metric = BinarySpecificityAtSensitivity(min_sensitivity=0.5, thresholds=None)
>>> metric(preds, target)
(tensor(1.), tensor(0.4000))
>>> metric = BinarySpecificityAtSensitivity(min_sensitivity=0.5, thresholds=5)
>>> metric(preds, target)
(tensor(1.), tensor(0.2500))
```

### MulticlassSpecificityAtSensitivity

```
class torchmetrics.classification.MulticlassSpecificityAtSensitivity(num_classes,
                                                                    min_sensitivity,
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True,
                                                                    **kwargs)
```

Compute the highest possible specificity value given the minimum sensitivity thresholds provided.

This is done by first calculating the Receiver Operating Characteristic (ROC) curve for different thresholds and then find the specificity for a given sensitivity level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{\text{samples}})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{\text{thresholds}} \times n_{\text{classes}})$  (constant memory).

#### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **min\_sensitivity** (`float`) – float value specifying minimum sensitivity threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation

- If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (*bool*) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

### Returns

a tuple of either 2 tensors or 2 lists containing

- **specificity**: an 1d tensor of size (n\_classes, ) with the maximum specificity for the given sensitivity level per class
- **thresholds**: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

### Return type

(tuple)

### Example

```
>>> from torchmetrics.classification import MulticlassSpecificityAtSensitivity
>>> from torch import tensor
>>> preds = tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                [0.05, 0.75, 0.05, 0.05, 0.05],
...                [0.05, 0.05, 0.75, 0.05, 0.05],
...                [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = tensor([0, 1, 3, 2])
>>> metric = MulticlassSpecificityAtSensitivity(num_classes=5, min_sensitivity=0.5,
↳ thresholds=None)
>>> metric(preds, target)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 5.0000e-02, 5.0000e-
↳ 02, 1.0000e+06]))
>>> metric = MulticlassSpecificityAtSensitivity(num_classes=5, min_sensitivity=0.5,
↳ thresholds=5)
>>> metric(preds, target)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 0.0000e+00, 0.
↳ 0000e+00, 1.0000e+06]))
```

## MultilabelSpecificityAtSensitivity

```
class torchmetrics.classification.MultilabelSpecificityAtSensitivity(num_labels,
                                                                    min_sensitivity,
                                                                    thresholds=None,
                                                                    ignore_index=None,
                                                                    validate_args=True,
                                                                    **kwargs)
```

Compute the highest possible specificity value given the minimum sensitivity thresholds provided.

This is done by first calculating the Receiver Operating Characteristic (ROC) curve for different thresholds and then find the specificity for a given sensitivity level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{labels})$  (constant memory).

#### Parameters

- **num\_labels** (`int`) – Integer specifying the number of labels
- **min\_sensitivity** (`float`) – float value specifying minimum sensitivity threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Returns

a tuple of either 2 tensors or 2 lists containing

- **specificity**: an 1d tensor of size (n\_classes, ) with the maximum specificity for the given sensitivity level per class
- **thresholds**: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

#### Return type

(tuple)

#### Example

```
>>> from torchmetrics.classification import MultilabelSpecificityAtSensitivity
>>> from torch import tensor
>>> preds = tensor([[0.75, 0.05, 0.35],
...                [0.45, 0.75, 0.05],
...                [0.05, 0.55, 0.75],
...                [0.05, 0.65, 0.05]])
>>> target = tensor([[1, 0, 1],
```

(continues on next page)

(continued from previous page)

```

...         [0, 0, 0],
...         [0, 1, 1],
...         [1, 1, 1]])
>>> metric = MultilabelSpecificityAtSensitivity(num_labels=3, min_sensitivity=0.5,
↳ thresholds=None)
>>> metric(preds, target)
(tensor([1.0000, 0.5000, 1.0000]), tensor([0.7500, 0.6500, 0.3500]))
>>> metric = MultilabelSpecificityAtSensitivity(num_labels=3, min_sensitivity=0.5,
↳ thresholds=5)
>>> metric(preds, target)
(tensor([1.0000, 0.5000, 1.0000]), tensor([0.7500, 0.5000, 0.2500]))

```

## 1.48.2 Functional Interface

### binary\_specificity\_at\_sensitivity

```

torchmetrics.functional.classification.binary_specificity_at_sensitivity(preds, target,
                                                                           min_sensitivity,
                                                                           thresholds=None,
                                                                           ignore_index=None,
                                                                           validate_args=True)

```

Compute the highest possible specificity value given the minimum sensitivity levels provided for binary tasks.

This is done by first calculating the Receiver Operating Characteristic (ROC) curve for different thresholds and then find the specificity for a given sensitivity level.

Accepts the following input tensors:

- **preds** (float tensor): (N, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds})$  (constant memory).

#### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **min\_sensitivity** ([float](#)) – float value specifying minimum sensitivity threshold.
- **thresholds** ([Union\[int, List\[float\], Tensor, None\]](#)) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.

- If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
- If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

**Returns**

a tuple of 2 tensors containing:

- specificity: a scalar tensor with the maximum specificity for the given sensitivity level
- threshold: a scalar tensor with the corresponding threshold level

**Return type**

(tuple)

**Example**

```
>>> from torchmetrics.functional.classification import binary_specificity_at_
    ↳ sensitivity
>>> preds = torch.tensor([0, 0.5, 0.4, 0.1])
>>> target = torch.tensor([0, 1, 1, 1])
>>> binary_specificity_at_sensitivity(preds, target, min_sensitivity=0.5,
    ↳ thresholds=None)
(tensor(1.), tensor(0.4000))
>>> binary_specificity_at_sensitivity(preds, target, min_sensitivity=0.5,
    ↳ thresholds=5)
(tensor(1.), tensor(0.2500))
```

**multiclass\_specificity\_at\_sensitivity**

```
torchmetrics.functional.classification.multiclass_specificity_at_sensitivity(preds, target,
                                                                              num_classes,
                                                                              min_sensitivity,
                                                                              thresh-
                                                                              olds=None,
                                                                              ig-
                                                                              nore_index=None,
                                                                              vali-
                                                                              date_args=True)
```

Compute the highest possible specificity value given the minimum sensitivity level provided for multiclass tasks.

This is done by first calculating the Receiver Operating Characteristic (ROC) curve for different thresholds and then find the specificity for a given sensitivity level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply softmax per sample.

- **target** (int tensor): (N, ...). Target should be a tensor containing ground truth labels, and therefore only contain values in the [0, n\_classes-1] range (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{samples})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{thresholds} \times n_{classes})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels
- **num\_classes** (int) – Integer specifying the number of classes
- **min\_sensitivity** (float) – float value specifying minimum sensitivity threshold.
- **thresholds** (Union[int, List[float], Tensor, None]) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (Optional[int]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (bool) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

#### Returns

a tuple of either 2 tensors or 2 lists containing

- **recall**: an 1d tensor of size (n\_classes, ) with the maximum recall for the given precision level per class
- **thresholds**: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

#### Return type

(tuple)



### Example

```
>>> from torchmetrics.functional.classification import multiclass_specificity_at_
    ↪ sensitivity
>>> preds = torch.tensor([[0.75, 0.05, 0.05, 0.05, 0.05],
...                        [0.05, 0.75, 0.05, 0.05, 0.05],
...                        [0.05, 0.05, 0.75, 0.05, 0.05],
...                        [0.05, 0.05, 0.05, 0.75, 0.05]])
>>> target = torch.tensor([0, 1, 3, 2])
>>> multiclass_specificity_at_sensitivity(preds, target, num_classes=5, min_
    ↪ sensitivity=0.5, thresholds=None)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 5.0000e-02, 5.0000e-
    ↪ 02, 1.0000e+06]))
>>> multiclass_specificity_at_sensitivity(preds, target, num_classes=5, min_
    ↪ sensitivity=0.5, thresholds=5)
(tensor([1., 1., 0., 0., 0.]), tensor([7.5000e-01, 7.5000e-01, 0.0000e+00, 0.
    ↪ 0000e+00, 1.0000e+06]))
```

### multilabel\_specificity\_at\_sensitivity

```
torchmetrics.functional.classification.multilabel_specificity_at_sensitivity(preds, target,
                                                                              num_labels,
                                                                              min_sensitivity,
                                                                              thresh-
                                                                              olds=None,
                                                                              ig-
                                                                              nore_index=None,
                                                                              vali-
                                                                              date_args=True)
```

Compute the highest possible specificity value given the minimum sensitivity level provided for multilabel tasks.

This is done by first calculating the Receiver Operating Characteristic (ROC) curve for different thresholds and then find the specificity for a given sensitivity level.

Accepts the following input tensors:

- **preds** (float tensor): (N, C, ...). Preds should be a tensor containing probabilities or logits for each observation. If preds has values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element.
- **target** (int tensor): (N, C, ...). Target should be a tensor containing ground truth labels, and therefore only contain {0,1} values (except if *ignore\_index* is specified).

Additional dimension ... will be flattened into the batch dimension.

The implementation both supports calculating the metric in a non-binned but accurate version and a binned version that is less accurate but more memory efficient. Setting the *thresholds* argument to *None* will activate the non-binned version that uses memory of size  $\mathcal{O}(n_{\text{samples}})$  whereas setting the *thresholds* argument to either an integer, list or a 1d tensor will use a binned version that uses memory of size  $\mathcal{O}(n_{\text{thresholds}} \times n_{\text{labels}})$  (constant memory).

#### Parameters

- **preds** (Tensor) – Tensor with predictions
- **target** (Tensor) – Tensor with true labels

- **num\_labels** (`int`) – Integer specifying the number of labels
- **min\_sensitivity** (`float`) – float value specifying minimum sensitivity threshold.
- **thresholds** (`Union[int, List[float], Tensor, None]`) – Can be one of:
  - If set to *None*, will use a non-binned approach where thresholds are dynamically calculated from all the data. Most accurate but also most memory consuming approach.
  - If set to an *int* (larger than 1), will use that number of thresholds linearly spaced from 0 to 1 as bins for the calculation.
  - If set to an *list* of floats, will use the indicated thresholds in the list as bins for the calculation
  - If set to an 1d *tensor* of floats, will use the indicated thresholds in the tensor as bins for the calculation.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to *False* for faster computations.

### Returns

a tuple of either 2 tensors or 2 lists containing

- **specificity**: an 1d tensor of size (n\_classes, ) with the maximum recall for the given precision level per class
- **thresholds**: an 1d tensor of size (n\_classes, ) with the corresponding threshold level per class

### Return type

(tuple)

### Example

```
>>> from torchmetrics.functional.classification import multilabel_specificity_at_
↳sensitivity
>>> preds = torch.tensor([[0.75, 0.05, 0.35],
...                        [0.45, 0.75, 0.05],
...                        [0.05, 0.55, 0.75],
...                        [0.05, 0.65, 0.05]])
>>> target = torch.tensor([[1, 0, 1],
...                         [0, 0, 0],
...                         [0, 1, 1],
...                         [1, 1, 1]])
>>> multilabel_specificity_at_sensitivity(preds, target, num_labels=3, min_
↳sensitivity=0.5, thresholds=None)
(tensor([1.0000, 0.5000, 1.0000]), tensor([0.7500, 0.6500, 0.3500]))
>>> multilabel_specificity_at_sensitivity(preds, target, num_labels=3, min_
↳sensitivity=0.5, thresholds=5)
(tensor([1.0000, 0.5000, 1.0000]), tensor([0.7500, 0.5000, 0.2500]))
```

## 1.49 Stat Scores

### 1.49.1 Module Interface

#### StatScores

```
class torchmetrics.StatScores(task: typing_extensions.Literal[binary, multiclass, multilabel], threshold:
    float = 0.5, num_classes: int | None = None, num_labels: int | None = None,
    average: typing_extensions.Literal[micro, macro, weighted, none] | None =
    'micro', multidim_average: typing_extensions.Literal[global, samplewise] |
    None = 'global', top_k: int | None = 1, ignore_index: int | None = None,
    validate_args: bool = True, **kwargs: ~typing.Any)
```

Compute the number of true positives, false positives, true negatives, false negatives and the support.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the `task` argument to either `'binary'`, `'multiclass'` or `multilabel`. See the documentation of `BinaryStatScores`, `MulticlassStatScores` and `MultilabelStatScores` for the specific details of each argument influence and examples.

#### Legacy Example:

```
>>> from torch import tensor
>>> preds = tensor([1, 0, 2, 1])
>>> target = tensor([1, 1, 2, 0])
>>> stat_scores = StatScores(task="multiclass", num_classes=3, average='micro')
>>> stat_scores(preds, target)
tensor([2, 2, 6, 2, 4])
>>> stat_scores = StatScores(task="multiclass", num_classes=3, average=None)
>>> stat_scores(preds, target)
tensor([[0, 1, 2, 1, 1],
        [1, 1, 1, 1, 2],
        [1, 0, 3, 0, 1]])
```

Initialize task metric.

#### BinaryStatScores

```
class torchmetrics.classification.BinaryStatScores(threshold=0.5, multidim_average='global',
    ignore_index=None, validate_args=True,
    **kwargs)
```

Compute true positives, false positives, true negatives, false negatives and the support for binary tasks.

Related to [Type I and Type II errors](#).

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int or float tensor of shape (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in `threshold`.
- **target** ([Tensor](#)): An int tensor of shape (N, ...)

As output to forward and compute the metric returns the following output:

- `bss` (`Tensor`): A tensor of shape  $(\dots, 5)$ , where the last dimension corresponds to  $[tp, fp, tn, fn, sup]$  (`sup` stands for support and equals  $tp + fn$ ). The shape depends on the `multidim_average` parameter:
- If `multidim_average` is set to `global`, the shape will be  $(5,)$
- If `multidim_average` is set to `samplewise`, the shape will be  $(N, 5)$

#### Parameters

- **`threshold`** (`float`) – Threshold for transforming probability to binary  $\{0,1\}$  predictions
- **`multidim_average`** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions  $\dots$  should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the  $N$  axis. The statistics in this case are calculated over the additional dimensions.
- **`ignore_index`** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **`validate_args`** (`bool`) – `bool` indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **`kwargs`** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import BinaryStatScores
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> metric = BinaryStatScores()
>>> metric(preds, target)
tensor([2, 1, 2, 1, 3])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import BinaryStatScores
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> metric = BinaryStatScores()
>>> metric(preds, target)
tensor([2, 1, 2, 1, 3])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import BinaryStatScores
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = BinaryStatScores(multidim_average='samplewise')
>>> metric(preds, target)
tensor([[2, 3, 0, 1, 3],
        [0, 2, 1, 3, 3]])
```

## MulticlassStatScores

```
class torchmetrics.classification.MulticlassStatScores(num_classes, top_k=1, average='macro',
                                                       multidim_average='global',
                                                       ignore_index=None, validate_args=True,
                                                       **kwargs)
```

Computes true positives, false positives, true negatives, false negatives and the support for multiclass tasks.

Related to [Type I and Type II errors](#).

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int tensor of shape (N, ...) or float tensor of shape (N, C, ...). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** ([Tensor](#)): An int tensor of shape (N, ...)

As output to forward and compute the metric returns the following output:

- **mcss** ([Tensor](#)): A tensor of shape (... , 5), where the last dimension corresponds to [tp, fp, tn, fn, sup] (sup stands for support and equals tp + fn). The shape depends on average and multidim\_average parameters:
- If multidim\_average is set to global
- If average='micro'/'macro'/'weighted', the shape will be (5,)
- If average=None/'none', the shape will be (C, 5)
- If multidim\_average is set to samplewise
- If average='micro'/'macro'/'weighted', the shape will be (N, 5)
- If average=None/'none', the shape will be (N, C, 5)

### Parameters

- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - micro: Sum statistics over all labels
  - macro: Calculate statistics for each label and average them
  - weighted: calculates statistics for each label and computes weighted average using their support
  - "none" or None: calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension
  - samplewise: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation

- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MulticlassStatScores
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> metric = MulticlassStatScores(num_classes=3, average='micro')
>>> metric(preds, target)
tensor([3, 1, 7, 1, 4])
>>> mcsc = MulticlassStatScores(num_classes=3, average=None)
>>> mcsc(preds, target)
tensor([[1, 0, 2, 1, 2],
        [1, 1, 2, 0, 1],
        [1, 0, 3, 0, 1]])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MulticlassStatScores
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> metric = MulticlassStatScores(num_classes=3, average='micro')
>>> metric(preds, target)
tensor([3, 1, 7, 1, 4])
>>> mcsc = MulticlassStatScores(num_classes=3, average=None)
>>> mcsc(preds, target)
tensor([[1, 0, 2, 1, 2],
        [1, 1, 2, 0, 1],
        [1, 0, 3, 0, 1]])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MulticlassStatScores
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> metric = MulticlassStatScores(num_classes=3, multidim_average="samplewise",
    ↪ average='micro')
>>> metric(preds, target)
tensor([[3, 3, 9, 3, 6],
        [2, 4, 8, 4, 6]])
>>> mcsc = MulticlassStatScores(num_classes=3, multidim_average="samplewise",
    ↪ average=None)
>>> mcsc(preds, target)
tensor([[[2, 1, 3, 0, 2],
        [0, 1, 3, 2, 2],
        [1, 1, 3, 1, 2]],
        [[0, 1, 4, 1, 1],
```

(continues on next page)

(continued from previous page)

```
[1, 1, 2, 2, 3],
[1, 2, 2, 1, 2]]])
```

## MultilabelStatScores

```
class torchmetrics.classification.MultilabelStatScores(num_labels, threshold=0.5,
                                                       average='macro',
                                                       multidim_average='global',
                                                       ignore_index=None, validate_args=True,
                                                       **kwargs)
```

Compute true positives, false positives, true negatives, false negatives and the support for multilabel tasks.

Related to [Type I and Type II errors](#).

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): An int or float tensor of shape (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** ([Tensor](#)): An int tensor of shape (N, C, ...)

As output to forward and compute the metric returns the following output:

- **mlss** ([Tensor](#)): A tensor of shape (... , 5), where the last dimension corresponds to [tp, fp, tn, fn, sup] (sup stands for support and equals tp + fn). The shape depends on **average** and **multidim\_average** parameters:
- If **multidim\_average** is set to **global**
- If **average**='micro'/'macro'/'weighted', the shape will be (5,)
- If **average**=None/'none', the shape will be (C, 5)
- If **multidim\_average** is set to **samplewise**
- If **average**='micro'/'macro'/'weighted', the shape will be (N, 5)
- If **average**=None/'none', the shape will be (N, C, 5)

### Parameters

- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - micro: Sum statistics over all labels
  - macro: Calculate statistics for each label and average them
  - weighted: calculates statistics for each label and computes weighted average using their support
  - "none" or None: calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - global: Additional dimensions are flattened along the batch dimension

- **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.classification import MultilabelStatScores
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> metric = MultilabelStatScores(num_labels=3, average='micro')
>>> metric(preds, target)
tensor([2, 1, 2, 1, 3])
>>> mlss = MultilabelStatScores(num_labels=3, average=None)
>>> mlss(preds, target)
tensor([[1, 0, 1, 0, 1],
        [0, 0, 1, 1, 1],
        [1, 1, 0, 0, 1]])
```

Example (preds is float tensor):

```
>>> from torchmetrics.classification import MultilabelStatScores
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> metric = MultilabelStatScores(num_labels=3, average='micro')
>>> metric(preds, target)
tensor([2, 1, 2, 1, 3])
>>> mlss = MultilabelStatScores(num_labels=3, average=None)
>>> mlss(preds, target)
tensor([[1, 0, 1, 0, 1],
        [0, 0, 1, 1, 1],
        [1, 1, 0, 0, 1]])
```

Example (multidim tensors):

```
>>> from torchmetrics.classification import MultilabelStatScores
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> metric = MultilabelStatScores(num_labels=3, multidim_average='samplewise',
    ↪ average='micro')
>>> metric(preds, target)
tensor([[2, 3, 0, 1, 3],
        [0, 2, 1, 3, 3]])
>>> mlss = MultilabelStatScores(num_labels=3, multidim_average='samplewise',
    ↪ average=None)
>>> mlss(preds, target)
tensor([[[1, 1, 0, 0, 1],
```

(continues on next page)



(continued from previous page)

```
[1, 1, 0, 0, 1],
[0, 1, 0, 1, 1]],
[[0, 0, 0, 2, 2],
[0, 2, 0, 0, 0],
[0, 0, 1, 1, 1]]])
```

## 1.49.2 Functional Interface

### stat\_scores

`torchmetrics.functional.stat_scores(preds, target, task, threshold=0.5, num_classes=None, num_labels=None, average='micro', multidim_average='global', top_k=1, ignore_index=None, validate_args=True)`

Compute the number of true positives, false positives, true negatives, false negatives and the support.

This function is a simple wrapper to get the task specific versions of this metric, which is done by setting the task argument to either 'binary', 'multiclass' or multilabel. See the documentation of `binary_stat_scores()`, `multiclass_stat_scores()` and `multilabel_stat_scores()` for the specific details of each argument influence and examples.

#### Return type

Tensor

#### Legacy Example:

```
>>> from torch import tensor
>>> preds = tensor([1, 0, 2, 1])
>>> target = tensor([1, 1, 2, 0])
>>> stat_scores(preds, target, task='multiclass', num_classes=3, average='micro
↳ ')
tensor([2, 2, 6, 2, 4])
>>> stat_scores(preds, target, task='multiclass', num_classes=3, average=None)
tensor([[0, 1, 2, 1, 1],
        [1, 1, 1, 1, 2],
        [1, 0, 3, 0, 1]])
```

### binary\_stat\_scores

`torchmetrics.functional.classification.binary_stat_scores(preds, target, threshold=0.5, multidim_average='global', ignore_index=None, validate_args=True)`

Compute the true positives, false positives, true negatives, false negatives, support for binary tasks.

Related to [Type I and Type II errors](#).

Accepts the following input tensors:

- **preds** (int or float tensor): (N, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** (`Tensor`) – Tensor with predictions
- **target** (`Tensor`) – Tensor with true labels
- **threshold** (`float`) – Threshold for transforming probability to binary {0,1} predictions
- **multidim\_average** (`Literal['global', 'samplewise']`) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - `global`: Additional dimensions are flattened along the batch dimension
  - `samplewise`: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** (`Optional[int]`) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** (`bool`) – bool indicating if input arguments and tensors should be validated for correctness. Set to `False` for faster computations.

### Return type

`Tensor`

### Returns

The metric returns a tensor of shape `(..., 5)`, where the last dimension corresponds to `[tp, fp, tn, fn, sup]` (sup stands for support and equals `tp + fn`). The shape depends on the `multidim_average` parameter:

- If `multidim_average` is set to `global`, the shape will be `(5,)`
- If `multidim_average` is set to `samplewise`, the shape will be `(N, 5)`

#### Example (preds is int tensor):

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import binary_stat_scores
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0, 0, 1, 1, 0, 1])
>>> binary_stat_scores(preds, target)
tensor([2, 1, 2, 1, 3])
```

#### Example (preds is float tensor):

```
>>> from torchmetrics.functional.classification import binary_stat_scores
>>> target = tensor([0, 1, 0, 1, 0, 1])
>>> preds = tensor([0.11, 0.22, 0.84, 0.73, 0.33, 0.92])
>>> binary_stat_scores(preds, target)
tensor([2, 1, 2, 1, 3])
```

#### Example (multidim tensors):

```
>>> from torchmetrics.functional.classification import binary_stat_scores
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> binary_stat_scores(preds, target, multidim_average='samplewise')
tensor([[2, 3, 0, 1, 3],
        [0, 2, 1, 3, 3]])
```

## multiclass\_stat\_scores

```
torchmetrics.functional.classification.multiclass_stat_scores(preds, target, num_classes,
                                                             average='macro', top_k=1,
                                                             multidim_average='global',
                                                             ignore_index=None,
                                                             validate_args=True)
```

Compute the true positives, false positives, true negatives, false negatives and support for multiclass tasks.

Related to [Type I and Type II errors](#).

Accepts the following input tensors:

- **preds**: (N, ...) (int tensor) or (N, C, ...) (float tensor). If preds is a floating point we apply `torch.argmax` along the C dimension to automatically convert probabilities/logits into an int tensor.
- **target** (int tensor): (N, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_classes** ([int](#)) – Integer specifying the number of classes
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **top\_k** ([int](#)) – Number of highest probability or logit score predictions considered to find the correct label. Only works when preds contain probabilities/logits.
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Return type

[Tensor](#)

### Returns

The metric returns a tensor of shape (... , 5), where the last dimension corresponds to [tp, fp, tn, fn, sup] (sup stands for support and equals tp + fn). The shape depends on **average** and **multidim\_average** parameters:

- If **multidim\_average** is set to **global**:

- If average='micro'/'macro'/'weighted', the shape will be (5,)
- If average=None/'none', the shape will be (C, 5)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N, 5)
  - If average=None/'none', the shape will be (N, C, 5)

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multiclass_stat_scores
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([2, 1, 0, 1])
>>> multiclass_stat_scores(preds, target, num_classes=3, average='micro')
tensor([3, 1, 7, 1, 4])
>>> multiclass_stat_scores(preds, target, num_classes=3, average=None)
tensor([[1, 0, 2, 1, 2],
        [1, 1, 2, 0, 1],
        [1, 0, 3, 0, 1]])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multiclass_stat_scores
>>> target = tensor([2, 1, 0, 0])
>>> preds = tensor([[0.16, 0.26, 0.58],
...                [0.22, 0.61, 0.17],
...                [0.71, 0.09, 0.20],
...                [0.05, 0.82, 0.13]])
>>> multiclass_stat_scores(preds, target, num_classes=3, average='micro')
tensor([3, 1, 7, 1, 4])
>>> multiclass_stat_scores(preds, target, num_classes=3, average=None)
tensor([[1, 0, 2, 1, 2],
        [1, 1, 2, 0, 1],
        [1, 0, 3, 0, 1]])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multiclass_stat_scores
>>> target = tensor([[[0, 1], [2, 1], [0, 2]], [[1, 1], [2, 0], [1, 2]]])
>>> preds = tensor([[[0, 2], [2, 0], [0, 1]], [[2, 2], [2, 1], [1, 0]]])
>>> multiclass_stat_scores(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average='micro')
tensor([[3, 3, 9, 3, 6],
        [2, 4, 8, 4, 6]])
>>> multiclass_stat_scores(preds, target, num_classes=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[[2, 1, 3, 0, 2],
        [0, 1, 3, 2, 2],
        [1, 1, 3, 1, 2]],
        [[0, 1, 4, 1, 1],
        [1, 1, 2, 2, 3],
        [1, 2, 2, 1, 2]]])
```

## multilabel\_stat\_scores

```
torchmetrics.functional.classification.multilabel_stat_scores(preds, target, num_labels,
                                                             threshold=0.5, average='macro',
                                                             multidim_average='global',
                                                             ignore_index=None,
                                                             validate_args=True)
```

Compute the true positives, false positives, true negatives, false negatives and support for multilabel tasks.

Related to [Type I and Type II errors](#).

Accepts the following input tensors:

- **preds** (int or float tensor): (N, C, ...). If preds is a floating point tensor with values outside [0,1] range we consider the input to be logits and will auto apply sigmoid per element. Additionally, we convert to int tensor with thresholding using the value in **threshold**.
- **target** (int tensor): (N, C, ...)

### Parameters

- **preds** ([Tensor](#)) – Tensor with predictions
- **target** ([Tensor](#)) – Tensor with true labels
- **num\_labels** ([int](#)) – Integer specifying the number of labels
- **threshold** ([float](#)) – Threshold for transforming probability to binary (0,1) predictions
- **average** ([Optional](#)[[Literal](#)['micro', 'macro', 'weighted', 'none']]) – Defines the reduction that is applied over labels. Should be one of the following:
  - **micro**: Sum statistics over all labels
  - **macro**: Calculate statistics for each label and average them
  - **weighted**: calculates statistics for each label and computes weighted average using their support
  - **"none"** or **None**: calculates statistic for each label and applies no reduction
- **multidim\_average** ([Literal](#)['global', 'samplewise']) – Defines how additionally dimensions ... should be handled. Should be one of the following:
  - **global**: Additional dimensions are flattened along the batch dimension
  - **samplewise**: Statistic will be calculated independently for each sample on the N axis. The statistics in this case are calculated over the additional dimensions.
- **ignore\_index** ([Optional](#)[[int](#)]) – Specifies a target value that is ignored and does not contribute to the metric calculation
- **validate\_args** ([bool](#)) – bool indicating if input arguments and tensors should be validated for correctness. Set to **False** for faster computations.

### Return type

[Tensor](#)

### Returns

The metric returns a tensor of shape (... , 5), where the last dimension corresponds to [tp, fp, tn, fn, sup] (sup stands for support and equals tp + fn). The shape depends on **average** and **multidim\_average** parameters:

- If **multidim\_average** is set to **global**:

- If average='micro'/'macro'/'weighted', the shape will be (5,)
- If average=None/'none', the shape will be (C, 5)
- If multidim\_average is set to samplewise:
  - If average='micro'/'macro'/'weighted', the shape will be (N, 5)
  - If average=None/'none', the shape will be (N, C, 5)

**Example (preds is int tensor):**

```
>>> from torch import tensor
>>> from torchmetrics.functional.classification import multilabel_stat_scores
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0, 0, 1], [1, 0, 1]])
>>> multilabel_stat_scores(preds, target, num_labels=3, average='micro')
tensor([2, 1, 2, 1, 3])
>>> multilabel_stat_scores(preds, target, num_labels=3, average=None)
tensor([[1, 0, 1, 0, 1],
        [0, 0, 1, 1, 1],
        [1, 1, 0, 0, 1]])
```

**Example (preds is float tensor):**

```
>>> from torchmetrics.functional.classification import multilabel_stat_scores
>>> target = tensor([[0, 1, 0], [1, 0, 1]])
>>> preds = tensor([[0.11, 0.22, 0.84], [0.73, 0.33, 0.92]])
>>> multilabel_stat_scores(preds, target, num_labels=3, average='micro')
tensor([2, 1, 2, 1, 3])
>>> multilabel_stat_scores(preds, target, num_labels=3, average=None)
tensor([[1, 0, 1, 0, 1],
        [0, 0, 1, 1, 1],
        [1, 1, 0, 0, 1]])
```

**Example (multidim tensors):**

```
>>> from torchmetrics.functional.classification import multilabel_stat_scores
>>> target = tensor([[[0, 1], [1, 0], [0, 1]], [[1, 1], [0, 0], [1, 0]]])
>>> preds = tensor([[[0.59, 0.91], [0.91, 0.99], [0.63, 0.04]],
...                [[0.38, 0.04], [0.86, 0.780], [0.45, 0.37]]])
>>> multilabel_stat_scores(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average='micro')
tensor([2, 3, 0, 1, 3],
        [0, 2, 1, 3, 3])
>>> multilabel_stat_scores(preds, target, num_labels=3, multidim_average=
↳ 'samplewise', average=None)
tensor([[[1, 1, 0, 0, 1],
         [1, 1, 0, 0, 1],
         [0, 1, 0, 1, 1]],
        [[0, 0, 0, 2, 2],
         [0, 2, 0, 0, 0],
         [0, 0, 1, 1, 1]]])
```

## 1.50 Complete Intersection Over Union (ciou)

### 1.50.1 Module Interface

```
class torchmetrics.detection.ciou.CompleteIntersectionOverUnion(box_format='xyxy',
                                                                iou_threshold=None,
                                                                class_metrics=False, **kwargs)
```

Computes Complete Intersection Over Union (CIoU) <<https://arxiv.org/abs/2005.03572>>`\_.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`List`): A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image). Parameters that should be provided per dict:
  - **boxes**: (`FloatTensor`) of shape `(num_boxes, 4)` containing `num_boxes` detection boxes of the format specified in the constructor. By default, this method expects `(xmin, ymin, xmax, ymax)` in absolute image coordinates.
  - **scores**: `FloatTensor` of shape `(num_boxes)` containing detection scores for the boxes.
  - **labels**: `IntTensor` of shape `(num_boxes)` containing 0-indexed detection classes for the boxes.
- **target** (`List`): A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image). Parameters that should be provided per dict:
  - **boxes**: `FloatTensor` of shape `(num_boxes, 4)` containing `num_boxes` ground truth boxes of the format specified in the constructor. By default, this method expects `(xmin, ymin, xmax, ymax)` in absolute image coordinates.
  - **labels**: `IntTensor` of shape `(num_boxes)` containing 0-indexed ground truth classes for the boxes.

As output of `forward` and `compute` the metric returns the following output:

- **ciou\_dict**: A dictionary containing the following key-values:
  - **ciou**: (`Tensor`)
  - **ciou/cl\_{cl}**: (`Tensor`), if argument `class_metrics=True`

#### Parameters

- **box\_format** (`str`) – Input format of given boxes. Supported formats are `[`xyxy`, `xywh`, `cxcywh`]`.
- **iou\_thresholds** – Optional IoU thresholds for evaluation. If set to `None` the threshold is ignored.
- **class\_metrics** (`bool`) – Option to enable per-class metrics for IoU. Has a performance impact.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> import torch
>>> from torchmetrics.detection import CompleteIntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.96,
↪ 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
>>> target = [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = CompleteIntersectionOverUnion()
>>> metric(preds, target)
{'ciou': tensor(-0.5694)}
```

#### Raises

**ModuleNotFoundError** – If torchvision is not installed with version 0.13.0 or newer.

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (**Union**[**Tensor**, **Sequence**[**Tensor**], **None**]) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (**Optional**[**Axes**]) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

**Tuple**[**Figure**, **Union**[**Axes**, **ndarray**]]

#### Returns

Figure object and Axes object

#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

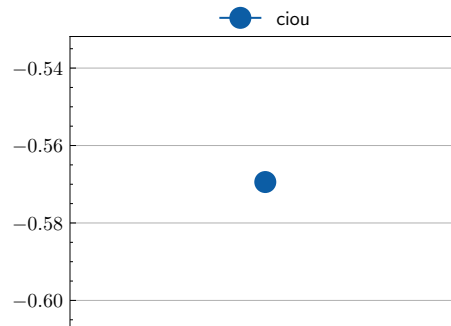
```
>>> # Example plotting single value
>>> import torch
>>> from torchmetrics.detection import CompleteIntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.
↪ 96, 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
```

(continues on next page)

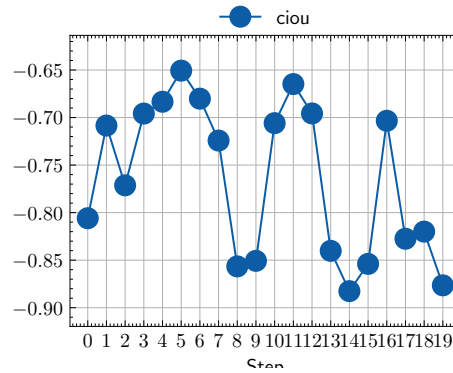


(continued from previous page)

```
>>> target = [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = CompleteIntersectionOverUnion()
>>> metric.update(preds, target)
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.detection import CompleteIntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.
... ↪96, 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
>>> target = lambda : [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]) + torch.
... ↪ randint(-10, 10, (1, 4)),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = CompleteIntersectionOverUnion()
>>> vals = []
>>> for _ in range(20):
...     vals.append(metric(preds, target()))
>>> fig_, ax_ = metric.plot(vals)
```



## 1.50.2 Functional Interface

`torchmetrics.functional.detection.ciou.complete_intersection_over_union`(*preds*, *target*,  
*iou\_threshold=None*,  
*replacement\_val=0*,  
*aggregate=True*)

Compute [Complete Intersection over Union](#) between two sets of boxes.

Both sets of boxes are expected to be in (x1, y1, x2, y2) format with  $0 \leq x1 < x2$  and  $0 \leq y1 < y2$ .

### Parameters

- **preds** ([Tensor](#)) – The input tensor containing the predicted bounding boxes.
- **target** ([Tensor](#)) – The tensor containing the ground truth.
- **iou\_threshold** ([Optional\[float\]](#)) – Optional IoU thresholds for evaluation. If set to *None* the threshold is ignored.
- **replacement\_val** ([float](#)) – Value to replace values under the threshold with.
- **aggregate** ([bool](#)) – Return the average value instead of the complete IoU matrix.

### Return type

[Tensor](#)

### Example

```
>>> import torch
>>> from torchmetrics.functional.detection import complete_intersection_over_union
>>> preds = torch.Tensor([[100, 100, 200, 200]])
>>> target = torch.Tensor([[110, 110, 210, 210]])
>>> complete_intersection_over_union(preds, target)
tensor(0.6724)
```

## 1.51 Distance Intersection Over Union (dIoU)

### 1.51.1 Module Interface

```
class torchmetrics.detection.diou.DistanceIntersectionOverUnion(box_format='xyxy',
                                                                iou_threshold=None,
                                                                class_metrics=False, **kwargs)
```

Computes Distance Intersection Over Union (DIOU) <<https://arxiv.org/abs/1911.08287v1>>`\_.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`List`): A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image). Parameters that should be provided per dict
  - **boxes**: (`FloatTensor`) of shape `(num_boxes, 4)` containing `num_boxes` detection boxes of the format specified in the constructor. By default, this method expects `(xmin, ymin, xmax, ymax)` in absolute image coordinates.
  - **scores**: `FloatTensor` of shape `(num_boxes)` containing detection scores for the boxes.
  - **labels**: `IntTensor` of shape `(num_boxes)` containing 0-indexed detection classes for the boxes.
- **target** (`List`): A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image). Parameters that should be provided per dict:
  - **boxes**: `FloatTensor` of shape `(num_boxes, 4)` containing `num_boxes` ground truth boxes of the format specified in the constructor. By default, this method expects `(xmin, ymin, xmax, ymax)` in absolute image coordinates.
  - **labels**: `IntTensor` of shape `(num_boxes)` containing 0-indexed ground truth classes for the boxes.

As output of `forward` and `compute` the metric returns the following output:

- **diou\_dict**: A dictionary containing the following key-values:
  - **diou**: (`Tensor`)
  - **diou/cl\_{cl}**: (`Tensor`), if argument `class_metrics=True`

#### Parameters

- **box\_format** (`str`) – Input format of given boxes. Supported formats are `['xyxy', 'xywh', 'cxcywh']`.
- **iou\_thresholds** – Optional IoU thresholds for evaluation. If set to `None` the threshold is ignored.
- **class\_metrics** (`bool`) – Option to enable per-class metrics for IoU. Has a performance impact.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> import torch
>>> from torchmetrics.detection import DistanceIntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.96,
↪ 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
>>> target = [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = DistanceIntersectionOverUnion()
>>> metric(preds, target)
{'diou': tensor(-0.0694)}
```

#### Raises

**ModuleNotFoundError** – If torchvision is not installed with version 0.13.0 or newer.

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (**Union**[**Tensor**, **Sequence**[**Tensor**], **None**]) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (**Optional**[**Axes**]) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

**Tuple**[**Figure**, **Union**[**Axes**, **ndarray**]]

#### Returns

Figure object and Axes object

#### Raises

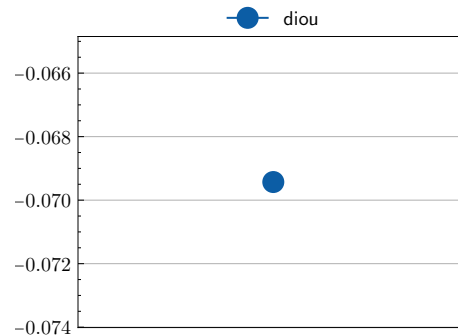
**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting single value
>>> import torch
>>> from torchmetrics.detection import DistanceIntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.
↪ 96, 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
```

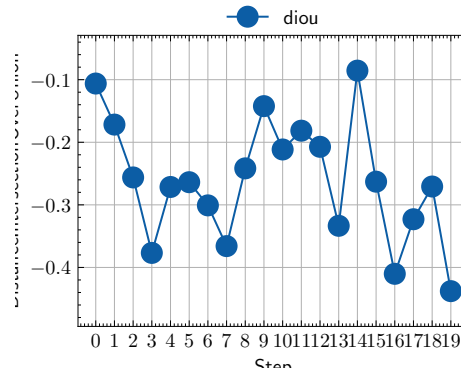
(continues on next page)

(continued from previous page)

```
>>> target = [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = DistanceIntersectionOverUnion()
>>> metric.update(preds, target)
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.detection import DistanceIntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.
... ↪96, 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
>>> target = lambda : [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]) + torch.
... ↪ randint(-10, 10, (1, 4)),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = DistanceIntersectionOverUnion()
>>> vals = []
>>> for _ in range(20):
...     vals.append(metric(preds, target()))
>>> fig_, ax_ = metric.plot(vals)
```



### 1.51.2 Functional Interface

`torchmetrics.functional.detection.dious.distance_intersection_over_union(preds, target, iou_threshold=None, replacement_val=0, aggregate=True)`

Compute [Distance Intersection over Union](#) between two sets of boxes.

Both sets of boxes are expected to be in (x1, y1, x2, y2) format with  $0 \leq x1 < x2$  and  $0 \leq y1 < y2$ .

#### Parameters

- **preds** ([Tensor](#)) – The input tensor containing the predicted bounding boxes.
- **target** ([Tensor](#)) – The tensor containing the ground truth.
- **iou\_threshold** ([Optional\[float\]](#)) – Optional IoU thresholds for evaluation. If set to *None* the threshold is ignored.
- **replacement\_val** ([float](#)) – Value to replace values under the threshold with.
- **aggregate** ([bool](#)) – Return the average value instead of the complete IoU matrix.

#### Return type

[Tensor](#)

#### Example

```
>>> import torch
>>> from torchmetrics.functional.detection import distance_intersection_over_union
>>> preds = torch.Tensor([[100, 100, 200, 200]])
>>> target = torch.Tensor([[110, 110, 210, 210]])
>>> distance_intersection_over_union(preds, target)
tensor(0.6724)
```

## 1.52 Generalized Intersection Over Union (giou)

### 1.52.1 Module Interface

```
class torchmetrics.detection.giou.GeneralizedIntersectionOverUnion(box_format='xyxy',
                                                                    iou_threshold=None,
                                                                    class_metrics=False,
                                                                    **kwargs)
```

Compute Generalized Intersection Over Union (GIoU) <<https://arxiv.org/abs/1902.09630>>`\_.

As input to forward and update the metric accepts the following input:

- **preds** (`List`): A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image). Parameters that should be provided per dict
  - **boxes**: (`FloatTensor`) of shape (`num_boxes`, 4) containing `num_boxes` detection boxes of the format specified in the constructor. By default, this method expects (`xmin`, `ymin`, `xmax`, `ymax`) in absolute image coordinates.
  - **scores**: `FloatTensor` of shape (`num_boxes`) containing detection scores for the boxes.
  - **labels**: `IntTensor` of shape (`num_boxes`) containing 0-indexed detection classes for the boxes.
- **target** (`List`): A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image). Parameters that should be provided per dict:
  - **boxes**: `FloatTensor` of shape (`num_boxes`, 4) containing `num_boxes` ground truth boxes of the format specified in the constructor. By default, this method expects (`xmin`, `ymin`, `xmax`, `ymax`) in absolute image coordinates.
  - **labels**: `IntTensor` of shape (`num_boxes`) containing 0-indexed ground truth classes for the boxes.

As output of forward and compute the metric returns the following output:

- **giou\_dict**: A dictionary containing the following key-values:
  - **giou**: (`Tensor`)
  - **giou/cl\_{cl}**: (`Tensor`), if argument `class_metrics=True`

#### Parameters

- **box\_format** (`str`) – Input format of given boxes. Supported formats are [``xyxy``, ``xywh``, ``cxcywh``].
- **iou\_thresholds** – Optional IoU thresholds for evaluation. If set to `None` the threshold is ignored.
- **class\_metrics** (`bool`) – Option to enable per-class metrics for IoU. Has a performance impact.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> import torch
>>> from torchmetrics.detection import GeneralizedIntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.96,
↪ 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
>>> target = [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = GeneralizedIntersectionOverUnion()
>>> metric(preds, target)
{'giou': tensor(-0.0694)}
```

#### Raises

**ModuleNotFoundError** – If torchvision is not installed with version 0.8.0 or newer.

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure object and Axes object

#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

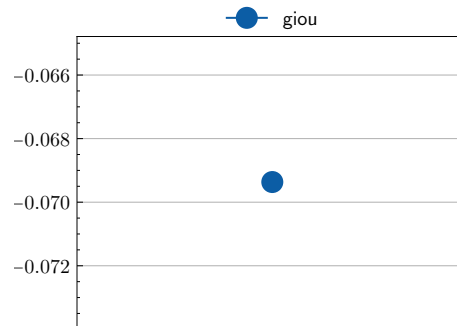
```
>>> # Example plotting single value
>>> import torch
>>> from torchmetrics.detection import GeneralizedIntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.
↪ 96, 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
```

(continues on next page)

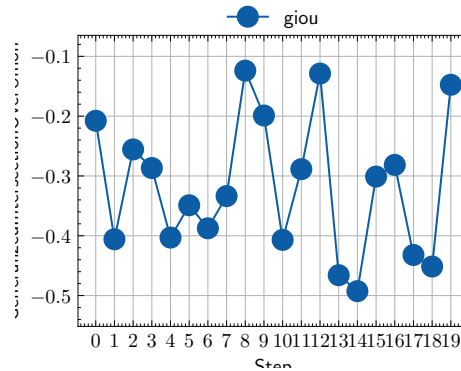


(continued from previous page)

```
>>> target = [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = GeneralizedIntersectionOverUnion()
>>> metric.update(preds, target)
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.detection import GeneralizedIntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.
... ↪96, 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
>>> target = lambda : [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]) + torch.
... ↪ randint(-10, 10, (1, 4)),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = GeneralizedIntersectionOverUnion()
>>> vals = []
>>> for _ in range(20):
...     vals.append(metric(preds, target()))
>>> fig_, ax_ = metric.plot(vals)
```



## 1.52.2 Functional Interface

`torchmetrics.functional.detection.giou.generalized_intersection_over_union(preds, target, iou_threshold=None, replacement_val=0, aggregate=True)`

Compute [Generalized Intersection over Union](#) between two sets of boxes.

Both sets of boxes are expected to be in (x1, y1, x2, y2) format with  $0 \leq x1 < x2$  and  $0 \leq y1 < y2$ .

### Parameters

- **preds** ([Tensor](#)) – The input tensor containing the predicted bounding boxes.
- **target** ([Tensor](#)) – The tensor containing the ground truth.
- **iou\_threshold** ([Optional\[float\]](#)) – Optional IoU thresholds for evaluation. If set to *None* the threshold is ignored.
- **replacement\_val** ([float](#)) – Value to replace values under the threshold with.
- **aggregate** ([bool](#)) – Return the average value instead of the complete IoU matrix.

### Return type

[Tensor](#)

### Example

```
>>> import torch
>>> from torchmetrics.functional.detection import generalized_intersection_over_
↪ union
>>> preds = torch.Tensor([[100, 100, 200, 200]])
>>> target = torch.Tensor([[110, 110, 210, 210]])
>>> generalized_intersection_over_union(preds, target)
tensor(0.6641)
```

## 1.53 Intersection Over Union (IoU)

### 1.53.1 Module Interface

```
class torchmetrics.detection.iou.IntersectionOverUnion(box_format='xyxy', iou_threshold=None,
                                                         class_metrics=False, respect_labels=True,
                                                         **kwargs)
```

Computes Intersection Over Union (IoU).

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`List`): A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image). Parameters that should be provided per dict
  - **boxes**: (`FloatTensor`) of shape (`num_boxes`, 4) containing `num_boxes` detection boxes of the format specified in the constructor. By default, this method expects (`xmin`, `ymin`, `xmax`, `ymax`) in absolute image coordinates.
  - **scores**: `FloatTensor` of shape (`num_boxes`) containing detection scores for the boxes.
  - **labels**: `IntTensor` of shape (`num_boxes`) containing 0-indexed detection classes for the boxes.
- **target** (`List`): A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image). Parameters that should be provided per dict:
  - **boxes**: `FloatTensor` of shape (`num_boxes`, 4) containing `num_boxes` ground truth boxes of the format specified in the constructor. By default, this method expects (`xmin`, `ymin`, `xmax`, `ymax`) in absolute image coordinates.
  - **labels**: `IntTensor` of shape (`num_boxes`) containing 0-indexed ground truth classes for the boxes.

As output of `forward` and `compute` the metric returns the following output:

- **iou\_dict**: A dictionary containing the following key-values:
  - **iou**: (`Tensor`)
  - **iou/cl\_{cl}**: (`Tensor`), if argument `class_metrics=True`

#### Parameters

- **box\_format** (`str`) – Input format of given boxes. Supported formats are [``xyxy``, ``xywh``, ``cxcywh``].
- **iou\_thresholds** – Optional IoU thresholds for evaluation. If set to `None` the threshold is ignored.
- **class\_metrics** (`bool`) – Option to enable per-class metrics for IoU. Has a performance impact.
- **respect\_labels** (`bool`) – Replace IoU values with the `invalid_val` if the labels do not match.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> import torch
>>> from torchmetrics.detection import IntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.96,
↪ 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
>>> target = [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = IntersectionOverUnion()
>>> metric(preds, target)
{'iou': tensor(0.4307)}
```

#### Raises

**ModuleNotFoundError** – If torchvision is not installed with version 0.8.0 or newer.

**plot**(*val=None*, *ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (**Union**[**Tensor**, **Sequence**[**Tensor**], **None**]) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (**Optional**[**Axes**]) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

**Tuple**[**Figure**, **Union**[**Axes**, **ndarray**]]

#### Returns

Figure object and Axes object

#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> import torch
>>> from torchmetrics.detection import IntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.
↪ 96, 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
>>> target = [
```

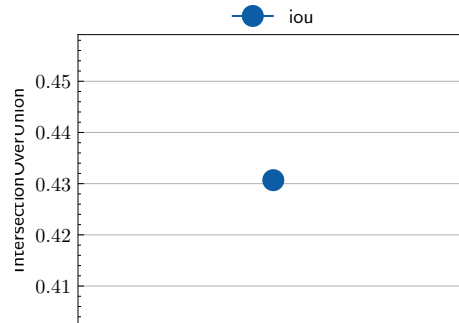
(continues on next page)

(continued from previous page)

```

...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = IntersectionOverUnion()
>>> metric.update(preds, target)
>>> fig_, ax_ = metric.plot()

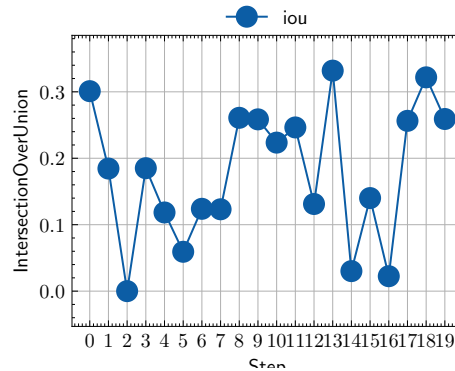
```



```

>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.detection import IntersectionOverUnion
>>> preds = [
...     {
...         "boxes": torch.tensor([[296.55, 93.96, 314.97, 152.79], [298.55, 98.
... ↪96, 314.97, 151.79]]),
...         "scores": torch.tensor([0.236, 0.56]),
...         "labels": torch.tensor([4, 5]),
...     }
... ]
>>> target = lambda : [
...     {
...         "boxes": torch.tensor([[300.00, 100.00, 315.00, 150.00]]) + torch.
... ↪randint(-10, 10, (1, 4)),
...         "labels": torch.tensor([5]),
...     }
... ]
>>> metric = IntersectionOverUnion()
>>> vals = []
>>> for _ in range(20):
...     vals.append(metric(preds, target()))
>>> fig_, ax_ = metric.plot(vals)

```



### 1.53.2 Functional Interface

`torchmetrics.functional.detection.iou.intersection_over_union(preds, target, iou_threshold=None, replacement_val=0, aggregate=True)`

Compute Intersection over Union between two sets of boxes.

Both sets of boxes are expected to be in (x1, y1, x2, y2) format with  $0 \leq x1 < x2$  and  $0 \leq y1 < y2$ .

#### Parameters

- **preds** (`Tensor`) – The input tensor containing the predicted bounding boxes.
- **target** (`Tensor`) – The tensor containing the ground truth.
- **iou\_threshold** (`Optional[float]`) – Optional IoU thresholds for evaluation. If set to `None` the threshold is ignored.
- **replacement\_val** (`float`) – Value to replace values under the threshold with.
- **aggregate** (`bool`) – Return the average value instead of the complete IoU matrix.

#### Return type

`Tensor`

#### Example

```
>>> import torch
>>> from torchmetrics.functional.detection import intersection_over_union
>>> preds = torch.Tensor([[100, 100, 200, 200]])
>>> target = torch.Tensor([[110, 110, 210, 210]])
>>> intersection_over_union(preds, target)
tensor(0.6807)
```

## 1.54 Mean-Average-Precision (mAP)

### 1.54.1 Module Interface

```
class torchmetrics.detection.mean_ap.MeanAveragePrecision(box_format='xyxy', iou_type='bbox',
                                                         iou_thresholds=None,
                                                         rec_thresholds=None,
                                                         max_detection_thresholds=None,
                                                         class_metrics=False, **kwargs)
```

Compute the [Mean-Average-Precision \(mAP\)](#) and [Mean-Average-Recall \(mAR\)](#) for object detection predictions.

$$\text{mAP} = \frac{1}{n} \sum_{i=1}^n AP_i$$

where  $AP_i$  is the average precision for class  $i$  and  $n$  is the number of classes. The average precision is defined as the area under the precision-recall curve. For object detection the recall and precision are defined based on the intersection of union (IoU) between the predicted bounding boxes and the ground truth bounding boxes e.g. if two boxes have an IoU  $> t$  (with  $t$  being some threshold) they are considered a match and therefore considered a true positive. The precision is then defined as the number of true positives divided by the number of all detected boxes and the recall is defined as the number of true positives divided by the number of all ground boxes.

As input to forward and update the metric accepts the following input:

- **preds** (List): A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image). Parameters that should be provided per dict
  - boxes: (FloatTensor) of shape (num\_boxes, 4) containing num\_boxes detection boxes of the format specified in the constructor. By default, this method expects (xmin, ymin, xmax, ymax) in absolute image coordinates.
  - scores: FloatTensor of shape (num\_boxes) containing detection scores for the boxes.
  - labels: IntTensor of shape (num\_boxes) containing 0-indexed detection classes for the boxes.
  - masks: bool of shape (num\_boxes, image\_height, image\_width) containing boolean masks. Only required when `iou_type="segm"`.
- **target** (List) A list consisting of dictionaries each containing the key-values (each dictionary corresponds to a single image). Parameters that should be provided per dict:
  - boxes: FloatTensor of shape (num\_boxes, 4) containing num\_boxes ground truth boxes of the format specified in the constructor. By default, this method expects (xmin, ymin, xmax, ymax) in absolute image coordinates.
  - labels: IntTensor of shape (num\_boxes) containing 0-indexed ground truth classes for the boxes.
  - masks: bool of shape (num\_boxes, image\_height, image\_width) containing boolean masks. Only required when `iou_type="segm"`.
  - iscrowd: IntTensor of shape (num\_boxes) containing 0/1 values indicating whether the bounding box/masks indicate a crowd of objects. Value is optional, and if not provided it will automatically be set to 0.
  - area: FloatTensor of shape (num\_boxes) containing the area of the object. Value if optional, and if not provided will be automatically calculated based on the bounding box/masks provided. Only affects which samples contribute to the `map_small`, `map_medium`, `map_large` values

As output of forward and compute the metric returns the following output:

- **map\_dict**: A dictionary containing the following key-values:

- `map`: ([Tensor](#)), global mean average precision
- `map_small`: ([Tensor](#)), mean average precision for small objects
- `map_medium`: ([Tensor](#)), mean average precision for medium objects
- `map_large`: ([Tensor](#)), mean average precision for large objects
- `mar_1`: ([Tensor](#)), mean average recall for 1 detection per image
- `mar_10`: ([Tensor](#)), mean average recall for 10 detections per image
- `mar_100`: ([Tensor](#)), mean average recall for 100 detections per image
- `mar_small`: ([Tensor](#)), mean average recall for small objects
- `mar_medium`: ([Tensor](#)), mean average recall for medium objects
- `mar_large`: ([Tensor](#)), mean average recall for large objects
- `map_50`: ([Tensor](#)) (-1 if 0.5 not in the list of iou thresholds), mean average precision at IoU=0.50
- `map_75`: ([Tensor](#)) (-1 if 0.75 not in the list of iou thresholds), mean average precision at IoU=0.75
- `map_per_class`: ([Tensor](#)) (-1 if class metrics are disabled), mean average precision per observed class
- `mar_100_per_class`: ([Tensor](#)) (-1 if class metrics are disabled), mean average recall for 100 detections per image per observed class
- `classes` ([Tensor](#)), list of all observed classes

For an example on how to use this metric check the [torchmetrics mAP example](#).

---

**Note:** `map` score is calculated with `@[ IoU=self.iou_thresholds | area=all | max_dets=max_detection_thresholds ]`. Caution: If the initialization parameters are changed, dictionary keys for mAR can change as well. The default properties are also accessible via fields and will raise an `AttributeError` if not available.

---

---

**Note:** This metric utilizes the official *pycocotools* implementation as its backend. This means that the metric requires you to have *pycocotools* installed. In addition we require *torchvision* version 0.8.0 or newer. Please install with `pip install torchmetrics[detection]`.

---

### Parameters

- **`box_format`** ([str](#)) – Input format of given boxes. Supported formats are [``xyxy``, ``xywh``, ``cxcywh``].
- **`iou_type`** ([Literal](#)[`'bbox'`, `'segm'`]) – Type of input (either masks or bounding-boxes) used for computing IOU. Supported IOU types are [`"bbox"`, `"segm"`]. If using `"segm"`, masks should be provided in input.
- **`iou_thresholds`** ([Optional](#)[[List](#)[[float](#)]]) – IoU thresholds for evaluation. If set to `None` it corresponds to the stepped range `[0.5, ..., 0.95]` with step `0.05`. Else provide a list of floats.
- **`rec_thresholds`** ([Optional](#)[[List](#)[[float](#)]]) – Recall thresholds for evaluation. If set to `None` it corresponds to the stepped range `[0, ..., 1]` with step `0.01`. Else provide a list of floats.
- **`max_detection_thresholds`** ([Optional](#)[[List](#)[[int](#)]]) – Thresholds on max detections per image. If set to `None` will use thresholds `[1, 10, 100]`. Else, please provide a list of ints.



- **class\_metrics** (`bool`) – Option to enable per-class metrics for mAP and mAR\_100. Has a performance impact.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **ModuleNotFoundError** – If pycocotools is not installed
- **ModuleNotFoundError** – If torchvision is not installed or version installed is lower than 0.8.0
- **ValueError** – If box\_format is not one of "xyxy", "xywh" or "cxcywh"
- **ValueError** – If iou\_type is not one of "bbox" or "segm"
- **ValueError** – If iou\_thresholds is not None or a list of floats
- **ValueError** – If rec\_thresholds is not None or a list of floats
- **ValueError** – If max\_detection\_thresholds is not None or a list of ints
- **ValueError** – If class\_metrics is not a boolean

#### Example

```
>>> from torch import tensor
>>> from torchmetrics.detection import MeanAveragePrecision
>>> preds = [
...     dict(
...         boxes=tensor([[258.0, 41.0, 606.0, 285.0]]),
...         scores=tensor([0.536]),
...         labels=tensor([0]),
...     )
... ]
>>> target = [
...     dict(
...         boxes=tensor([[214.0, 41.0, 562.0, 285.0]]),
...         labels=tensor([0]),
...     )
... ]
>>> metric = MeanAveragePrecision()
>>> metric.update(preds, target)
>>> from pprint import pprint
>>> pprint(metric.compute())
{'classes': tensor(0, dtype=torch.int32),
 'map': tensor(0.6000),
 'map_50': tensor(1.),
 'map_75': tensor(1.),
 'map_large': tensor(0.6000),
 'map_medium': tensor(-1.),
 'map_per_class': tensor(-1.),
 'map_small': tensor(-1.),
 'mar_1': tensor(0.6000),
 'mar_10': tensor(0.6000),
 'mar_100': tensor(0.6000),
 'mar_100_per_class': tensor(-1.),
```

(continues on next page)

(continued from previous page)

```
'mar_large': tensor(0.6000),
'mar_medium': tensor(-1.),
'mar_small': tensor(-1.)}
```

**static** `coco_to_tm(coco_preds, coco_target, iou_type='bbox')`

Utility function for converting .json coco format files to the input format of this metric.

The function accepts a file for the predictions and a file for the target in coco format and converts them to a list of dictionaries containing the boxes, labels and scores in the input format of this metric.

#### Parameters

- **coco\_preds** (`str`) – Path to the json file containing the predictions in coco format
- **coco\_target** (`str`) – Path to the json file containing the targets in coco format
- **iou\_type** (`Literal['bbox', 'segm']`) – Type of input, either *bbox* for bounding boxes or *segm* for segmentation masks

#### Returns

List of dictionaries containing the predictions in the input format of this metric target: List of dictionaries containing the targets in the input format of this metric

#### Return type

`preds`

### Example

```
>>> # File formats are defined at https://cocodataset.org/#format-data
>>> # Example files can be found at
>>> # https://github.com/cocodataset/cocoapi/tree/master/results
>>> from torchmetrics.detection import MeanAveragePrecision
>>> preds, target = MeanAveragePrecision.coco_to_tm(
...     "instances_val2014_fakebbox100_results.json.json",
...     "val2014_fake_eval_res.txt.json"
...     iou_type="bbox"
... )
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Dict[str, Tensor], Sequence[Dict[str, Tensor]], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure object and Axes object

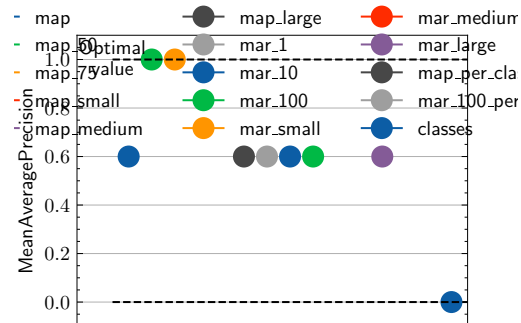
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```

>>> from torch import tensor
>>> from torchmetrics.detection.mean_ap import MeanAveragePrecision
>>> preds = [dict(
...     boxes=tensor([[258.0, 41.0, 606.0, 285.0]]),
...     scores=tensor([0.536]),
...     labels=tensor([0]),
... )]
>>> target = [dict(
...     boxes=tensor([[214.0, 41.0, 562.0, 285.0]]),
...     labels=tensor([0]),
... )]
>>> metric = MeanAveragePrecision()
>>> metric.update(preds, target)
>>> fig_, ax_ = metric.plot()

```



```

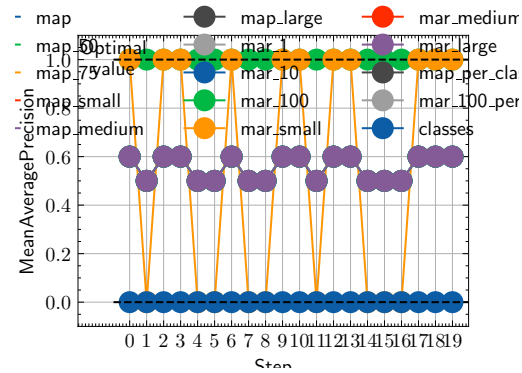
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.detection.mean_ap import MeanAveragePrecision
>>> preds = lambda: [dict(
...     boxes=torch.tensor([[258.0, 41.0, 606.0, 285.0]]) + torch.randint(10, (1,4)),
...     scores=torch.tensor([0.536]) + 0.1*torch.rand(1),
...     labels=torch.tensor([0]),
... )]
>>> target = [dict(
...     boxes=torch.tensor([[214.0, 41.0, 562.0, 285.0]]),
...     labels=torch.tensor([0]),
... )]
>>> metric = MeanAveragePrecision()
>>> vals = []
>>> for _ in range(20):
...     vals.append(metric(preds(), target))
>>> fig_, ax_ = metric.plot(vals)

```

`tm_to_coco(name='tm_map_input')`

Utility function for converting the input for this metric to coco format and saving it to a json file.

This function should be used after calling `.update(...)` or `.forward(...)` on all data that should be written to the file, as the input is then internally cached. The function then converts to information to coco format and writes it to json files.



## Parameters

**name (str)** – Name of the output file, which will be appended with “\_preds.json” and “\_target.json”

### Return type

None

### Example

```
>>> from torch import tensor
>>> from torchmetrics.detection import MeanAveragePrecision
>>> preds = [
...     dict(
...         boxes=tensor([[258.0, 41.0, 606.0, 285.0]]),
...         scores=tensor([0.536]),
...         labels=tensor([0]),
...     )
... ]
>>> target = [
...     dict(
...         boxes=tensor([[214.0, 41.0, 562.0, 285.0]]),
...         labels=tensor([0]),
...     )
... ]
>>> metric = MeanAveragePrecision()
>>> metric.update(preds, target)
>>> metric.tm_to_coco("tm_map_input")
```

### 1.55 Modified Panoptic Quality

### 1.55.1 Module Interface

`torchmetrics.ModifiedPanopticQuality`

alias of `_ModifiedPanopticQuality`

## 1.55.2 Functional Interface

`torchmetrics.functional.modified_panoptic_quality(preds, target, things, stuffs,`  
`allow_unknown_preds_category=False)`

Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([[[[0, 0], [0, 1], [6, 0], [7, 0], [0, 2], [1, 0]]]])
>>> target = tensor([[[[0, 1], [0, 0], [6, 0], [7, 0], [6, 0], [255, 0]]]])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _modified_panoptic_quality(preds, target, things = {0, 1}, stuffs = {6, 7})
tensor(0.7667, dtype=torch.float64)
```

## 1.56 Panoptic Quality

### 1.56.1 Module Interface

`torchmetrics.PanopticQuality`  
 alias of `_PanopticQuality`

### 1.56.2 Functional Interface

`torchmetrics.functional.panoptic_quality(preds, target, things, stuffs,`  
`allow_unknown_preds_category=False)`

Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([[[[6, 0], [0, 0], [6, 0], [6, 0]],
...                  [[0, 0], [0, 0], [6, 0], [0, 1]],
...                  [[0, 0], [0, 0], [6, 0], [0, 1]],
...                  [[0, 0], [7, 0], [6, 0], [1, 0]],
...                  [[0, 0], [7, 0], [7, 0], [7, 0]]]])
>>> target = tensor([[[[6, 0], [0, 1], [6, 0], [0, 1]],
...                  [[0, 1], [0, 1], [6, 0], [0, 1]],
...                  [[0, 1], [0, 1], [6, 0], [1, 0]],
...                  [[0, 1], [7, 0], [1, 0], [1, 0]],
...                  [[0, 1], [7, 0], [7, 0], [7, 0]]]])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _panoptic_quality(preds, target, things = {0, 1}, stuffs = {6, 7})
tensor(0.5463, dtype=torch.float64)
```

## 1.57 Error Relative Global Dim. Synthesis (ERGAS)

### 1.57.1 Module Interface

```
class torchmetrics.image.ergas.ErrorRelativeGlobalDimensionlessSynthesis(ratio=4, reduction='elementwise_mean', **kwargs)
```

Calculate [Relative dimensionless global error synthesis](#) (ERGAS).

This metric is used to calculate the accuracy of Pan sharpened image considering normalized average error of each band of the result image.

As input to `forward` and `update` the metric accepts the following input

- **preds** ([Tensor](#)): Predictions from model
- **target** ([Tensor](#)): Ground truth values

As output of `forward` and `compute` the metric returns the following output

- **ergas** ([Tensor](#)): if `reduction!='none'` returns float scalar tensor with average ERGAS value over sample else returns tensor of shape `(N,)` with ERGAS values per sample

#### Parameters

- **ratio** ([Union](#)[[int](#), [float](#)]) – ratio of high resolution to low resolution
- **reduction** ([Literal](#)['[elementwise\\_mean](#)', '[sum](#)', '[none](#)', `None`]) – a method to reduce metric score over labels.
  - '[elementwise\\_mean](#)': takes the mean (default)
  - '[sum](#)': takes the sum
  - '[none](#)' or `None`: no reduction will be applied
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> import torch
>>> from torchmetrics.image import ErrorRelativeGlobalDimensionlessSynthesis
>>> preds = torch.rand([16, 1, 16, 16], generator=torch.manual_seed(42))
>>> target = preds * 0.75
>>> ergas = ErrorRelativeGlobalDimensionlessSynthesis()
>>> torch.round(ergas(preds, target))
tensor(154.)
```

```
plot(val=None, ax=None)
```

Plot a single or multiple values from the metric.

#### Parameters

- **val** ([Union](#)[[Tensor](#), [Sequence](#)[[Tensor](#)], `None`]) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** ([Optional](#)[[Axes](#)]) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

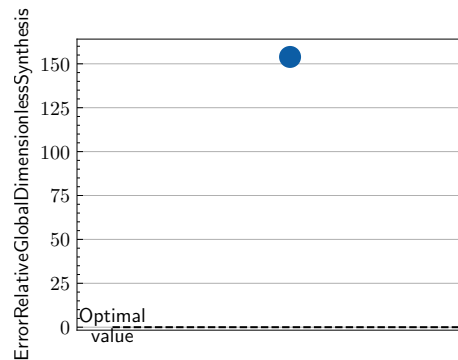
**Returns**

Figure and Axes object

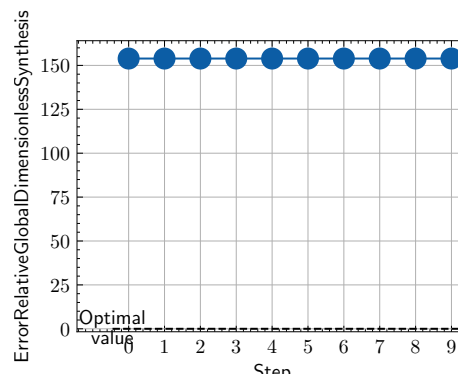
**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.image import ErrorRelativeGlobalDimensionlessSynthesis
>>> preds = torch.rand([16, 1, 16, 16], generator=torch.manual_seed(42))
>>> target = preds * 0.75
>>> metric = ErrorRelativeGlobalDimensionlessSynthesis()
>>> metric.update(preds, target)
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.image import ErrorRelativeGlobalDimensionlessSynthesis
>>> preds = torch.rand([16, 1, 16, 16], generator=torch.manual_seed(42))
>>> target = preds * 0.75
>>> metric = ErrorRelativeGlobalDimensionlessSynthesis()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(preds, target))
>>> fig_, ax_ = metric.plot(values)
```



## 1.57.2 Functional Interface

`torchmetrics.functional.error_relative_global_dimensionless_synthesis(preds, target, ratio=4, reduction='elementwise_mean')`

Wrapper for deprecated import.

```
>>> import torch
>>> preds = torch.rand([16, 1, 16, 16], generator=torch.manual_seed(42))
>>> target = preds * 0.75
>>> ergds = _error_relative_global_dimensionless_synthesis(preds, target)
:rtype: :py:class:`~torch.Tensor`
```

```
>>> torch.round(ergds)
tensor(154.)
```

## 1.58 Frechet Inception Distance (FID)

### 1.58.1 Module Interface

`class torchmetrics.image.fid.FrechetInceptionDistance(feature=2048, reset_real_features=True, normalize=False, **kwargs)`

Calculate Fréchet inception distance ([FID](#)) which is used to access the quality of generated images.

$$FID = \|\mu - \mu_w\|^2 + \text{tr}(\Sigma + \Sigma_w - 2(\Sigma\Sigma_w)^{\frac{1}{2}})$$

where  $\mathcal{N}(\mu, \Sigma)$  is the multivariate normal distribution estimated from Inception v3 ([fid ref1](#)) features calculated on real life images and  $\mathcal{N}(\mu_w, \Sigma_w)$  is the multivariate normal distribution estimated from Inception v3 features calculated on generated (fake) images. The metric was originally proposed in [fid ref1](#).

Using the default feature extraction (Inception v3 using the original weights from [fid ref2](#)), the input is expected to be mini-batches of 3-channel RGB images of shape (3 x H x W). If argument `normalize` is `True` images are expected to be dtype `float` and have values in the `[0, 1]` range, else if `normalize` is set to `False` images are expected to have dtype `uint8` and take values in the `[0, 255]` range. All images will be resized to 299 x 299 which is the size of the original training data. The boolean flag `real` determines if the images should update the statistics of the real distribution or the fake distribution.

This metric is known to be unstable in its calculations, and we recommend for the best results using this metric that you calculate using `torch.float64` (default is `torch.float32`) which can be set using the `.set_dtype` method of the metric.

---

**Note:** using this metrics requires you to have torch 1.9 or higher installed

---

---

**Note:** using this metric with the default feature extractor requires that `torch-fidelity` is installed. Either install as `pip install torchmetrics[image]` or `pip install torch-fidelity`

---

As input to `forward` and `update` the metric accepts the following input

- `imgs` ([Tensor](#)): tensor with images feed to the feature extractor with



- **real** (`bool`): `bool` indicating if `imgs` belong to the real or the fake distribution

As output of *forward* and *compute* the metric returns the following output

- **fid** (`Tensor`): float scalar tensor with mean FID value over samples

#### Parameters

- **feature** (`Union[int, Module]`) – Either an integer or `nn.Module`:
  - an integer will indicate the inceptionv3 feature layer to choose. Can be one of the following: 64, 192, 768, 2048
  - an `nn.Module` for using a custom feature extractor. Expects that its forward method returns an  $(N, d)$  matrix where  $N$  is the batch size and  $d$  is the feature size.
- **reset\_real\_features** (`bool`) – Whether to also reset the real features. Since in many cases the real dataset does not change, the features can be cached them to avoid recomputing them which is costly. Set this to `False` if your dataset does not change.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

#### Raises

- **ValueError** – If torch version is lower than 1.9
- **ModuleNotFoundError** – If `feature` is set to an `int` (default settings) and `torch-fidelity` is not installed
- **ValueError** – If `feature` is set to an `int` not in [64, 192, 768, 2048]
- **TypeError** – If `feature` is not an `str`, `int` or `torch.nn.Module`
- **ValueError** – If `reset_real_features` is not an `bool`

#### Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.fid import FrechetInceptionDistance
>>> fid = FrechetInceptionDistance(feature=64)
>>> # generate two slightly overlapping image intensity distributions
>>> imgs_dist1 = torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.uint8)
>>> imgs_dist2 = torch.randint(100, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> fid.update(imgs_dist1, real=True)
>>> fid.update(imgs_dist2, real=False)
>>> fid.compute()
tensor(12.7202)
```

**plot** (`val=None, ax=None`)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

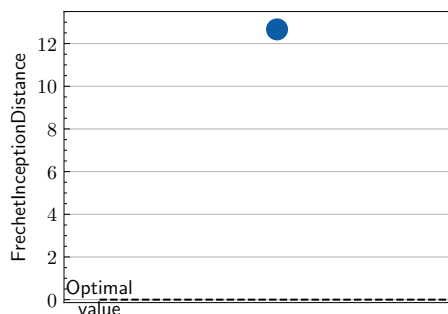
**Returns**

Figure and Axes object

**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.image.fid import FrechetInceptionDistance
>>> imgs_dist1 = torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.uint8)
>>> imgs_dist2 = torch.randint(100, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> metric = FrechetInceptionDistance(feature=64)
>>> metric.update(imgs_dist1, real=True)
>>> metric.update(imgs_dist2, real=False)
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.image.fid import FrechetInceptionDistance
>>> imgs_dist1 = lambda: torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.
↳ uint8)
>>> imgs_dist2 = lambda: torch.randint(100, 255, (100, 3, 299, 299),
↳ dtype=torch.uint8)
>>> metric = FrechetInceptionDistance(feature=64)
>>> values = [ ]
>>> for _ in range(3):
...     metric.update(imgs_dist1(), real=True)
...     metric.update(imgs_dist2(), real=False)
...     values.append(metric.compute())
...     metric.reset()
>>> fig_, ax_ = metric.plot(values)
```

**reset()**

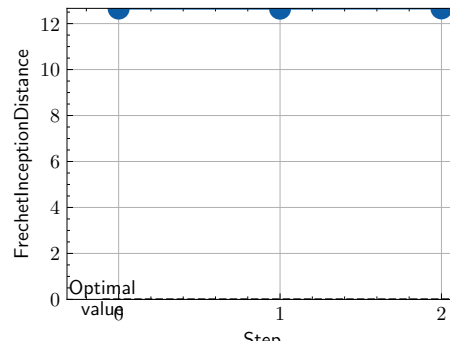
Reset metric states.

**Return type**

`None`

**set\_dtype(dst\_type)**

Transfer all metric state to specific dtype. Special version of standard *type* method.

**Parameters**

**dst\_type** (*type* or *string*) – the desired type.

**Return type**

*Metric*

## 1.59 Image Gradients

### 1.59.1 Functional Interface

`torchmetrics.functional.image_gradients(img)`

Wrapper for deprecated import.

```
>>> import torch
>>> image = torch.arange(0, 1*1*5*5, dtype=torch.float32)
>>> image = torch.reshape(image, (1, 1, 5, 5))
>>> dy, dx = _image_gradients(image)
>>> dy[0, 0, :, :]
tensor([[5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        :rtype: :py:data:`~typing.Tuple`[:py:class:`~torch.Tensor`, :py:class:`~torch.
↪Tensor`]
```

```
[5., 5., 5., 5., 5.], [0., 0., 0., 0., 0.]])
```

## 1.60 Inception Score

### 1.60.1 Module Interface

**class** `torchmetrics.image.inception.InceptionScore`(*feature='logits\_unbiased', splits=10, normalize=False, \*\*kwargs*)

Calculate the Inception Score (IS) which is used to access how realistic generated images are.

$$IS = \exp(\mathbb{E}_x KL(p(y|x)||p(y)))$$

where  $KL(p(y|x)||p(y))$  is the KL divergence between the conditional distribution  $p(y|x)$  and the marginal distribution  $p(y)$ . Both the conditional and marginal distribution is calculated from features extracted from the images. The score is calculated on random splits of the images such that both a mean and standard deviation of the score are returned. The metric was originally proposed in [inception ref1](#).

Using the default feature extraction (Inception v3 using the original weights from [inception ref2](#)), the input is expected to be mini-batches of 3-channel RGB images of shape (3 x H x W). If argument `normalize` is `True` images are expected to be dtype `float` and have values in the `[0, 1]` range, else if `normalize` is set to `False` images are expected to have dtype `uint8` and take values in the `[0, 255]` range. All images will be resized to 299 x 299 which is the size of the original training data.

---

**Note:** using this metric with the default feature extractor requires that `torch-fidelity` is installed. Either install as `pip install torchmetrics[image]` or `pip install torch-fidelity`

---

As input to `forward` and `update` the metric accepts the following input

- `imgs` (`Tensor`): tensor with images feed to the feature extractor

As output of `forward` and `compute` the metric returns the following output

- `fid` (`Tensor`): float scalar tensor with mean FID value over samples

#### Parameters

- **feature** (`Union[str, int, Module]`) – Either an str, integer or `nn.Module`:
  - an str or integer will indicate the inceptionv3 feature layer to choose. Can be one of the following: 'logits\_unbiased', 64, 192, 768, 2048
  - an `nn.Module` for using a custom feature extractor. Expects that its forward method returns an (N, d) matrix where N is the batch size and d is the feature size.
- **splits** (`int`) – integer determining how many splits the inception score calculation should be split among
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **ValueError** – If feature is set to an str or int and `torch-fidelity` is not installed
- **ValueError** – If feature is set to an str or int and not one of ('logits\_unbiased', 64, 192, 768, 2048)
- **TypeError** – If feature is not an str, int or `torch.nn.Module`

#### Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.inception import InceptionScore
>>> inception = InceptionScore()
>>> # generate some images
>>> imgs = torch.randint(0, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> inception.update(imgs)
>>> inception.compute()
(tensor(1.0544), tensor(0.0117))
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

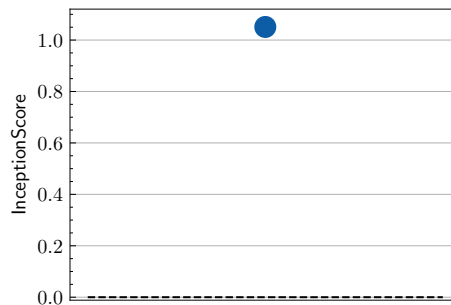
#### Returns

Figure and Axes object

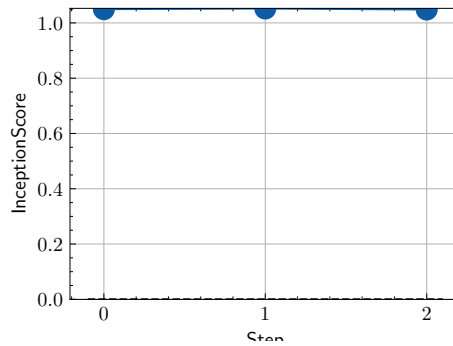
#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.image.inception import InceptionScore
>>> metric = InceptionScore()
>>> metric.update(torch.randint(0, 255, (50, 3, 299, 299), dtype=torch.uint8))
>>> fig_, ax_ = metric.plot() # the returned plot only shows the mean value by_
↳ default
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.image.inception import InceptionScore
>>> metric = InceptionScore()
>>> values = [ ]
>>> for _ in range(3):
...     # we index by 0 such that only the mean value is plotted
...     values.append(metric(torch.randint(0, 255, (50, 3, 299, 299),
↳ dtype=torch.uint8))[0])
>>> fig_, ax_ = metric.plot(values)
```



## 1.61 Kernel Inception Distance

### 1.61.1 Module Interface

```
class torchmetrics.image.kid.KernelInceptionDistance(feature=2048, subsets=100, subset_size=1000,  
                                                    degree=3, gamma=None, coef=1.0,  
                                                    reset_real_features=True, normalize=False,  
                                                    **kwargs)
```

Calculate Kernel Inception Distance (KID) which is used to access the quality of generated images.

$$KID = MMD(f_{real}, f_{fake})^2$$

where  $MMD$  is the maximum mean discrepancy and  $I_{real}, I_{fake}$  are extracted features from real and fake images, see [kid ref1](#) for more details. In particular, calculating the MMD requires the evaluation of a polynomial kernel function  $k$

$$k(x, y) = (\gamma * x^T y + coef)^{degree}$$

which controls the distance between two features. In practise the MMD is calculated over a number of subsets to be able to both get the mean and standard deviation of KID.

Using the default feature extraction (Inception v3 using the original weights from [kid ref2](#)), the input is expected to be mini-batches of 3-channel RGB images of shape (3 x H x W). If argument `normalize` is `True` images are expected to be dtype `float` and have values in the `[0, 1]` range, else if `normalize` is set to `False` images are expected to have dtype `uint8` and take values in the `[0, 255]` range. All images will be resized to 299 x 299 which is the size of the original training data. The boolean flag `real` determines if the images should update the statistics of the real distribution or the fake distribution.

---

**Note:** using this metric with the default feature extractor requires that `torch-fidelity` is installed. Either install as `pip install torchmetrics[image]` or `pip install torch-fidelity`

---

As input to `forward` and `update` the metric accepts the following input

- `imgs` ([Tensor](#)): tensor with images feed to the feature extractor of shape (N,C,H,W)
- `real` (`bool`): bool indicating if `imgs` belong to the real or the fake distribution

As output of `forward` and `compute` the metric returns the following output

- `kid_mean` ([Tensor](#)): float scalar tensor with mean value over subsets
- `kid_std` ([Tensor](#)): float scalar tensor with mean value over subsets

### Parameters

- **feature** (`Union[str, int, Module]`) – Either an str, integer or `nn.Module`:
  - an str or integer will indicate the inceptionv3 feature layer to choose. Can be one of the following: ‘logits\_unbiased’, 64, 192, 768, 2048
  - an `nn.Module` for using a custom feature extractor. Expects that its forward method returns an (N,d) matrix where N is the batch size and d is the feature size.
- **subsets** (`int`) – Number of subsets to calculate the mean and standard deviation scores over
- **subset\_size** (`int`) – Number of randomly picked samples in each subset
- **degree** (`int`) – Degree of the polynomial kernel function
- **gamma** (`Optional[float]`) – Scale-length of polynomial kernel. If set to None will be automatically set to the feature size
- **coef** (`float`) – Bias term in the polynomial kernel.
- **reset\_real\_features** (`bool`) – Whether to also reset the real features. Since in many cases the real dataset does not change, the features can be cached to avoid recomputing them which is costly. Set this to False if your dataset does not change.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

### Raises

- **ValueError** – If feature is set to an int (default settings) and torch-fidelity is not installed
- **ValueError** – If feature is set to an int not in (64, 192, 768, 2048)
- **ValueError** – If subsets is not an integer larger than 0
- **ValueError** – If subset\_size is not an integer larger than 0
- **ValueError** – If degree is not an integer larger than 0
- **ValueError** – If gamma is neither None or a float larger than 0
- **ValueError** – If coef is not a float larger than 0
- **ValueError** – If reset\_real\_features is not a bool

### Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.kid import KernelInceptionDistance
>>> kid = KernelInceptionDistance(subset_size=50)
>>> # generate two slightly overlapping image intensity distributions
>>> imgs_dist1 = torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.uint8)
>>> imgs_dist2 = torch.randint(100, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> kid.update(imgs_dist1, real=True)
>>> kid.update(imgs_dist2, real=False)
>>> kid_mean, kid_std = kid.compute()
>>> print((kid_mean, kid_std))
(tensor(0.0337), tensor(0.0023))
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

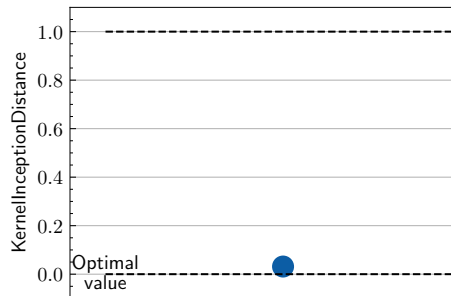
#### Returns

Figure and Axes object

#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.image.kid import KernelInceptionDistance
>>> imgs_dist1 = torch.randint(0, 200, (30, 3, 299, 299), dtype=torch.uint8)
>>> imgs_dist2 = torch.randint(100, 255, (30, 3, 299, 299), dtype=torch.uint8)
>>> metric = KernelInceptionDistance(subsets=3, subset_size=20)
>>> metric.update(imgs_dist1, real=True)
>>> metric.update(imgs_dist2, real=False)
>>> fig_, ax_ = metric.plot()
```



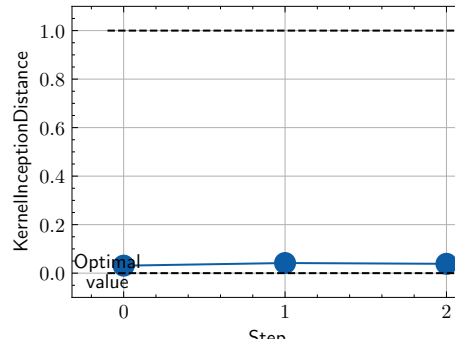
```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.image.kid import KernelInceptionDistance
>>> imgs_dist1 = lambda: torch.randint(0, 200, (30, 3, 299, 299), dtype=torch.
↳ uint8)
>>> imgs_dist2 = lambda: torch.randint(100, 255, (30, 3, 299, 299), dtype=torch.
↳ uint8)
>>> metric = KernelInceptionDistance(subsets=3, subset_size=20)
>>> values = [ ]
>>> for _ in range(3):
...     metric.update(imgs_dist1(), real=True)
...     metric.update(imgs_dist2(), real=False)
...     values.append(metric.compute()[0])
```

(continues on next page)



(continued from previous page)

```
...     metric.reset()
>>> fig_, ax_ = metric.plot(values)
```

**reset()**

Reset metric states.

**Return type**

None

## 1.62 Learned Perceptual Image Patch Similarity (LPIPS)

### 1.62.1 Module Interface

```
class torchmetrics.image.lpip.LearnedPerceptualImagePatchSimilarity(net_type='alex',  
                                                                    reduction='mean',  
                                                                    normalize=False,  
                                                                    **kwargs)
```

The Learned Perceptual Image Patch Similarity (*LPIPS*) calculates the perceptual similarity between two images.

LPIPS essentially computes the similarity between the activations of two image patches for some pre-defined network. This measure has been shown to match human perception well. A low LPIPS score means that image patches are perceptual similar.

Both input image patches are expected to have shape  $(N, 3, H, W)$ . The minimum size of  $H, W$  depends on the chosen backbone (see *net\_type* arg).

---

**Note:** using this metrics requires you to have `lpips` package installed. Either install as `pip install torchmetrics[image]` or `pip install lpips`

---



---

**Note:** this metric is not scriptable when using `torch<1.8`. Please update your pytorch installation if this is a issue.

---

As input to forward and update the metric accepts the following input

- `img1` (`Tensor`): tensor with images of shape  $(N, 3, H, W)$

- `img2` (`Tensor`): tensor with images of shape (N, 3, H, W)

As output of *forward* and *compute* the metric returns the following output

- `lpips` (`Tensor`): returns float scalar tensor with average LPIPS value over samples

#### Parameters

- **`net_type`** (`Literal`['vgg', 'alex', 'squeeze']) – str indicating backbone network type to use. Choose between 'alex', 'vgg' or 'squeeze'
- **`reduction`** (`Literal`['sum', 'mean']) – str indicating how to reduce over the batch dimension. Choose between 'sum' or 'mean'.
- **`normalize`** (`bool`) – by default this is False meaning that the input is expected to be in the [-1,1] range. If set to True will instead expect input to be in the [0,1] range.
- **`kwargs`** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **`ModuleNotFoundError`** – If lpips package is not installed
- **`ValueError`** – If `net_type` is not one of "vgg", "alex" or "squeeze"
- **`ValueError`** – If `reduction` is not one of "mean" or "sum"

#### Example

```
>>> import torch
>>> _ = torch.manual_seed(123)
>>> from torchmetrics.image.lpip import LearnedPerceptualImagePatchSimilarity
>>> lpips = LearnedPerceptualImagePatchSimilarity(net_type='squeeze')
>>> # LPIPS needs the images to be in the [-1, 1] range.
>>> img1 = (torch.rand(10, 3, 100, 100) * 2) - 1
>>> img2 = (torch.rand(10, 3, 100, 100) * 2) - 1
>>> lpips(img1, img2)
tensor(0.1046, grad_fn=<SqueezeBackward0>)
```

**`plot`**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **`val`** (`Union`[`Tensor`, `Sequence`[`Tensor`], `None`]) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **`ax`** (`Optional`[`Axes`]) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple`[`Figure`, `Union`[`Axes`, `ndarray`]]

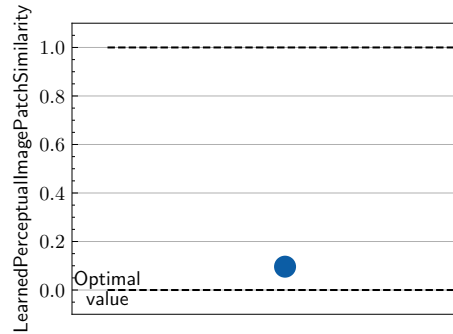
#### Returns

Figure and Axes object

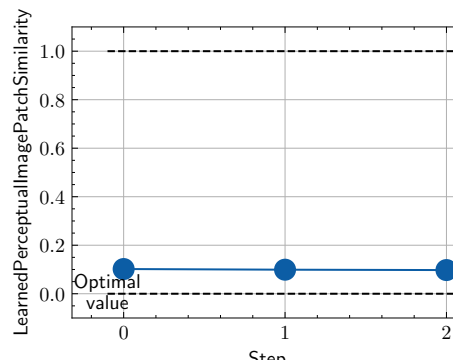
#### Raises

**`ModuleNotFoundError`** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.image.lpip import LearnedPerceptualImagePatchSimilarity
>>> metric = LearnedPerceptualImagePatchSimilarity(net_type='squeeze')
>>> metric.update(torch.rand(10, 3, 100, 100), torch.rand(10, 3, 100, 100))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.image.lpip import LearnedPerceptualImagePatchSimilarity
>>> metric = LearnedPerceptualImagePatchSimilarity(net_type='squeeze')
>>> values = [ ]
>>> for _ in range(3):
...     values.append(metric(torch.rand(10, 3, 100, 100), torch.rand(10, 3, 100,
... ↪ 100)))
>>> fig_, ax_ = metric.plot(values)
```



## 1.63 Memorization-Informed Frechet Inception Distance (MiFID)

### 1.63.1 Module Interface

```
class torchmetrics.image.mifid.MemorizationInformedFrechetInceptionDistance(feature=2048, re-  
set_real_features=True,  
normal-  
ize=False,  
co-  
sine_distance_eps=0.1,  
**kwargs)
```

Calculate Memorization-Informed Frechet Inception Distance (**MIFID**).

MIFID is a improved variation of the Frechet Inception Distance (**FID**) that penalizes memorization of the training set by the generator. It is calculated as

$$MIFID = \frac{FID(F_{real}, F_{fake})}{M(F_{real}, F_{fake})}$$

where  $FID$  is the normal FID score and  $M$  is the memorization penalty. The memorization penalty essentially corresponds to the average minimum cosine distance between the features of the real and fake distribution.

Using the default feature extraction (Inception v3 using the original weights from [fid ref2](#)), the input is expected to be mini-batches of 3-channel RGB images of shape (3 x H x W). If argument `normalize` is `True` images are expected to be dtype `float` and have values in the `[0, 1]` range, else if `normalize` is set to `False` images are expected to have dtype `uint8` and take values in the `[0, 255]` range. All images will be resized to 299 x 299 which is the size of the original training data. The boolean flag `real` determines if the images should update the statistics of the real distribution or the fake distribution.

---

**Note:** using this metrics requires you to have `scipy` install. Either install as `pip install torchmetrics[image]` or `pip install scipy`

---

---

**Note:** using this metric with the default feature extractor requires that `torch-fidelity` is installed. Either install as `pip install torchmetrics[image]` or `pip install torch-fidelity`

---

As input to `forward` and `update` the metric accepts the following input

- `imgs` (**Tensor**): tensor with images feed to the feature extractor with
- `real` (**bool**): bool indicating if `imgs` belong to the real or the fake distribution

As output of `forward` and `compute` the metric returns the following output

- `mifid` (**Tensor**): float scalar tensor with mean MIFID value over samples

#### Parameters

- **feature** (**Union[int, Module]**) – Either an integer or `nn.Module`:
  - an integer will indicate the inceptionv3 feature layer to choose. Can be one of the following: 64, 192, 768, 2048
  - an `nn.Module` for using a custom feature extractor. Expects that its forward method returns an (N, d) matrix where N is the batch size and d is the feature size.

- **reset\_real\_features** (`bool`) – Whether to also reset the real features. Since in many cases the real dataset does not change, the features can be cached them to avoid recomputing them which is costly. Set this to `False` if your dataset does not change.
- **cosine\_distance\_eps** (`float`) – Epsilon value for the cosine distance. If the cosine distance is larger than this value it is set to 1 and thus ignored in the MIFID calculation.
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Raises**

- **RuntimeError** – If `torch` is version less than 1.10
- **ValueError** – If `feature` is set to an `int` and `torch-fidelity` is not installed
- **ValueError** – If `feature` is set to an `int` not in `[64, 192, 768, 2048]`
- **TypeError** – If `feature` is not an `str`, `int` or `torch.nn.Module`
- **ValueError** – If `reset_real_features` is not an `bool`

**Example::**

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> from torchmetrics.image.mifid import_
↳ MemorizationInformedFrechetInceptionDistance
>>> mifid = MemorizationInformedFrechetInceptionDistance(feature=64)
>>> # generate two slightly overlapping image intensity distributions
>>> imgs_dist1 = torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.uint8)
>>> imgs_dist2 = torch.randint(100, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> mifid.update(imgs_dist1, real=True)
>>> mifid.update(imgs_dist2, real=False)
>>> mifid.compute()
tensor(3003.3691)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

**Parameters**

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An `matplotlib` axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

**Returns**

Figure and Axes object

**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.image.mifid import_
↳ MemorizationInformedFrechetInceptionDistance
```

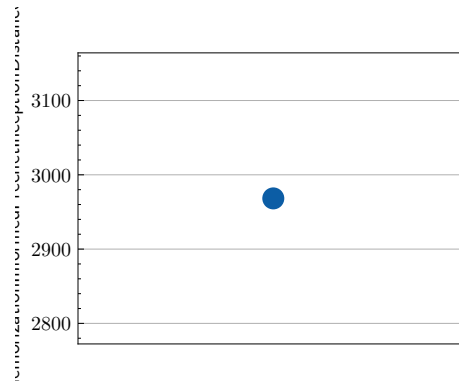
(continues on next page)

(continued from previous page)

```

>>> imgs_dist1 = torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.uint8)
>>> imgs_dist2 = torch.randint(100, 255, (100, 3, 299, 299), dtype=torch.uint8)
>>> metric = MemorizationInformedFrechetInceptionDistance(feature=64)
>>> metric.update(imgs_dist1, real=True)
>>> metric.update(imgs_dist2, real=False)
>>> fig_, ax_ = metric.plot()

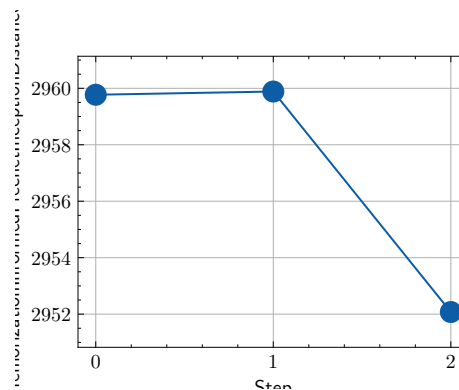
```



```

>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.image.mifid import _
>>> _MemorizationInformedFrechetInceptionDistance
>>> imgs_dist1 = lambda: torch.randint(0, 200, (100, 3, 299, 299), dtype=torch.
>>> _uint8)
>>> imgs_dist2 = lambda: torch.randint(100, 255, (100, 3, 299, 299), _
>>> _dtype=torch.uint8)
>>> metric = MemorizationInformedFrechetInceptionDistance(feature=64)
>>> values = [ ]
>>> for _ in range(3):
...     metric.update(imgs_dist1(), real=True)
...     metric.update(imgs_dist2(), real=False)
...     values.append(metric.compute())
...     metric.reset()
>>> fig_, ax_ = metric.plot(values)

```

**reset()**

Reset metric states.

**Return type**

None

## 1.64 Multi-Scale SSIM

### 1.64.1 Module Interface

`torchmetrics.MultiScaleStructuralSimilarityIndexMeasure`alias of `_MultiScaleStructuralSimilarityIndexMeasure`

### 1.64.2 Functional Interface

```
torchmetrics.functional.multiscale_structural_similarity_index_measure(preds, target,
                                                                       gaussian_kernel=True,
                                                                       sigma=1.5,
                                                                       kernel_size=11, reduction='elementwise_mean',
                                                                       data_range=None,
                                                                       k1=0.01, k2=0.03,
                                                                       betas=(0.0448, 0.2856,
                                                                       0.3001, 0.2363,
                                                                       0.1333),
                                                                       normalize='relu')
```

Wrapper for deprecated import.

```
>>> import torch
>>> preds = torch.rand([3, 3, 256, 256], generator=torch.manual_seed(42))
>>> target = preds * 0.75
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _multiscale_structural_similarity_index_measure(preds, target, data_range=1.0)
tensor(0.9627)
```

## 1.65 Peak Signal-to-Noise Ratio (PSNR)

### 1.65.1 Module Interface

`torchmetrics.PeakSignalNoiseRatio`alias of `_PeakSignalNoiseRatio`

## 1.65.2 Functional Interface

`torchmetrics.functional.peak_signal_noise_ratio(preds, target, data_range=None, base=10.0, reduction='elementwise_mean', dim=None)`

Wrapper for deprecated import.

```
>>> from torch import tensor
>>> pred = tensor([[0.0, 1.0], [2.0, 3.0]])
>>> target = tensor([[3.0, 2.0], [1.0, 0.0]])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _peak_signal_noise_ratio(pred, target)
tensor(2.5527)
```

## 1.66 Peak Signal To Noise Ratio With Blocked Effect

### 1.66.1 Module Interface

`class torchmetrics.image.PeakSignalNoiseRatioWithBlockedEffect(block_size=8, **kwargs)`

Computes [Peak Signal to Noise Ratio With Blocked Effect](#) (PSNRB).

$$\text{PSNRB}(I, J) = 10 * \log_{10} \left( \frac{\max(I)^2}{\text{MSE}(I, J) - B(I, J)} \right)$$

Where MSE denotes the [mean-squared-error](#) function. This metric is a modified version of PSNR that better supports evaluation of images with blocked artifacts, that oftens occur in compressed images.

---

**Note:** Metric only supports grayscale images. If you have RGB images, please convert them to grayscale first.

---

As input to `forward` and `update` the metric accepts the following input

- `preds` ([Tensor](#)): Predictions from model of shape  $(N, 1, H, W)$
- `target` ([Tensor](#)): Ground truth values of shape  $(N, 1, H, W)$

As output of `forward` and `compute` the metric returns the following output

- `psnr` ([Tensor](#)): float scalar tensor with aggregated PSNRB value

#### Parameters

- `block_size` ([int](#)) – integer indication the block size
- `kwargs` ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.



## Example

```
>>> import torch
>>> from torchmetrics.image import PeakSignalNoiseRatioWithBlockedEffect
>>> metric = PeakSignalNoiseRatioWithBlockedEffect()
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(2, 1, 10, 10)
>>> target = torch.rand(2, 1, 10, 10)
>>> metric(preds, target)
tensor(7.2893)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

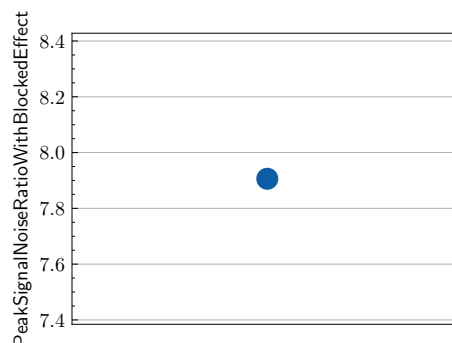
### Returns

Figure and Axes object

### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.image import PeakSignalNoiseRatioWithBlockedEffect
>>> metric = PeakSignalNoiseRatioWithBlockedEffect()
>>> metric.update(torch.rand(2, 1, 10, 10), torch.rand(2, 1, 10, 10))
>>> fig_, ax_ = metric.plot()
```

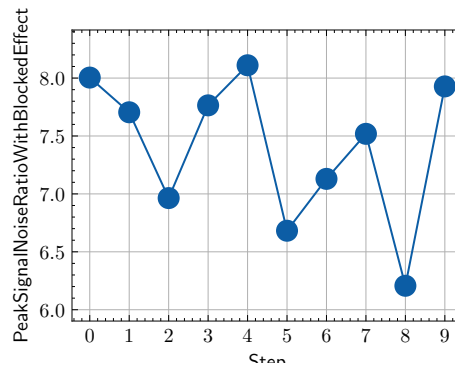


```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.image import PeakSignalNoiseRatioWithBlockedEffect
>>> metric = PeakSignalNoiseRatioWithBlockedEffect()
```

(continues on next page)

(continued from previous page)

```
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.rand(2, 1, 10, 10), torch.rand(2, 1, 10, 10),
...                             ↪10)))
>>> fig_, ax_ = metric.plot(values)
```



## 1.66.2 Functional Interface

`torchmetrics.functional.image.peak_signal_noise_ratio_with_blocked_effect(preds, target, block_size=8)`

Computes *Peak Signal to Noise Ratio With Blocked Effect* (PSNRB) metrics.

$$\text{PSNRB}(I, J) = 10 * \log_{10} \left( \frac{\max(I)^2}{\text{MSE}(I, J) - B(I, J)} \right)$$

Where MSE denotes the [mean-squared-error](#) function.

### Parameters

- **preds** ([Tensor](#)) – estimated signal
- **target** ([Tensor](#)) – ground truth signal
- **block\_size** ([int](#)) – integer indication the block size

### Return type

[Tensor](#)

### Returns

Tensor with PSNRB score

### Example

```
>>> import torch
>>> from torchmetrics.functional.image import peak_signal_noise_ratio_with_blocked_
...     ↪effect
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand(1, 1, 28, 28)
>>> target = torch.rand(1, 1, 28, 28)
```

(continues on next page)

(continued from previous page)

```
>>> peak_signal_noise_ratio_with_blocked_effect(preds, target)
tensor(7.8402)
```

## 1.67 Relative Average Spectral Error (RASE)

### 1.67.1 Module Interface

`torchmetrics.RelativeAverageSpectralError`  
alias of `_RelativeAverageSpectralError`

### 1.67.2 Functional Interface

`torchmetrics.functional.relative_average_spectral_error(preds, target, window_size=8)`  
Wrapper for deprecated import.

```
>>> import torch
>>> g = torch.manual_seed(22)
>>> preds = torch.rand(4, 3, 16, 16)
>>> target = torch.rand(4, 3, 16, 16)
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _relative_average_spectral_error(preds, target)
tensor(5114.6641)
```

## 1.68 Root Mean Squared Error Using Sliding Window

### 1.68.1 Module Interface

`torchmetrics.RootMeanSquaredErrorUsingSlidingWindow`  
alias of `_RootMeanSquaredErrorUsingSlidingWindow`

### 1.68.2 Functional Interface

`torchmetrics.functional.root_mean_squared_error_using_sliding_window(preds, target, window_size=8, return_rmse_map=False)`  
Wrapper for deprecated import.

```
>>> import torch
>>> g = torch.manual_seed(22)
>>> preds = torch.rand(4, 3, 16, 16)
>>> target = torch.rand(4, 3, 16, 16)
:rtype: :py:data:`~typing.Union`[:py:class:`~torch.Tensor`, :py:obj:`~None`, :py:
↪data:`~typing.Tuple`[:py:data:`~typing.Optional`[:py:class:`~torch.Tensor`], :
↪py:class:`~torch.Tensor`]]
```

```
>>> _root_mean_squared_error_using_sliding_window(preds, target)
tensor(0.3999)
```

## 1.69 Spectral Angle Mapper

### 1.69.1 Module Interface

`torchmetrics.SpectralAngleMapper`  
alias of `_SpectralAngleMapper`

### 1.69.2 Functional Interface

`torchmetrics.functional.spectral_angle_mapper(preds, target, reduction='elementwise_mean')`  
Wrapper for deprecated import.

```
>>> import torch
>>> preds = torch.rand([16, 3, 16, 16], generator=torch.manual_seed(42))
>>> target = torch.rand([16, 3, 16, 16], generator=torch.manual_seed(123))
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _spectral_angle_mapper(preds, target)
tensor(0.5943)
```

## 1.70 Spectral Distortion Index

### 1.70.1 Module Interface

`torchmetrics.SpectralDistortionIndex`  
alias of `_SpectralDistortionIndex`

### 1.70.2 Functional Interface

`torchmetrics.functional.spectral_distortion_index(preds, target, p=1, reduction='elementwise_mean')`  
Wrapper for deprecated import.

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> preds = torch.rand([16, 3, 16, 16])
>>> target = torch.rand([16, 3, 16, 16])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _spectral_distortion_index(preds, target)
tensor(0.0234)
```

## 1.71 Structural Similarity Index Measure (SSIM)

### 1.71.1 Module Interface

`torchmetrics.StructuralSimilarityIndexMeasure`  
alias of `_StructuralSimilarityIndexMeasure`

### 1.71.2 Functional Interface

`torchmetrics.functional.structural_similarity_index_measure(preds, target, gaussian_kernel=True, sigma=1.5, kernel_size=11, reduction='elementwise_mean', data_range=None, k1=0.01, k2=0.03, return_full_image=False, return_contrast_sensitivity=False)`

Wrapper for deprecated import.

```
>>> import torch
>>> preds = torch.rand([3, 3, 256, 256])
>>> target = preds * 0.75
:rtype: :py:data:`~typing.Union\[:py:class:`~torch.Tensor`, :py:data:`~typing.
↳ Tuple\[:py:class:`~torch.Tensor`, :py:class:`~torch.Tensor`]
```

```
>>> _structural_similarity_index_measure(preds, target)
tensor(0.9219)
```

## 1.72 Total Variation (TV)

### 1.72.1 Module Interface

`torchmetrics.TotalVariation`  
alias of `_TotalVariation`

### 1.72.2 Functional Interface

`torchmetrics.functional.total_variation(img, reduction='sum')`

Wrapper for deprecated import.

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> img = torch.rand(5, 3, 28, 28)
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _total_variation(img)
tensor(7546.8018)
```

## 1.73 Universal Image Quality Index

### 1.73.1 Module Interface

`torchmetrics.UniversalImageQualityIndex`  
alias of `_UniversalImageQualityIndex`

### 1.73.2 Functional Interface

`torchmetrics.functional.universal_image_quality_index(preds, target, kernel_size=(11, 11),  
sigma=(1.5, 1.5),  
reduction='elementwise_mean')`

Wrapper for deprecated import.

```
>>> import torch
>>> preds = torch.rand([16, 1, 16, 16])
>>> target = preds * 0.75
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _universal_image_quality_index(preds, target)
tensor(0.9216)
```

## 1.74 CLIP Score

### 1.74.1 Module Interface

`class torchmetrics.multimodal.clip_score.CLIPScore(model_name_or_path='openai/clip-vit-large-patch14', **kwargs)`

Calculates *CLIP Score* which is a text-to-image similarity metric.

CLIP is a reference free metric that can be used to evaluate the correlation between a generated caption for an image and the actual content of the image. It has been found to be highly correlated with human judgement. The metric is defined as:

$$\text{CLIPScore}(I, C) = \max(100 * \cos(E_I, E_C), 0)$$

which corresponds to the cosine similarity between visual CLIP embedding  $E_i$  for an image  $i$  and textual CLIP embedding  $E_C$  for an caption  $C$ . The score is bound between 0 and 100 and the closer to 100 the better.

---

**Note:** Metric is not scriptable

---

#### Parameters

- **model\_name\_or\_path** (`Literal['openai/clip-vit-base-patch16', 'openai/clip-vit-base-patch32', 'openai/clip-vit-large-patch14-336', 'openai/clip-vit-large-patch14']`) – string indicating the version of the CLIP model to use. Available models are:
  - `"openai/clip-vit-base-patch16"`

- "openai/clip-vit-base-patch32"
- "openai/clip-vit-large-patch14-336"
- "openai/clip-vit-large-patch14"
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises**

**ModuleNotFoundError** – If transformers package is not installed or version is lower than 4.10.0

**Example**

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> from torchmetrics.multimodal import CLIPScore
>>> metric = CLIPScore(model_name_or_path="openai/clip-vit-base-patch16")
>>> score = metric(torch.randint(255, (3, 224, 224)), "a photo of a cat")
>>> print(score.detach())
tensor(24.7691)
```

**compute()**

Compute accumulated clip score.

**Return type**

*Tensor*

**plot(val=None, ax=None)**

Plot a single or multiple values from the metric.

**Parameters**

- **val** (*Union[Tensor, Sequence[Tensor], None]*) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (*Optional[Axes]*) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

*Tuple[Figure, Union[Axes, ndarray]]*

**Returns**

Figure and Axes object

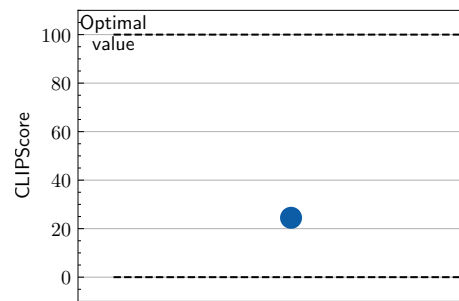
**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.multimodal import CLIPScore
>>> metric = CLIPScore(model_name_or_path="openai/clip-vit-base-patch16")
>>> metric.update(torch.randint(255, (3, 224, 224)), "a photo of a cat")
>>> fig_, ax_ = metric.plot()
```

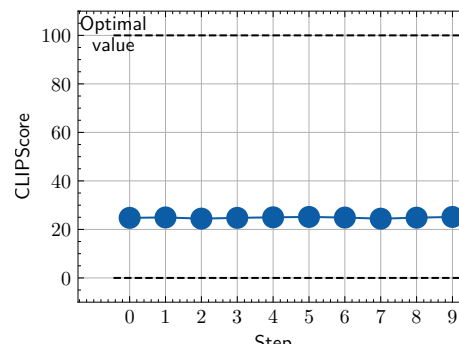
```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.multimodal import CLIPScore
>>> metric = CLIPScore(model_name_or_path="openai/clip-vit-base-patch16")
```

(continues on next page)



(continued from previous page)

```
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.randint(255, (3, 224, 224)), "a photo of a
↳ cat"))
>>> fig_, ax_ = metric.plot(values)
```

**update(images, text)**

Update CLIP score on a batch of images and text.

**Parameters**

- **images** (`Union[Tensor, List\[Tensor\]]`) – Either a single [N, C, H, W] tensor or a list of [C, H, W] tensors
- **text** (`Union[str, List\[str\]]`) – Either a single caption or a list of captions

**Raises**

- **[ValueError](#)** – If not all images have format [C, H, W]
- **[ValueError](#)** – If the number of images and captions do not match

**Return type**`None`



## 1.74.2 Functional Interface

```
torchmetrics.functional.multimodal.clip_score.clip_score(images, text,
                                                         model_name_or_path='openai/clip-vit-
                                                         large-patch14')
```

Calculate *CLIP Score* which is a text-to-image similarity metric.

CLIP is a reference free metric that can be used to evaluate the correlation between a generated caption for an image and the actual content of the image. It has been found to be highly correlated with human judgement. The metric is defined as:

$$\text{CLIPScore}(I, C) = \max(100 * \cos(E_I, E_C), 0)$$

which corresponds to the cosine similarity between visual CLIP embedding  $E_i$  for an image  $i$  and textual CLIP embedding  $E_C$  for a caption  $C$ . The score is bound between 0 and 100 and the closer to 100 the better.

---

**Note:** Metric is not scriptable

---

### Parameters

- **images** (`Union[Tensor, List\[Tensor\]]`) – Either a single [N, C, H, W] tensor or a list of [C, H, W] tensors
- **text** (`Union[str, List\[str\]]`) – Either a single caption or a list of captions
- **model\_name\_or\_path** (`Literal['openai/clip-vit-base-patch16', 'openai/clip-vit-base-patch32', 'openai/clip-vit-large-patch14-336', 'openai/clip-vit-large-patch14']`) – string indicating the version of the CLIP model to use. Available models are “openai/clip-vit-base-patch16”, “openai/clip-vit-base-patch32”, “openai/clip-vit-large-patch14-336” and “openai/clip-vit-large-patch14”,

### Raises

- **[ModuleNotFoundError](#)** – If transformers package is not installed or version is lower than 4.10.0
- **[ValueError](#)** – If not all images have format [C, H, W]
- **[ValueError](#)** – If the number of images and captions do not match

### Return type

[Tensor](#)

### Example

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> from torchmetrics.functional.multimodal import clip_score
>>> score = clip_score(torch.randint(255, (3, 224, 224)), "a photo of a cat",
↪ "openai/clip-vit-base-patch16")
>>> print(score.detach())
tensor(24.4255)
```

## 1.75 Cramer's V

### 1.75.1 Module Interface

**class** torchmetrics.CramersV(*num\_classes*, *bias\_correction=True*, *nan\_strategy='replace'*,  
*nan\_replace\_value=0.0*, *\*\*kwargs*)

Compute *Cramer's V* statistic measuring the association between two categorical (nominal) data series.

$$V = \sqrt{\frac{\chi^2/n}{\min(r-1, k-1)}}$$

where

$$\chi^2 = \sum_{i,j} \text{frac}\left(n_{ij} - \frac{n_{i.}n_{.j}}{n}\right)^2 \frac{n_{i.}n_{.j}}{n}$$

where  $n_{ij}$  denotes the number of times the values  $(A_i, B_j)$  are observed with  $A_i, B_j$  represent frequencies of values in `preds` and `target`, respectively.

Cramer's V is a symmetric coefficient, i.e.  $V(\text{preds}, \text{target}) = V(\text{target}, \text{preds})$ .

The output values lies in  $[0, 1]$  with 1 meaning the perfect association.

#### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **bias\_correction** (`bool`) – Indication of whether to use bias correction.
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN's when `nan_strategy = 'replace'`
- **kwargs** (`Any`) – Additional keyword arguments, see *Advanced metric settings* for more info.

#### Returns

Cramer's V statistic

#### Raises

- **ValueError** – If `nan_strategy` is not one of `'replace'` and `'drop'`
- **ValueError** – If `nan_strategy` is equal to `'replace'` and `nan_replace_value` is not an `int` or `float`

#### Example

```
>>> from torchmetrics.nominal import CramersV
>>> _ = torch.manual_seed(42)
>>> preds = torch.randint(0, 4, (100,))
>>> target = torch.round(preds + torch.randn(100)).clamp(0, 4)
>>> cramers_v = CramersV(num_classes=5)
>>> cramers_v(preds, target)
tensor(0.5284)
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

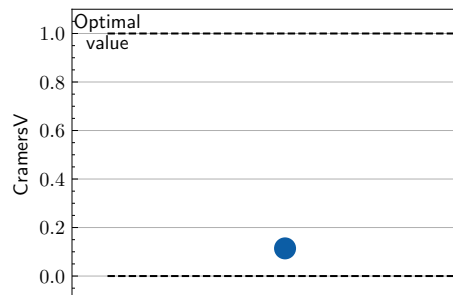
#### Returns

Figure and Axes object

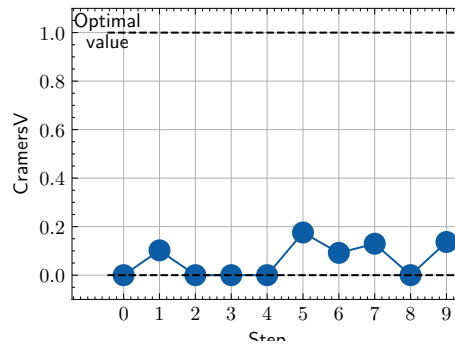
#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.nominal import CramersV
>>> metric = CramersV(num_classes=5)
>>> metric.update(torch.randint(0, 4, (100,)), torch.randint(0, 4, (100,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.nominal import CramersV
>>> metric = CramersV(num_classes=5)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.randint(0, 4, (100,)), torch.randint(0, 4, (100,))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.75.2 Functional Interface

`torchmetrics.functional.cramers_v(preds, target, bias_correction=True, nan_strategy='replace', nan_replace_value=0.0)`

Compute *Cramer's V* statistic measuring the association between two categorical (nominal) data series.

$$V = \sqrt{\frac{\chi^2/n}{\min(r-1, k-1)}}$$

where

$$\chi^2 = \sum_{i,j} \text{frac}\left(n_{ij} - \frac{n_{i.}n_{.j}}{n}\right)^2 \frac{n_{i.}n_{.j}}{n}$$

where  $n_{ij}$  denotes the number of times the values  $(A_i, B_j)$  are observed with  $A_i, B_j$  represent frequencies of values in `preds` and `target`, respectively.

Cramer's V is a symmetric coefficient, i.e.  $V(\text{preds}, \text{target}) = V(\text{target}, \text{preds})$ .

The output values lies in  $[0, 1]$  with 1 meaning the perfect association.

#### Parameters

- **preds** (`Tensor`) – 1D or 2D tensor of categorical (nominal) data - 1D shape: (batch\_size,) - 2D shape: (batch\_size, num\_classes)
- **target** (`Tensor`) – 1D or 2D tensor of categorical (nominal) data - 1D shape: (batch\_size,) - 2D shape: (batch\_size, num\_classes)
- **bias\_correction** (`bool`) – Indication of whether to use bias correction.
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN's when `nan_strategy = 'replace'`

#### Return type

`Tensor`

#### Returns

Cramer's V statistic

### Example

```
>>> from torchmetrics.functional.nominal import cramers_v
>>> _ = torch.manual_seed(42)
>>> preds = torch.randint(0, 4, (100,))
>>> target = torch.round(preds + torch.randn(100)).clamp(0, 4)
>>> cramers_v(preds, target)
tensor(0.5284)
```

### cramers\_v\_matrix

`torchmetrics.functional.nominal.cramers_v_matrix(matrix, bias_correction=True, nan_strategy='replace', nan_replace_value=0.0)`

Compute *Cramer's V* statistic between a set of multiple variables.

This can serve as a convenient tool to compute Cramer's V statistic for analyses of correlation between categorical variables in your dataset.

#### Parameters

- **matrix** (`Tensor`) – A tensor of categorical (nominal) data, where: - rows represent a number of data points - columns represent a number of categorical (nominal) features
- **bias\_correction** (`bool`) – Indication of whether to use bias correction.
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN's when ``nan\_strategy = 'replace'`

#### Return type

`Tensor`

#### Returns

Cramer's V statistic for a dataset of categorical variables

### Example

```
>>> from torchmetrics.functional.nominal import cramers_v_matrix
>>> _ = torch.manual_seed(42)
>>> matrix = torch.randint(0, 4, (200, 5))
>>> cramers_v_matrix(matrix)
tensor([[1.0000, 0.0637, 0.0000, 0.0542, 0.1337],
        [0.0637, 1.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 1.0000, 0.0000, 0.0649],
        [0.0542, 0.0000, 0.0000, 1.0000, 0.1100],
        [0.1337, 0.0000, 0.0649, 0.1100, 1.0000]])
```

## 1.76 Fleiss Kappa

### 1.76.1 Module Interface

**class** torchmetrics.FleissKappa(mode='counts', \*\*kwargs)

Calculates *Fleiss kappa* a statistical measure for inter agreement between raters.

$$\kappa = \frac{\bar{p} - \bar{p}_e}{1 - \bar{p}_e}$$

where  $\bar{p}$  is the mean of the agreement probability over all raters and  $\bar{p}_e$  is the mean agreement probability over all raters if they were randomly assigned. If the raters are in complete agreement then the score 1 is returned, if there is no agreement among the raters (other than what would be expected by chance) then a score smaller than 0 is returned.

As input to forward and update the metric accepts the following input:

- **ratings** (**Tensor**): Ratings of shape `[n_samples, n_categories]` or `[n_samples, n_categories, n_raters]` dependent on mode. If mode is counts, ratings must be integer and contain the number of raters that chose each category. If mode is probs, ratings must be floating point and contain the probability/logits that each rater chose each category.

As output of forward and compute the metric returns the following output:

- **fleiss\_k** (**Tensor**): A float scalar tensor with the calculated Fleiss' kappa score.

#### Parameters

- **mode** (**Literal**['counts', 'probs']) – Whether *ratings* will be provided as counts or probabilities.
- **kwargs** (**Any**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> # Ratings are provided as counts
>>> import torch
>>> from torchmetrics.nominal import FleissKappa
>>> _ = torch.manual_seed(42)
>>> ratings = torch.randint(0, 10, size=(100, 5)).long() # 100 samples, 5
↳ categories, 10 raters
>>> metric = FleissKappa(mode='counts')
>>> metric(ratings)
tensor(0.0089)
```

#### Example

```
>>> # Ratings are provided as probabilities
>>> import torch
>>> from torchmetrics.nominal import FleissKappa
>>> _ = torch.manual_seed(42)
>>> ratings = torch.randn(100, 5, 10).softmax(dim=1) # 100 samples, 5 categories,
↳ 10 raters
```

(continues on next page)

(continued from previous page)

```
>>> metric = FleissKappa(mode='probs')
>>> metric(ratings)
tensor(-0.0105)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

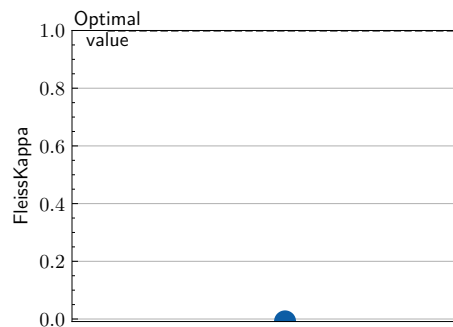
#### Returns

Figure and Axes object

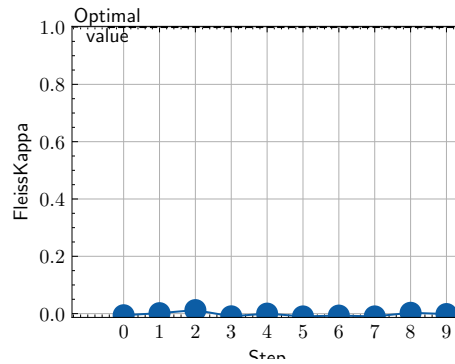
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.nominal import FleissKappa
>>> metric = FleissKappa(mode="probs")
>>> metric.update(torch.randn(100, 5, 10).softmax(dim=1))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.nominal import FleissKappa
>>> metric = FleissKappa(mode="probs")
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.randn(100, 5, 10).softmax(dim=1)))
>>> fig_, ax_ = metric.plot(values)
```



## 1.76.2 Functional Interface

`torchmetrics.functional.fleiss_kappa(ratings, mode='counts')`

Calculates *Fleiss kappa* a statistical measure for inter agreement between raters.

$$\kappa = \frac{\bar{p} - \bar{p}_e}{1 - \bar{p}_e}$$

where  $\bar{p}$  is the mean of the agreement probability over all raters and  $\bar{p}_e$  is the mean agreement probability over all raters if they were randomly assigned. If the raters are in complete agreement then the score 1 is returned, if there is no agreement among the raters (other than what would be expected by chance) then a score smaller than 0 is returned.

### Parameters

- **ratings** (`Tensor`) – Ratings of shape `[n_samples, n_categories]` or `[n_samples, n_categories, n_raters]` dependent on *mode*. If *mode* is *counts*, *ratings* must be integer and contain the number of raters that chose each category. If *mode* is *probs*, *ratings* must be floating point and contain the probability/logits that each rater chose each category.
- **mode** (`Literal['counts', 'probs']`) – Whether *ratings* will be provided as counts or probabilities.

### Return type

`Tensor`

### Example

```
>>> # Ratings are provided as counts
>>> import torch
>>> from torchmetrics.functional.nominal import fleiss_kappa
>>> _ = torch.manual_seed(42)
>>> ratings = torch.randint(0, 10, size=(100, 5)).long() # 100 samples, 5
↳ categories, 10 raters
>>> fleiss_kappa(ratings)
tensor(0.0089)
```



### Example

```
>>> # Ratings are provided as probabilities
>>> import torch
>>> from torchmetrics.functional.nominal import fleiss_kappa
>>> _ = torch.manual_seed(42)
>>> ratings = torch.randn(100, 5, 10).softmax(dim=1) # 100 samples, 5 categories, 10 raters
>>> fleiss_kappa(ratings, mode='probs')
tensor(-0.0105)
```

## 1.77 Pearson's Contingency Coefficient

### 1.77.1 Module Interface

**class** torchmetrics.PearsonsContingencyCoefficient(*num\_classes*, *nan\_strategy*='replace', *nan\_replace\_value*=0.0, *\*\*kwargs*)

Compute *Pearson's Contingency Coefficient* statistic.

This metric measures the association between two categorical (nominal) data series.

$$Pearson = \sqrt{\frac{\chi^2/n}{1 + \chi^2/n}}$$

where

$$\chi^2 = \sum_{i,j} \frac{\left(n_{ij} - \frac{n_{i.}n_{.j}}{n}\right)^2}{\frac{n_{i.}n_{.j}}{n}}$$

where  $n_{ij}$  denotes the number of times the values  $(A_i, B_j)$  are observed with  $A_i, B_j$  represent frequencies of values in preds and target, respectively.

Pearson's Contingency Coefficient is a symmetric coefficient, i.e.  $Pearson(preds, target) = Pearson(target, preds)$ .

The output values lies in [0, 1] with 1 meaning the perfect association.

#### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN's when `nan_strategy = 'replace'`
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Returns

Pearson's Contingency Coefficient statistic

#### Raises

- **ValueError** – If `nan_strategy` is not one of 'replace' and 'drop'
- **ValueError** – If `nan_strategy` is equal to 'replace' and `nan_replace_value` is not an `int` or `float`

## Example

```
>>> from torchmetrics.nominal import PearsonsContingencyCoefficient
>>> _ = torch.manual_seed(42)
>>> preds = torch.randint(0, 4, (100,))
>>> target = torch.round(preds + torch.randn(100)).clamp(0, 4)
>>> pearsons_contingency_coefficient = PearsonsContingencyCoefficient(num_classes=5)
>>> pearsons_contingency_coefficient(preds, target)
tensor(0.6948)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

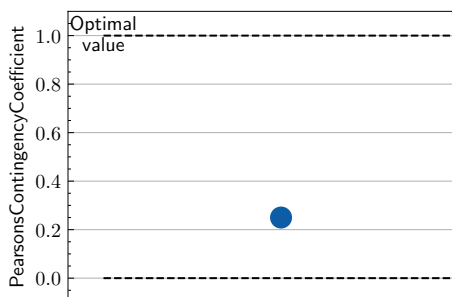
### Returns

Figure and Axes object

### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.nominal import PearsonsContingencyCoefficient
>>> metric = PearsonsContingencyCoefficient(num_classes=5)
>>> metric.update(torch.randint(0, 4, (100,)), torch.randint(0, 4, (100,)))
>>> fig_, ax_ = metric.plot()
```

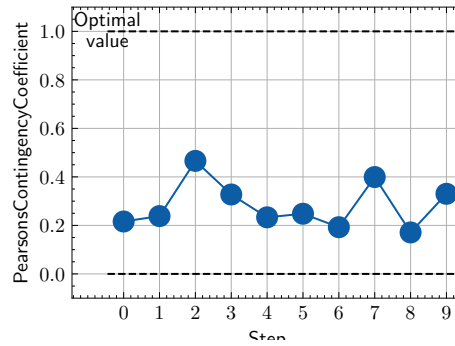


```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.nominal import PearsonsContingencyCoefficient
>>> metric = PearsonsContingencyCoefficient(num_classes=5)
>>> values = [ ]
```

(continues on next page)

(continued from previous page)

```
>>> for _ in range(10):
...     values.append(metric(torch.randint(0, 4, (100,)), torch.randint(0, 4, (100,))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.77.2 Functional Interface

`torchmetrics.functional.pearsons_contingency_coefficient(preds, target, nan_strategy='replace', nan_replace_value=0.0)`

Compute *Pearson's Contingency Coefficient* for measuring the association between two categorical data series.

$$Pearson = \sqrt{\frac{\chi^2/n}{1 + \chi^2/n}}$$

where

$$\chi^2 = \sum_{i,j} \frac{(n_{ij} - \frac{n_{i.}n_{.j}}{n})^2}{\frac{n_{i.}n_{.j}}{n}}$$

where  $n_{ij}$  denotes the number of times the values  $(A_i, B_j)$  are observed with  $A_i, B_j$  represent frequencies of values in `preds` and `target`, respectively.

Pearson's Contingency Coefficient is a symmetric coefficient, i.e.  $Pearson(preds, target) = Pearson(target, preds)$ .

The output values lies in  $[0, 1]$  with 1 meaning the perfect association.

#### Parameters

- **preds** (`Tensor`) – 1D or 2D tensor of categorical (nominal) data:
  - 1D shape: (batch\_size,)
  - 2D shape: (batch\_size, num\_classes)
- **target** (`Tensor`) – 1D or 2D tensor of categorical (nominal) data:
  - 1D shape: (batch\_size,)
  - 2D shape: (batch\_size, num\_classes)
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values

- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN`s when ``nan\_strategy = 'replace'`

**Return type**`Tensor`**Returns**

Pearson's Contingency Coefficient

**Example**

```
>>> from torchmetrics.functional.nominal import pearsons_contingency_coefficient
>>> _ = torch.manual_seed(42)
>>> preds = torch.randint(0, 4, (100,))
>>> target = torch.round(preds + torch.randn(100)).clamp(0, 4)
>>> pearsons_contingency_coefficient(preds, target)
tensor(0.6948)
```

**pearsons\_contingency\_coefficient\_matrix**

`torchmetrics.functional.nominal.pearsons_contingency_coefficient_matrix(matrix,`

*nan\_strategy='replace',*  
*nan\_replace\_value=0.0)*

Compute *Pearson's Contingency Coefficient* statistic between a set of multiple variables.

This can serve as a convenient tool to compute Pearson's Contingency Coefficient for analyses of correlation between categorical variables in your dataset.

**Parameters**

- **matrix** (`Tensor`) – A tensor of categorical (nominal) data, where:
  - rows represent a number of data points
  - columns represent a number of categorical (nominal) features
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN`s when ``nan\_strategy = 'replace'`

**Return type**`Tensor`**Returns**

Pearson's Contingency Coefficient statistic for a dataset of categorical variables

## Example

```
>>> from torchmetrics.functional.nominal import pearsons_contingency_coefficient_
↳ matrix
>>> _ = torch.manual_seed(42)
>>> matrix = torch.randint(0, 4, (200, 5))
>>> pearsons_contingency_coefficient_matrix(matrix)
tensor([[1.0000, 0.2326, 0.1959, 0.2262, 0.2989],
        [0.2326, 1.0000, 0.1386, 0.1895, 0.1329],
        [0.1959, 0.1386, 1.0000, 0.1840, 0.2335],
        [0.2262, 0.1895, 0.1840, 1.0000, 0.2737],
        [0.2989, 0.1329, 0.2335, 0.2737, 1.0000]])
```

## 1.78 Theil's U

### 1.78.1 Module Interface

**class** torchmetrics.TheilsU(num\_classes, nan\_strategy='replace', nan\_replace\_value=0.0, \*\*kwargs)

Compute *Theil's U* statistic measuring the association between two categorical (nominal) data series.

$$U(X|Y) = \frac{H(X) - H(X|Y)}{H(X)}$$

where  $H(X)$  is entropy of variable  $X$  while  $H(X|Y)$  is the conditional entropy of  $X$  given  $Y$ . It is also known as the Uncertainty Coefficient.

Theil's U is an asymmetric coefficient, i.e.  $TheilsU(preds, target) \neq TheilsU(target, preds)$ .

The output values lie in  $[0, 1]$ . 0 means  $y$  has no information about  $x$  while value 1 means  $y$  has complete information about  $x$ .

#### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN's when `nan_strategy = 'replace'`
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Returns

Tensor

#### Return type

Theil's U Statistic

## Example

```
>>> from torchmetrics.nominal import TheilsU
>>> _ = torch.manual_seed(42)
>>> preds = torch.randint(10, (10,))
>>> target = torch.randint(10, (10,))
>>> metric = TheilsU(num_classes=10)
>>> metric(preds, target)
tensor(0.8530)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

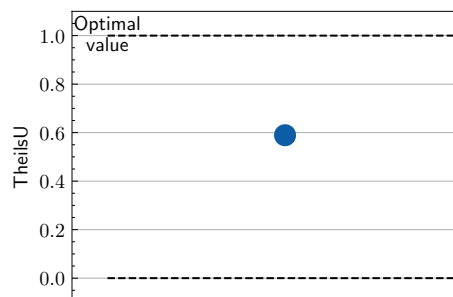
### Returns

Figure and Axes object

### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.nominal import TheilsU
>>> metric = TheilsU(num_classes=10)
>>> metric.update(torch.randint(10, (10,)), torch.randint(10, (10,)))
>>> fig_, ax_ = metric.plot()
```

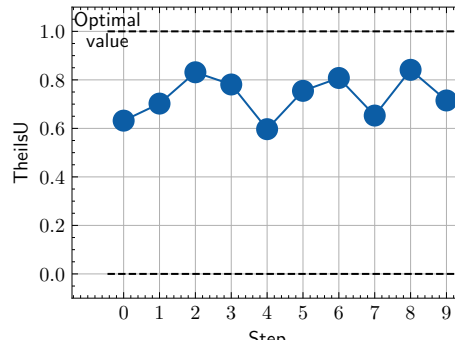


```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.nominal import TheilsU
>>> metric = TheilsU(num_classes=10)
>>> values = [ ]
```

(continues on next page)

(continued from previous page)

```
>>> for _ in range(10):
...     values.append(metric(torch.randint(10, (10,)), torch.randint(10, (10,
...     )))
>>> fig_, ax_ = metric.plot(values)
```



## 1.78.2 Functional Interface

`torchmetrics.functional.theils_u(preds, target, nan_strategy='replace', nan_replace_value=0.0)`

Compute *Theil's U* statistic (Uncertainty Coef.) measuring the association between two nominal data series.

$$U(X|Y) = \frac{H(X) - H(X|Y)}{H(X)}$$

where  $H(X)$  is entropy of variable  $X$  while  $H(X|Y)$  is the conditional entropy of  $X$  given  $Y$ .

Theil's  $U$  is an asymmetric coefficient, i.e.  $TheilsU(preds, target) \neq TheilsU(target, preds)$ .

The output values lies in  $[0, 1]$ . 0 means  $y$  has no information about  $x$  while value 1 means  $y$  has complete information about  $x$ .

### Parameters

- **preds** (**Tensor**) – 1D or 2D tensor of categorical (nominal) data - 1D shape: (batch\_size,) - 2D shape: (batch\_size, num\_classes)
- **target** (**Tensor**) – 1D or 2D tensor of categorical (nominal) data - 1D shape: (batch\_size,) - 2D shape: (batch\_size, num\_classes)
- **nan\_strategy** (**Literal**['replace', 'drop']) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (**Union**[int, float, None]) – Value to replace NaN's when `nan_strategy = 'replace'`

### Returns

Tensor

### Return type

Theil's  $U$  Statistic

### Example

```
>>> from torchmetrics.functional.nominal import theils_u
>>> _ = torch.manual_seed(42)
>>> preds = torch.randint(10, (10,))
>>> target = torch.randint(10, (10,))
>>> theils_u(preds, target)
tensor(0.8530)
```

### theils\_u\_matrix

`torchmetrics.functional.nominal.theils_u_matrix(matrix, nan_strategy='replace', nan_replace_value=0.0)`

Compute *Theil's U* statistic between a set of multiple variables.

This can serve as a convenient tool to compute Theil's U statistic for analyses of correlation between categorical variables in your dataset.

#### Parameters

- **matrix** (`Tensor`) – A tensor of categorical (nominal) data, where: - rows represent a number of data points - columns represent a number of categorical (nominal) features
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN's when `nan_strategy = 'replace'`

#### Return type

`Tensor`

#### Returns

Theil's U statistic for a dataset of categorical variables

### Example

```
>>> from torchmetrics.functional.nominal import theils_u_matrix
>>> _ = torch.manual_seed(42)
>>> matrix = torch.randint(0, 4, (200, 5))
>>> theils_u_matrix(matrix)
tensor([[1.0000, 0.0202, 0.0142, 0.0196, 0.0353],
        [0.0202, 1.0000, 0.0070, 0.0136, 0.0065],
        [0.0143, 0.0070, 1.0000, 0.0125, 0.0206],
        [0.0198, 0.0137, 0.0125, 1.0000, 0.0312],
        [0.0352, 0.0065, 0.0204, 0.0308, 1.0000]])
```



## 1.79 Tschuprow's T

### 1.79.1 Module Interface

**class** torchmetrics.TschuprowsT(*num\_classes*, *bias\_correction=True*, *nan\_strategy='replace'*, *nan\_replace\_value=0.0*, *\*\*kwargs*)

Compute *Tschuprow's T* statistic measuring the association between two categorical (nominal) data series.

$$T = \sqrt{\frac{\chi^2/n}{\sqrt{(r-1) * (k-1)}}}$$

where

$$\chi^2 = \sum_{i,j} \text{frac}\left(n_{ij} - \frac{n_{i.}n_{.j}}{n}\right)^2 \frac{n_{i.}n_{.j}}{n}$$

where  $n_{ij}$  denotes the number of times the values  $(A_i, B_j)$  are observed with  $A_i, B_j$  represent frequencies of values in `preds` and `target`, respectively.

Tschuprow's T is a symmetric coefficient, i.e.  $T(\text{preds}, \text{target}) = T(\text{target}, \text{preds})$ .

The output values lies in  $[0, 1]$  with 1 meaning the perfect association.

#### Parameters

- **num\_classes** (`int`) – Integer specifying the number of classes
- **bias\_correction** (`bool`) – Indication of whether to use bias correction.
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN's when `nan_strategy = 'replace'`
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Returns

Tschuprow's T statistic

#### Raises

- **ValueError** – If `nan_strategy` is not one of `'replace'` and `'drop'`
- **ValueError** – If `nan_strategy` is equal to `'replace'` and `nan_replace_value` is not an `int` or `float`

#### Example

```
>>> from torchmetrics.nominal import TschuprowsT
>>> _ = torch.manual_seed(42)
>>> preds = torch.randint(0, 4, (100,))
>>> target = torch.round(preds + torch.randn(100)).clamp(0, 4)
>>> tschuprows_t = TschuprowsT(num_classes=5)
>>> tschuprows_t(preds, target)
tensor(0.4930)
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

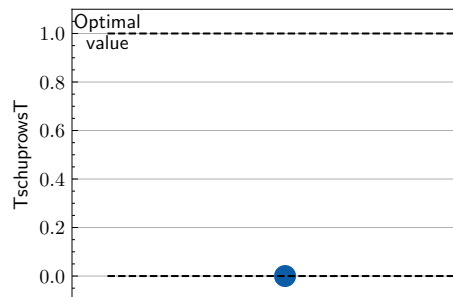
#### Returns

Figure and Axes object

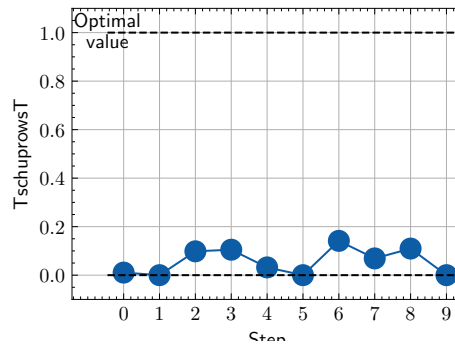
#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.nominal import TschuprowsT
>>> metric = TschuprowsT(num_classes=5)
>>> metric.update(torch.randint(0, 4, (100,)), torch.randint(0, 4, (100,)))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.nominal import TschuprowsT
>>> metric = TschuprowsT(num_classes=5)
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.randint(0, 4, (100,)), torch.randint(0, 4, (100,))))
>>> fig_, ax_ = metric.plot(values)
```



### 1.79.2 Functional Interface

`torchmetrics.functional.tschuprows_t(preds, target, bias_correction=True, nan_strategy='replace', nan_replace_value=0.0)`

Compute *Tschuprow's T* statistic measuring the association between two categorical (nominal) data series.

$$T = \sqrt{\frac{\chi^2/n}{\sqrt{(r-1) * (k-1)}}}$$

where

$$\chi^2 = \sum_{i,j} \frac{\left(n_{ij} - \frac{n_{i.}n_{.j}}{n}\right)^2 n_{i.}n_{.j}}{n}$$

where  $n_{ij}$  denotes the number of times the values  $(A_i, B_j)$  are observed with  $A_i, B_j$  represent frequencies of values in `preds` and `target`, respectively.

Tschuprow's T is a symmetric coefficient, i.e.  $T(preds, target) = T(target, preds)$ .

The output values lies in  $[0, 1]$  with 1 meaning the perfect association.

#### Parameters

- **preds** (`Tensor`) – 1D or 2D tensor of categorical (nominal) data:
  - 1D shape: (batch\_size,)
  - 2D shape: (batch\_size, num\_classes)
- **target** (`Tensor`) – 1D or 2D tensor of categorical (nominal) data:
  - 1D shape: (batch\_size,)
  - 2D shape: (batch\_size, num\_classes)
- **bias\_correction** (`bool`) – Indication of whether to use bias correction.
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN's when `nan_strategy = 'replace'`

#### Return type

`Tensor`

#### Returns

Tschuprow's T statistic

### Example

```
>>> from torchmetrics.functional.nominal import tschuprows_t
>>> _ = torch.manual_seed(42)
>>> preds = torch.randint(0, 4, (100,))
>>> target = torch.round(preds + torch.randn(100)).clamp(0, 4)
>>> tschuprows_t(preds, target)
tensor(0.4930)
```

### tschuprows\_t\_matrix

```
torchmetrics.functional.nominal.tschuprows_t_matrix(matrix, bias_correction=True,
                                                    nan_strategy='replace',
                                                    nan_replace_value=0.0)
```

Compute *Tschuprow's T* statistic between a set of multiple variables.

This can serve as a convenient tool to compute Tschuprow's T statistic for analyses of correlation between categorical variables in your dataset.

#### Parameters

- **matrix** (`Tensor`) – A tensor of categorical (nominal) data, where:
  - rows represent a number of data points
  - columns represent a number of categorical (nominal) features
- **bias\_correction** (`bool`) – Indication of whether to use bias correction.
- **nan\_strategy** (`Literal['replace', 'drop']`) – Indication of whether to replace or drop NaN values
- **nan\_replace\_value** (`Union[int, float, None]`) – Value to replace NaN's when ``nan\_strategy = 'replace'``

#### Return type

`Tensor`

#### Returns

Tschuprow's T statistic for a dataset of categorical variables

### Example

```
>>> from torchmetrics.functional.nominal import tschuprows_t_matrix
>>> _ = torch.manual_seed(42)
>>> matrix = torch.randint(0, 4, (200, 5))
>>> tschuprows_t_matrix(matrix)
tensor([[1.0000, 0.0637, 0.0000, 0.0542, 0.1337],
        [0.0637, 1.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 1.0000, 0.0000, 0.0649],
        [0.0542, 0.0000, 0.0000, 1.0000, 0.1100],
        [0.1337, 0.0000, 0.0649, 0.1100, 1.0000]])
```

## 1.80 Cosine Similarity

### 1.80.1 Functional Interface

`torchmetrics.functional.pairwise_cosine_similarity(x, y=None, reduction=None, zero_diagonal=None)`

Calculate pairwise cosine similarity.

$$s_{cos}(x, y) = \frac{\langle x, y \rangle}{\|x\| \cdot \|y\|} = \frac{\sum_{d=1}^D x_d \cdot y_d}{\sqrt{\sum_{d=1}^D x_d^2} \cdot \sqrt{\sum_{d=1}^D y_d^2}}$$

If both  $x$  and  $y$  are passed in, the calculation will be performed pairwise between the rows of  $x$  and  $y$ . If only  $x$  is passed in, the calculation will be performed between the rows of  $x$ .

#### Parameters

- **x** (`Tensor`) – Tensor with shape  $[N, d]$
- **y** (`Optional[Tensor]`) – Tensor with shape  $[M, d]$ , optional
- **reduction** (`Optional[Literal['mean', 'sum', 'none', None]]`) – reduction to apply along the last dimension. Choose between `'mean'`, `'sum'` (applied along column dimension) or `'none'`, `None` for no reduction
- **zero\_diagonal** (`Optional[bool]`) – if the diagonal of the distance matrix should be set to 0. If only  $x$  is given this defaults to `True` else if  $y$  is also given it defaults to `False`

#### Return type

`Tensor`

#### Returns

A  $[N, N]$  matrix of distances if only  $x$  is given, else a  $[N, M]$  matrix

#### Example

```
>>> import torch
>>> from torchmetrics.functional.pairwise import pairwise_cosine_similarity
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_cosine_similarity(x, y)
tensor([[0.5547, 0.8682],
        [0.5145, 0.8437],
        [0.5300, 0.8533]])
>>> pairwise_cosine_similarity(x)
tensor([[0.0000, 0.9989, 0.9996],
        [0.9989, 0.0000, 0.9998],
        [0.9996, 0.9998, 0.0000]])
```

## 1.81 Euclidean Distance

### 1.81.1 Functional Interface

`torchmetrics.functional.pairwise_euclidean_distance(x, y=None, reduction=None, zero_diagonal=None)`

Calculate pairwise euclidean distances.

$$d_{\text{euc}}(x, y) = \|x - y\|_2 = \sqrt{\sum_{d=1}^D (x_d - y_d)^2}$$

If both  $x$  and  $y$  are passed in, the calculation will be performed pairwise between the rows of  $x$  and  $y$ . If only  $x$  is passed in, the calculation will be performed between the rows of  $x$ .

#### Parameters

- **x** (`Tensor`) – Tensor with shape  $[N, d]$
- **y** (`Optional[Tensor]`) – Tensor with shape  $[M, d]$ , optional
- **reduction** (`Optional[Literal['mean', 'sum', 'none', None]]`) – reduction to apply along the last dimension. Choose between ‘mean’, ‘sum’ (applied along column dimension) or ‘none’, *None* for no reduction
- **zero\_diagonal** (`Optional[bool]`) – if the diagonal of the distance matrix should be set to 0. If only  $x$  is given this defaults to *True* else if  $y$  is also given it defaults to *False*

#### Return type

`Tensor`

#### Returns

A  $[N, N]$  matrix of distances if only  $x$  is given, else a  $[N, M]$  matrix

#### Example

```
>>> import torch
>>> from torchmetrics.functional.pairwise import pairwise_euclidean_distance
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_euclidean_distance(x, y)
tensor([[3.1623, 2.0000],
        [5.3852, 4.1231],
        [8.9443, 7.6158]])
>>> pairwise_euclidean_distance(x)
tensor([[0.0000, 2.2361, 5.8310],
        [2.2361, 0.0000, 3.6056],
        [5.8310, 3.6056, 0.0000]])
```

## 1.82 Linear Similarity

### 1.82.1 Functional Interface

`torchmetrics.functional.pairwise_linear_similarity(x, y=None, reduction=None, zero_diagonal=None)`

Calculate pairwise linear similarity.

$$s_{lin}(x, y) = \langle x, y \rangle = \sum_{d=1}^D x_d \cdot y_d$$

If both  $x$  and  $y$  are passed in, the calculation will be performed pairwise between the rows of  $x$  and  $y$ . If only  $x$  is passed in, the calculation will be performed between the rows of  $x$ .

#### Parameters

- **x** (`Tensor`) – Tensor with shape  $[N, d]$
- **y** (`Optional[Tensor]`) – Tensor with shape  $[M, d]$ , optional
- **reduction** (`Optional[Literal['mean', 'sum', 'none', None]]`) – reduction to apply along the last dimension. Choose between ‘*mean*’, ‘*sum*’ (applied along column dimension) or ‘*none*’, *None* for no reduction
- **zero\_diagonal** (`Optional[bool]`) – if the diagonal of the distance matrix should be set to 0. If only  $x$  is given this defaults to *True* else if  $y$  is also given it defaults to *False*

#### Return type

`Tensor`

#### Returns

A  $[N, N]$  matrix of distances if only  $x$  is given, else a  $[N, M]$  matrix

#### Example

```
>>> import torch
>>> from torchmetrics.functional.pairwise import pairwise_linear_similarity
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_linear_similarity(x, y)
tensor([[ 2.,  7.],
        [ 3., 11.],
        [ 5., 18.]])
>>> pairwise_linear_similarity(x)
tensor([[ 0., 21., 34.],
        [21.,  0., 55.],
        [34., 55.,  0.]])
```

## 1.83 Manhattan Distance

### 1.83.1 Functional Interface

`torchmetrics.functional.pairwise_manhattan_distance(x, y=None, reduction=None, zero_diagonal=None)`

Calculate pairwise manhattan distance.

$$d_{man}(x, y) = ||x - y||_1 = \sum_{d=1}^D |x_d - y_d|$$

If both  $x$  and  $y$  are passed in, the calculation will be performed pairwise between the rows of  $x$  and  $y$ . If only  $x$  is passed in, the calculation will be performed between the rows of  $x$ .

#### Parameters

- **x** (`Tensor`) – Tensor with shape  $[N, d]$
- **y** (`Optional[Tensor]`) – Tensor with shape  $[M, d]$ , optional
- **reduction** (`Optional[Literal['mean', 'sum', 'none', None]]`) – reduction to apply along the last dimension. Choose between ‘*mean*’, ‘*sum*’ (applied along column dimension) or ‘*none*’, *None* for no reduction
- **zero\_diagonal** (`Optional[bool]`) – if the diagonal of the distance matrix should be set to 0. If only  $x$  is given this defaults to *True* else if  $y$  is also given it defaults to *False*

#### Return type

`Tensor`

#### Returns

A  $[N, N]$  matrix of distances if only  $x$  is given, else a  $[N, M]$  matrix

#### Example

```
>>> import torch
>>> from torchmetrics.functional.pairwise import pairwise_manhattan_distance
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_manhattan_distance(x, y)
tensor([[ 4.,  2.],
        [ 7.,  5.],
        [12., 10.]])
>>> pairwise_manhattan_distance(x)
tensor([[0., 3., 8.],
        [3., 0., 5.],
        [8., 5., 0.]])
```



## 1.84 Minkowski Distance

### 1.84.1 Functional Interface

`torchmetrics.functional.pairwise_minkowski_distance(x, y=None, exponent=2, reduction=None, zero_diagonal=None)`

Calculate pairwise minkowski distances.

$$d_{minkowski}(x, y, p) = \|x - y\|_p = \sqrt[p]{\sum_{d=1}^D (x_d - y_d)^p}$$

If both  $x$  and  $y$  are passed in, the calculation will be performed pairwise between the rows of  $x$  and  $y$ . If only  $x$  is passed in, the calculation will be performed between the rows of  $x$ .

#### Parameters

- **x** (`Tensor`) – Tensor with shape  $[N, d]$
- **y** (`Optional[Tensor]`) – Tensor with shape  $[M, d]$ , optional
- **exponent** (`Union[int, float]`) – int or float larger than 1, exponent to which the difference between preds and target is to be raised
- **reduction** (`Optional[Literal['mean', 'sum', 'none', None]]`) – reduction to apply along the last dimension. Choose between `'mean'`, `'sum'` (applied along column dimension) or `'none'`, `None` for no reduction
- **zero\_diagonal** (`Optional[bool]`) – if the diagonal of the distance matrix should be set to 0. If only  $x$  is given this defaults to `True` else if  $y$  is also given it defaults to `False`

#### Return type

`Tensor`

#### Returns

A  $[N, N]$  matrix of distances if only  $x$  is given, else a  $[N, M]$  matrix

#### Example

```
>>> import torch
>>> from torchmetrics.functional.pairwise import pairwise_minkowski_distance
>>> x = torch.tensor([[2, 3], [3, 5], [5, 8]], dtype=torch.float32)
>>> y = torch.tensor([[1, 0], [2, 1]], dtype=torch.float32)
>>> pairwise_minkowski_distance(x, y, exponent=4)
tensor([[3.0092, 2.0000],
        [5.0317, 4.0039],
        [8.1222, 7.0583]])
>>> pairwise_minkowski_distance(x, exponent=4)
tensor([[0.0000, 2.0305, 5.1547],
        [2.0305, 0.0000, 3.1383],
        [5.1547, 3.1383, 0.0000]])
```

## 1.85 Concordance Corr. Coef.

### 1.85.1 Module Interface

**class** torchmetrics.ConcordanceCorrCoef(num\_outputs=1, \*\*kwargs)

Compute concordance correlation coefficient that measures the agreement between two variables.

$$\rho_c = \frac{2\rho\sigma_x\sigma_y}{\sigma_x^2 + \sigma_y^2 + (\mu_x - \mu_y)^2}$$

where  $\mu_x, \mu_y$  is the means for the two variables,  $\sigma_x^2, \sigma_y^2$  are the corresponding variances and rho is the pearson correlation coefficient between the two variables.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): either single output float tensor with shape (N,) or multioutput float tensor of shape (N,d)
- **target** (**Tensor**): either single output float tensor with shape (N,) or multioutput float tensor of shape (N,d)

As output of forward and compute the metric returns the following output:

- **concordance** (**Tensor**): A scalar float tensor with the concordance coefficient(s) for non-multioutput input or a float tensor with shape (d,) for multioutput input

#### Parameters

- **num\_outputs** (**int**) – Number of outputs in multioutput setting
- **kwargs** (**Any**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (single output regression):**

```
>>> from torchmetrics.regression import ConcordanceCorrCoef
>>> from torch import tensor
>>> target = tensor([3, -0.5, 2, 7])
>>> preds = tensor([2.5, 0.0, 2, 8])
>>> concordance = ConcordanceCorrCoef()
>>> concordance(preds, target)
tensor(0.9777)
```

**Example (multi output regression):**

```
>>> from torchmetrics.regression import ConcordanceCorrCoef
>>> target = tensor([[3, -0.5], [2, 7]])
>>> preds = tensor([[2.5, 0.0], [2, 8]])
>>> concordance = ConcordanceCorrCoef(num_outputs=2)
>>> concordance(preds, target)
tensor([0.7273, 0.9887])
```

**plot**(val=None, ax=None)

Plot a single or multiple values from the metric.

#### Parameters

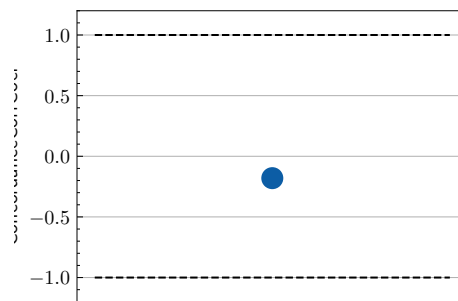
- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

Figure and Axes object

**Raises****`ModuleNotFoundError`** – If `matplotlib` is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import ConcordanceCorrCoef
>>> metric = ConcordanceCorrCoef()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import ConcordanceCorrCoef
>>> metric = ConcordanceCorrCoef()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```

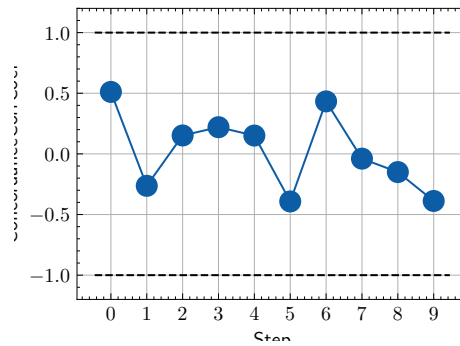
## 1.85.2 Functional Interface

`torchmetrics.functional.concordance_corrcoef(preds, target)`

Compute concordance correlation coefficient that measures the agreement between two variables.

$$\rho_c = \frac{2\rho\sigma_x\sigma_y}{\sigma_x^2 + \sigma_y^2 + (\mu_x - \mu_y)^2}$$

where  $\mu_x, \mu_y$  is the means for the two variables,  $\sigma_x^2, \sigma_y^2$  are the corresponding variances and rho is the pearson correlation coefficient between the two variables.

**Parameters**

- **preds** (`Tensor`) – estimated scores
- **target** (`Tensor`) – ground truth scores

**Return type**`Tensor`**Example (single output regression):**

```
>>> from torchmetrics.functional.regression import concordance_corrcoef
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> concordance_corrcoef(preds, target)
tensor([0.9777])
```

**Example (multi output regression):**

```
>>> from torchmetrics.functional.regression import concordance_corrcoef
>>> target = torch.tensor([[3, -0.5], [2, 7]])
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> concordance_corrcoef(preds, target)
tensor([0.7273, 0.9887])
```

## 1.86 Cosine Similarity

### 1.86.1 Module Interface

**class** `torchmetrics.CosineSimilarity`(*reduction='sum', \*\*kwargs*)

Compute the *Cosine Similarity*.

$$\text{cos}_{sim}(x, y) = \frac{x \cdot y}{\|x\| \cdot \|y\|} = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}}$$

where  $y$  is a tensor of target values, and  $x$  is a tensor of predictions.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`Tensor`): Predicted float tensor with shape (N, d)
- **target** (`Tensor`): Ground truth float tensor with shape (N, d)

As output of `forward` and `compute` the metric returns the following output:

- `cosine_similarity` (`Tensor`): A float tensor with the cosine similarity

#### Parameters

- **reduction** (`Literal`['mean', 'sum', 'none', None]) – how to reduce over the batch dimension using 'sum', 'mean' or 'none' (taking the individual scores)
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torch import tensor
>>> from torchmetrics.regression import CosineSimilarity
>>> target = tensor([[0, 1], [1, 1]])
>>> preds = tensor([[0, 1], [0, 1]])
>>> cosine_similarity = CosineSimilarity(reduction = 'mean')
>>> cosine_similarity(preds, target)
tensor(0.8536)
```

`plot`(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union`[`Tensor`, `Sequence`[`Tensor`], None]) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional`[`Axes`]) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple`[`Figure`, `Union`[`Axes`, `ndarray`]]

#### Returns

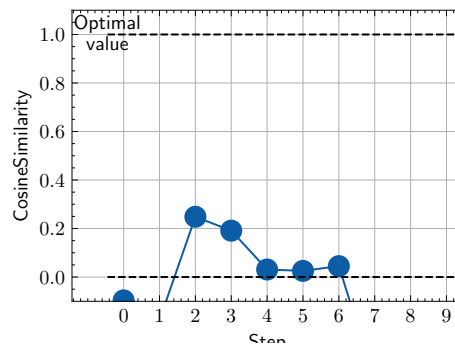
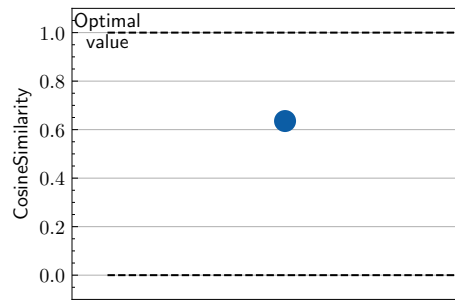
Figure and Axes object

#### Raises

`ModuleNotFoundError` – If `matplotlib` is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import CosineSimilarity
>>> metric = CosineSimilarity()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```

```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import CosineSimilarity
>>> metric = CosineSimilarity()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```



## 1.86.2 Functional Interface

`torchmetrics.functional.cosine_similarity(preds, target, reduction='sum')`

Compute the *Cosine Similarity*.

$$\text{cos}_{sim}(x, y) = \frac{x \cdot y}{\|x\| \cdot \|y\|} = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}}$$

where  $y$  is a tensor of target values, and  $x$  is a tensor of predictions.

### Parameters

- **preds** (`Tensor`) – Predicted tensor with shape (N,d)
- **target** (`Tensor`) – Ground truth tensor with shape (N,d)
- **reduction** (`Optional[str]`) – The method of reducing along the batch dimension using sum, mean or taking the individual scores

### Return type

`Tensor`

### Example

```
>>> from torchmetrics.functional.regression import cosine_similarity
>>> target = torch.tensor([[1, 2, 3, 4],
...                        [1, 2, 3, 4]])
>>> preds = torch.tensor([[1, 2, 3, 4],
...                       [-1, -2, -3, -4]])
>>> cosine_similarity(preds, target, 'none')
tensor([ 1.0000, -1.0000])
```

## 1.87 Explained Variance

### 1.87.1 Module Interface

**class** torchmetrics.ExplainedVariance(*multioutput='uniform\_average', \*\*kwargs*)

Compute *explained variance*.

$$\text{ExplainedVariance} = 1 - \frac{\text{Var}(y - \hat{y})}{\text{Var}(y)}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): Predictions from model in float tensor with shape (N,) or (N, ...) (multioutput)
- **target** (**Tensor**): Ground truth values in long tensor with shape (N,) or (N, ...) (multioutput)

As output of forward and compute the metric returns the following output:

- **explained\_variance** (**Tensor**): A tensor with the explained variance(s)

In the case of multioutput, as default the variances will be uniformly averaged over the additional dimensions. Please see argument *multioutput* for changing this behavior.

#### Parameters

- **multioutput** (**Literal**['raw\_values', 'uniform\_average', 'variance\_weighted']) – Defines aggregation in the case of multiple output scores. Can be one of the following strings (default is 'uniform\_average'.):
  - 'raw\_values' returns full set of scores
  - 'uniform\_average' scores are uniformly averaged
  - 'variance\_weighted' scores are weighted by their individual variances
- **kwargs** (**Any**) – Additional keyword arguments, see *Advanced metric settings* for more info.

#### Raises

**ValueError** – If multioutput is not one of "raw\_values", "uniform\_average" or "variance\_weighted".

## Example

```
>>> from torch import tensor
>>> from torchmetrics.regression import ExplainedVariance
>>> target = tensor([3, -0.5, 2, 7])
>>> preds = tensor([2.5, 0.0, 2, 8])
>>> explained_variance = ExplainedVariance()
>>> explained_variance(preds, target)
tensor(0.9572)
```

```
>>> target = tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = tensor([[0, 2], [-1, 2], [8, -5]])
>>> explained_variance = ExplainedVariance(multioutput='raw_values')
>>> explained_variance(preds, target)
tensor([0.9677, 1.0000])
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

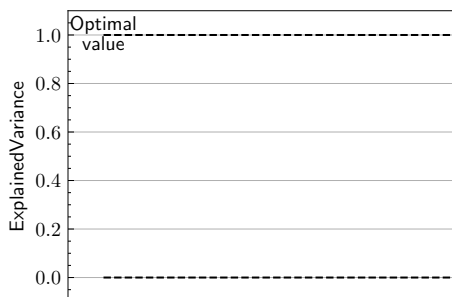
### Returns

Figure and Axes object

### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import ExplainedVariance
>>> metric = ExplainedVariance()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```

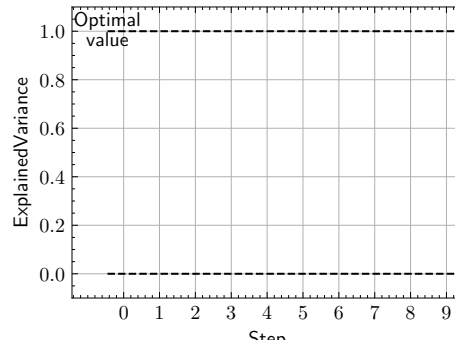




```

>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import ExplainedVariance
>>> metric = ExplainedVariance()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)

```



### 1.87.2 Functional Interface

`torchmetrics.functional.explained_variance(preds, target, multioutput='uniform_average')`

Compute explained variance.

#### Parameters

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels
- **multioutput** (`Literal['raw_values', 'uniform_average', 'variance_weighted']`) – Defines aggregation in the case of multiple output scores. Can be one of the following strings:
  - 'raw\_values' returns full set of scores
  - 'uniform\_average' scores are uniformly averaged
  - 'variance\_weighted' scores are weighted by their individual variances

#### Return type

`Union[Tensor, Sequence[Tensor]]`

### Example

```
>>> from torchmetrics.functional.regression import explained_variance
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> explained_variance(preds, target)
tensor(0.9572)
```

```
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> explained_variance(preds, target, multioutput='raw_values')
tensor([0.9677, 1.0000])
```

## 1.88 Kendall Rank Corr. Coef.

### 1.88.1 Module Interface

**class** torchmetrics.KendallRankCorrCoef(*variant='b', t\_test=False, alternative='two-sided', num\_outputs=1, \*\*kwargs*)

Compute Kendall Rank Correlation Coefficient.

$$\tau_a = \frac{C - D}{C + D}$$

where  $C$  represents concordant pairs,  $D$  stands for discordant pairs.

$$\tau_b = \frac{C - D}{\sqrt{(C + D + T_{preds}) * (C + D + T_{target})}}$$

where  $C$  represents concordant pairs,  $D$  stands for discordant pairs and  $T$  represents a total number of ties.

$$\tau_c = 2 * \frac{C - D}{n^2 * \frac{m-1}{m}}$$

where  $C$  represents concordant pairs,  $D$  stands for discordant pairs,  $n$  is a total number of observations and  $m$  is a min of unique values in `preds` and `target` sequence.

Definitions according to Definition according to [The Treatment of Ties in Ranking Problems](#).

As input to `forward` and `update` the metric accepts the following input:

- `preds` ([Tensor](#)): Sequence of data in float tensor of either shape  $(N,)$  or  $(N, d)$
- `target` ([Tensor](#)): Sequence of data in float tensor of either shape  $(N,)$  or  $(N, d)$

As output of `forward` and `compute` the metric returns the following output:

- `kendall` ([Tensor](#)): A tensor with the correlation tau statistic, and if it is not None, the p-value of corresponding statistical test.

#### Parameters

- **variant** ([Literal](#)['a', 'b', 'c']) – Indication of which variant of Kendall's tau to be used
- **t\_test** ([bool](#)) – Indication whether to run t-test

- **alternative** (`Optional[Literal['two-sided', 'less', 'greater']]`) – Alternative hypothesis for t-test. Possible values: - ‘two-sided’: the rank correlation is nonzero - ‘less’: the rank correlation is negative (less than zero) - ‘greater’: the rank correlation is positive (greater than zero)
- **num\_outputs** (`int`) – Number of outputs in multioutput setting
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises**

- **ValueError** – If `t_test` is not of a type `bool`
- **ValueError** – If `t_test=True` and `alternative=None`

**Example (single output regression):**

```
>>> from torch import tensor
>>> from torchmetrics.regression import KendallRankCorrCoef
>>> preds = tensor([2.5, 0.0, 2, 8])
>>> target = tensor([3, -0.5, 2, 1])
>>> kendall = KendallRankCorrCoef()
>>> kendall(preds, target)
tensor(0.3333)
```

**Example (multi output regression):**

```
>>> from torchmetrics.regression import KendallRankCorrCoef
>>> preds = tensor([[2.5, 0.0], [2, 8]])
>>> target = tensor([[3, -0.5], [2, 1]])
>>> kendall = KendallRankCorrCoef(num_outputs=2)
>>> kendall(preds, target)
tensor([1., 1.])
```

**Example (single output regression with t-test):**

```
>>> from torchmetrics.regression import KendallRankCorrCoef
>>> preds = tensor([2.5, 0.0, 2, 8])
>>> target = tensor([3, -0.5, 2, 1])
>>> kendall = KendallRankCorrCoef(t_test=True, alternative='two-sided')
>>> kendall(preds, target)
(tensor(0.3333), tensor(0.4969))
```

**Example (multi output regression with t-test):**

```
>>> from torchmetrics.regression import KendallRankCorrCoef
>>> preds = tensor([[2.5, 0.0], [2, 8]])
>>> target = tensor([[3, -0.5], [2, 1]])
>>> kendall = KendallRankCorrCoef(t_test=True, alternative='two-sided', num_
↳ outputs=2)
>>> kendall(preds, target)
(tensor([1., 1.]), tensor([nan, nan]))
```

**plot** (`val=None, ax=None`)

Plot a single or multiple values from the metric.

**Parameters**

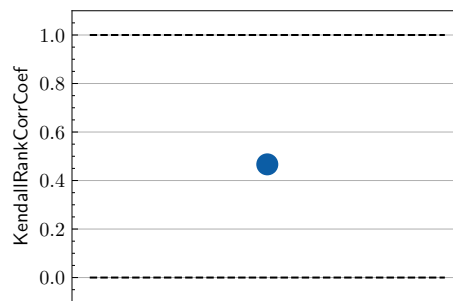
- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union[Axes, ndarray]]`**Returns**

Figure and Axes object

**Raises****`ModuleNotFoundError`** – If `matplotlib` is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import KendallRankCorrCoef
>>> metric = KendallRankCorrCoef()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```



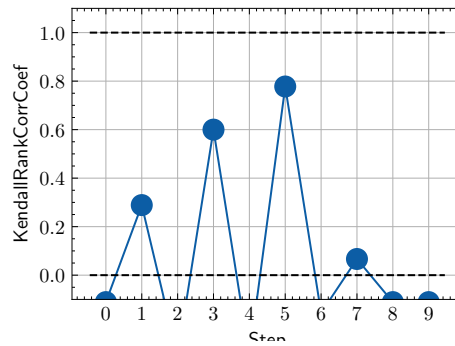
```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import KendallRankCorrCoef
>>> metric = KendallRankCorrCoef()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```

## 1.88.2 Functional Interface

`torchmetrics.functional.kendall_rank_corrcoef(preds, target, variant='b', t_test=False, alternative='two-sided')`

Compute Kendall Rank Correlation Coefficient.

$$\tau_a = \frac{C - D}{C + D}$$



where  $C$  represents concordant pairs,  $D$  stands for discordant pairs.

$$\tau_{ab} = \frac{C - D}{\sqrt{(C + D + T_{preds}) * (C + D + T_{target})}}$$

where  $C$  represents concordant pairs,  $D$  stands for discordant pairs and  $T$  represents a total number of ties.

$$\tau_c = 2 * \frac{C - D}{n^2 * \frac{m-1}{m}}$$

where  $C$  represents concordant pairs,  $D$  stands for discordant pairs,  $n$  is a total number of observations and  $m$  is a min of unique values in `preds` and `target` sequence.

Definitions according to Definition according to [The Treatment of Ties in Ranking Problems](#).

#### Parameters

- **preds** ([Tensor](#)) – Sequence of data of either shape  $(N,)$  or  $(N,d)$
- **target** ([Tensor](#)) – Sequence of data of either shape  $(N,)$  or  $(N,d)$
- **variant** ([Literal](#)['a', 'b', 'c']) – Indication of which variant of Kendall's tau to be used
- **t\_test** ([bool](#)) – Indication whether to run t-test
- **alternative** ([Optional](#)[[Literal](#)['two-sided', 'less', 'greater']]) – Alternative hypothesis for t-test. Possible values: - 'two-sided': the rank correlation is nonzero - 'less': the rank correlation is negative (less than zero) - 'greater': the rank correlation is positive (greater than zero)

#### Return type

[Union](#)[[Tensor](#), [Tuple](#)[[Tensor](#), [Tensor](#)]]

#### Returns

Correlation tau statistic (Optional) p-value of corresponding statistical test (asymptotic)

#### Raises

- **ValueError** – If `t_test` is not of a type `bool`
- **ValueError** – If `t_test=True` and `alternative=None`

**Example (single output regression):**

```
>>> from torchmetrics.functional.regression import kendall_rank_corrcoef
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> target = torch.tensor([3, -0.5, 2, 1])
>>> kendall_rank_corrcoef(preds, target)
tensor(0.3333)
```

Example (multi output regression):

```
>>> from torchmetrics.functional.regression import kendall_rank_corrcoef
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> target = torch.tensor([[3, -0.5], [2, 1]])
>>> kendall_rank_corrcoef(preds, target)
tensor([1., 1.])
```

Example (single output regression with t-test)

```
>>> from torchmetrics.functional.regression import kendall_rank_corrcoef
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> target = torch.tensor([3, -0.5, 2, 1])
>>> kendall_rank_corrcoef(preds, target, t_test=True, alternative='two-sided')
(tensor(0.3333), tensor(0.4969))
```

Example (multi output regression with t-test):

```
>>> from torchmetrics.functional.regression import kendall_rank_corrcoef
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> target = torch.tensor([[3, -0.5], [2, 1]])
>>> kendall_rank_corrcoef(preds, target, t_test=True, alternative='two-sided')
(tensor([1., 1.]), tensor([nan, nan]))
```

## 1.89 KL Divergence

### 1.89.1 Module Interface

**class** torchmetrics.KLDivergence(log\_prob=False, reduction='mean', \*\*kwargs)

Compute the *KL divergence*.

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)}$$

Where  $P$  and  $Q$  are probability distributions where  $P$  usually represents a distribution over data and  $Q$  is often a prior or approximation of  $P$ . It should be noted that the KL divergence is a non-symmetrical metric i.e.  $D_{KL}(P||Q) \neq D_{KL}(Q||P)$ .

As input to forward and update the metric accepts the following input:

- **p** (**Tensor**): a data distribution with shape (N, d)
- **q** (**Tensor**): prior or approximate distribution with shape (N, d)

As output of forward and compute the metric returns the following output:

- **kl\_divergence** (**Tensor**): A tensor with the KL divergence

**Parameters**

- **log\_prob** (`bool`) – bool indicating if input is log-probabilities or probabilities. If given as probabilities, will normalize to make sure the distributes sum to 1.
- **reduction** (`Literal['mean', 'sum', 'none', None]`) – Determines how to reduce over the N/batch dimension:
  - 'mean' [default]: Averages score across samples
  - 'sum': Sum score across samples
  - 'none' or None: Returns score per sample
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Raises**

- **TypeError** – If log\_prob is not an bool.
- **ValueError** – If reduction is not one of 'mean', 'sum', 'none' or None.

---

**Note:** Half precision is only support on GPU for this metric

---

**Example**

```
>>> from torch import tensor
>>> from torchmetrics.regression import KLDivergence
>>> p = tensor([[0.36, 0.48, 0.16]])
>>> q = tensor([[1/3, 1/3, 1/3]])
>>> kl_divergence = KLDivergence()
>>> kl_divergence(p, q)
tensor(0.0853)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

**Parameters**

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

**Returns**

Figure and Axes object

**Raises**

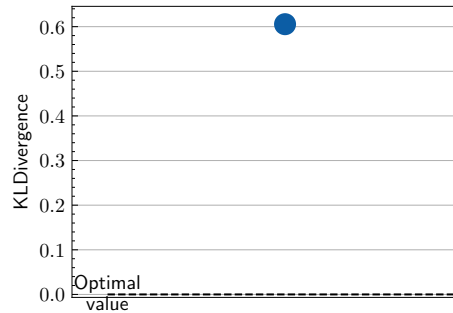
**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import KLDivergence
>>> metric = KLDivergence()
```

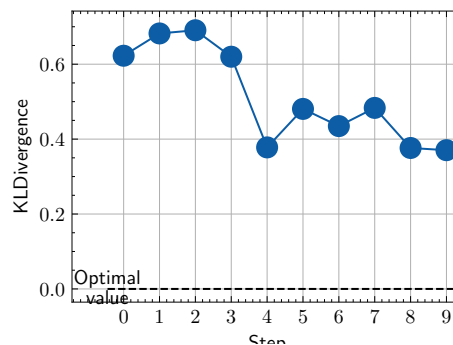
(continues on next page)

(continued from previous page)

```
>>> metric.update(randn(10,3).softmax(dim=-1), randn(10,3).softmax(dim=-1))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import KLDivergence
>>> metric = KLDivergence()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,3).softmax(dim=-1), randn(10,3).
... softmax(dim=-1)))
>>> fig, ax = metric.plot(values)
```



## 1.89.2 Functional Interface

`torchmetrics.functional.kl_divergence(p, q, log_prob=False, reduction='mean')`

Compute *KL divergence*.

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)}$$

Where  $P$  and  $Q$  are probability distributions where  $P$  usually represents a distribution over data and  $Q$  is often a prior or approximation of  $P$ . It should be noted that the KL divergence is a non-symmetrical metric i.e.  $D_{KL}(P||Q) \neq D_{KL}(Q||P)$ .



**Parameters**

- **p** (`Tensor`) – data distribution with shape  $[N, d]$
- **q** (`Tensor`) – prior or approximate distribution with shape  $[N, d]$
- **log\_prob** (`bool`) – bool indicating if input is log-probabilities or probabilities. If given as probabilities, will normalize to make sure the distributes sum to 1
- **reduction** (`Literal['mean', 'sum', 'none', None]`) – Determines how to reduce over the  $N$ /batch dimension:
  - 'mean' [default]: Averages score across samples
  - 'sum': Sum score across samples
  - 'none' or `None`: Returns score per sample

**Return type**`Tensor`**Example**

```
>>> from torch import tensor
>>> p = tensor([[0.36, 0.48, 0.16]])
>>> q = tensor([[1/3, 1/3, 1/3]])
>>> kl_divergence(p, q)
tensor(0.0853)
```

## 1.90 Log Cosh Error

### 1.90.1 Module Interface

**class** `torchmetrics.LogCoshError`(*num\_outputs=1, \*\*kwargs*)

Compute the `LogCosh` Error.

$$\text{LogCoshError} = \log \left( \frac{\exp(\hat{y} - y) + \exp(y - \hat{y})}{2} \right)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to forward and update the metric accepts the following input:

- **preds** (`Tensor`): Estimated labels with shape  $(\text{batch\_size},)$  or  $(\text{batch\_size}, \text{num\_outputs})$
- **target** (`Tensor`): Ground truth labels with shape  $(\text{batch\_size},)$  or  $(\text{batch\_size}, \text{num\_outputs})$

As output of forward and compute the metric returns the following output:

- **log\_cosh\_error** (`Tensor`): A tensor with the log cosh error

**Parameters**

- **num\_outputs** (`int`) – Number of outputs in multioutput setting
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (single output regression)::**

```
>>> from torchmetrics.regression import LogCoshError
>>> preds = torch.tensor([3.0, 5.0, 2.5, 7.0])
>>> target = torch.tensor([2.5, 5.0, 4.0, 8.0])
>>> log_cosh_error = LogCoshError()
>>> log_cosh_error(preds, target)
tensor(0.3523)
```

**Example (multi output regression)::**

```
>>> from torchmetrics.regression import LogCoshError
>>> preds = torch.tensor([[3.0, 5.0, 1.2], [-2.1, 2.5, 7.0]])
>>> target = torch.tensor([[2.5, 5.0, 1.3], [0.3, 4.0, 8.0]])
>>> log_cosh_error = LogCoshError(num_outputs=3)
>>> log_cosh_error(preds, target)
tensor([0.9176, 0.4277, 0.2194])
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

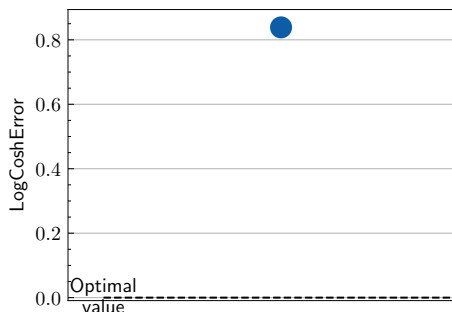
#### Returns

Figure and Axes object

#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

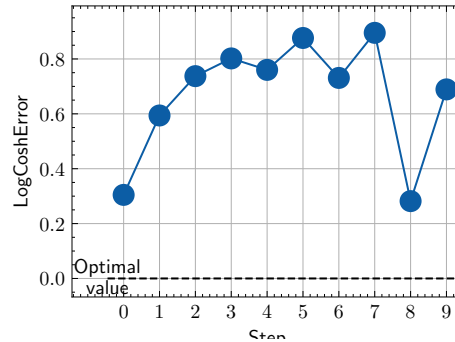
```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import LogCoshError
>>> metric = LogCoshError()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```



```

>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import LogCoshError
>>> metric = LogCoshError()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)

```



## 1.90.2 Functional Interface

`torchmetrics.functional.log_cosh_error(preds, target)`

Compute the `LogCosh` Error.

$$\text{LogCoshError} = \log \left( \frac{\exp(\hat{y} - y) + \exp(y - \hat{y})}{2} \right)$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

### Parameters

- **preds** (`Tensor`) – estimated labels with shape `(batch_size,)` or `(batch_size, num_outputs)`
- **target** (`Tensor`) – ground truth labels with shape `(batch_size,)` or `(batch_size, num_outputs)`

### Return type

`Tensor`

### Returns

Tensor with LogCosh error

**Example (single output regression)::**

```

>>> from torchmetrics.functional.regression import log_cosh_error
>>> preds = torch.tensor([3.0, 5.0, 2.5, 7.0])
>>> target = torch.tensor([2.5, 5.0, 4.0, 8.0])
>>> log_cosh_error(preds, target)
tensor(0.3523)

```

**Example (multi output regression)::**

```
>>> from torchmetrics.functional.regression import log_cosh_error
>>> preds = torch.tensor([[3.0, 5.0, 1.2], [-2.1, 2.5, 7.0]])
>>> target = torch.tensor([[2.5, 5.0, 1.3], [0.3, 4.0, 8.0]])
>>> log_cosh_error(preds, target)
tensor([0.9176, 0.4277, 0.2194])
```

## 1.91 Mean Absolute Error (MAE)

### 1.91.1 Module Interface

**class** torchmetrics.MeanAbsoluteError(\*\*kwargs)

Compute Mean Absolute Error (MAE).

$$\text{MAE} = \frac{1}{N} \sum_i^N |y_i - \hat{y}_i|$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): Predictions from model
- **target** ([Tensor](#)): Ground truth values

As output of forward and compute the metric returns the following output:

- **mean\_absolute\_error** ([Tensor](#)): A tensor with the mean absolute error over the state

#### Parameters

**kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torch import tensor
>>> from torchmetrics.regression import MeanAbsoluteError
>>> target = tensor([3.0, -0.5, 2.0, 7.0])
>>> preds = tensor([2.5, 0.0, 2.0, 8.0])
>>> mean_absolute_error = MeanAbsoluteError()
>>> mean_absolute_error(preds, target)
tensor(0.5000)
```

**plot**(val=None, ax=None)

Plot a single or multiple values from the metric.

#### Parameters

- **val** ([Union](#)[[Tensor](#), [Sequence](#)[[Tensor](#)], [None](#)]) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** ([Optional](#)[[Axes](#)]) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

[Tuple](#)[[Figure](#), [Union](#)[[Axes](#), [ndarray](#)]]

**Returns**

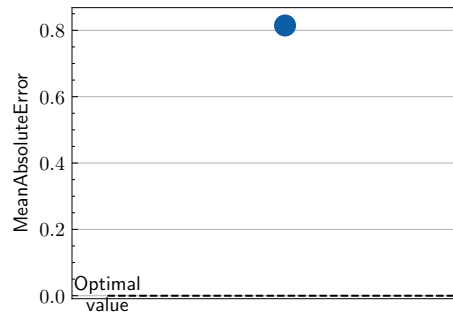
Figure and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```

>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import MeanAbsoluteError
>>> metric = MeanAbsoluteError()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()

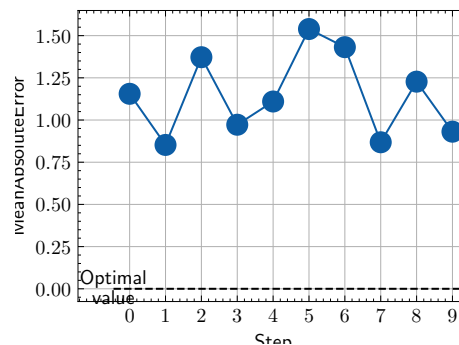
```



```

>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import MeanAbsoluteError
>>> metric = MeanAbsoluteError()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)

```



### 1.91.2 Functional Interface

`torchmetrics.functional.mean_absolute_error(preds, target)`

Compute mean absolute error.

**Parameters**

- **preds** ([Tensor](#)) – estimated labels
- **target** ([Tensor](#)) – ground truth labels

**Return type**

[Tensor](#)

**Returns**

Tensor with MAE

**Example**

```
>>> from torchmetrics.functional.regression import mean_absolute_error
>>> x = torch.tensor([0., 1, 2, 3])
>>> y = torch.tensor([0., 1, 2, 2])
>>> mean_absolute_error(x, y)
tensor(0.2500)
```

## 1.92 Mean Absolute Percentage Error (MAPE)

### 1.92.1 Module Interface

`class torchmetrics.MeanAbsolutePercentageError(**kwargs)`

Compute [Mean Absolute Percentage Error](#) (MAPE).

$$\text{MAPE} = \frac{1}{n} \sum_{i=1}^n \frac{|y_i - \hat{y}_i|}{\max(\epsilon, |y_i|)}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to `forward` and `update` the metric accepts the following input:

- **preds** ([Tensor](#)): Predictions from model
- **target** ([Tensor](#)): Ground truth values

As output of `forward` and `compute` the metric returns the following output:

- **mean\_abs\_percentage\_error** ([Tensor](#)): A tensor with the mean absolute percentage error over state

**Parameters**

**kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

---

**Note:** MAPE output is a non-negative floating point. Best result is `0.0`. But it is important to note that, bad predictions, can lead to arbitrarily large values. Especially when some `target` values are close to 0. This [MAPE implementation](#) returns a very large number instead of `inf`.

---

## Example

```
>>> from torch import tensor
>>> from torchmetrics.regression import MeanAbsolutePercentageError
>>> target = tensor([1, 10, 1e6])
>>> preds = tensor([0.9, 15, 1.2e6])
>>> mean_abs_percentage_error = MeanAbsolutePercentageError()
>>> mean_abs_percentage_error(preds, target)
tensor(0.2667)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

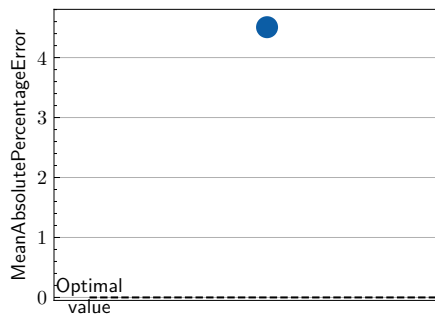
### Returns

Figure and Axes object

### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import MeanAbsolutePercentageError
>>> metric = MeanAbsolutePercentageError()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```

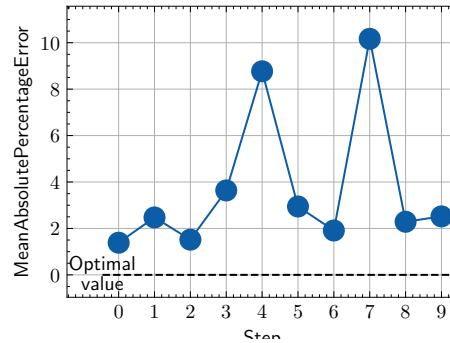


```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import MeanAbsolutePercentageError
>>> metric = MeanAbsolutePercentageError()
>>> values = []
```

(continues on next page)

(continued from previous page)

```
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```



## 1.92.2 Functional Interface

`torchmetrics.functional.mean_absolute_percentage_error(preds, target)`

Compute mean absolute percentage error.

### Parameters

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

### Return type

`Tensor`

### Returns

Tensor with MAPE

---

**Note:** The epsilon value is taken from [scikit-learn's](#) implementation of MAPE.

---

## Example

```
>>> from torchmetrics.functional.regression import mean_absolute_percentage_error
>>> target = torch.tensor([1, 10, 1e6])
>>> preds = torch.tensor([0.9, 15, 1.2e6])
>>> mean_absolute_percentage_error(preds, target)
tensor(0.2667)
```



## 1.93 Mean Squared Error (MSE)

### 1.93.1 Module Interface

**class** `torchmetrics.MeanSquaredError`(*squared=True, \*\*kwargs*)

Compute mean squared error (MSE).

$$\text{MSE} = \frac{1}{N} \sum_i^N (y_i - \hat{y}_i)^2$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to `forward` and `update` the metric accepts the following input:

- `preds` (`Tensor`): Predictions from model
- `target` (`Tensor`): Ground truth values

As output of `forward` and `compute` the metric returns the following output:

- `mean_squared_error` (`Tensor`): A tensor with the mean squared error

#### Parameters

- **squared** (`bool`) – If True returns MSE value, if False returns RMSE value.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torch import tensor
>>> from torchmetrics.regression import MeanSquaredError
>>> target = tensor([2.5, 5.0, 4.0, 8.0])
>>> preds = tensor([3.0, 5.0, 2.5, 7.0])
>>> mean_squared_error = MeanSquaredError()
>>> mean_squared_error(preds, target)
tensor(0.8750)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

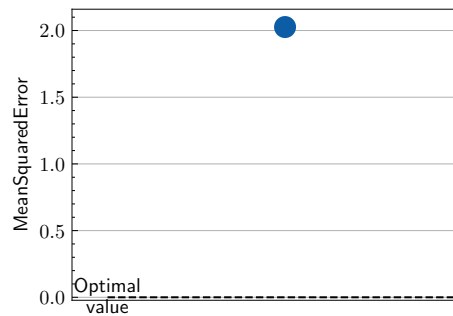
#### Returns

Figure and Axes object

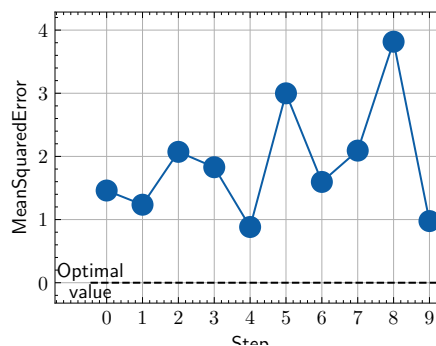
#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import MeanSquaredError
>>> metric = MeanSquaredError()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import MeanSquaredError
>>> metric = MeanSquaredError()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```



### 1.93.2 Functional Interface

`torchmetrics.functional.mean_squared_error(preds, target, squared=True)`

Compute mean squared error.

#### Parameters

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

- **squared** (`bool`) – returns RMSE value if set to False

**Return type**`Tensor`**Returns**

Tensor with MSE

**Example**

```
>>> from torchmetrics.functional.regression import mean_squared_error
>>> x = torch.tensor([0., 1, 2, 3])
>>> y = torch.tensor([0., 1, 2, 2])
>>> mean_squared_error(x, y)
tensor(0.2500)
```

## 1.94 Mean Squared Log Error (MSLE)

### 1.94.1 Module Interface

**class** `torchmetrics.MeanSquaredLogError` (\*\*kwargs)

Compute mean squared logarithmic error (MSLE).

$$\text{MSLE} = \frac{1}{N} \sum_i^N (\log_e(1 + y_i) - \log_e(1 + \hat{y}_i))^2$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to forward and update the metric accepts the following input:

- **preds** (`Tensor`): Predictions from model
- **target** (`Tensor`): Ground truth values

As output of forward and compute the metric returns the following output:

- **mean\_squared\_log\_error** (`Tensor`): A tensor with the mean squared log error

**Parameters**

**kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example**

```
>>> from torch import tensor
>>> from torchmetrics.regression import MeanSquaredLogError
>>> target = tensor([2.5, 5, 4, 8])
>>> preds = tensor([3, 5, 2.5, 7])
>>> mean_squared_log_error = MeanSquaredLogError()
>>> mean_squared_log_error(preds, target)
tensor(0.0397)
```

---

**Note:** Half precision is only support on GPU for this metric

---

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

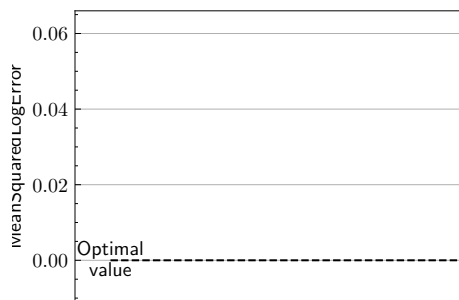
#### Returns

Figure and Axes object

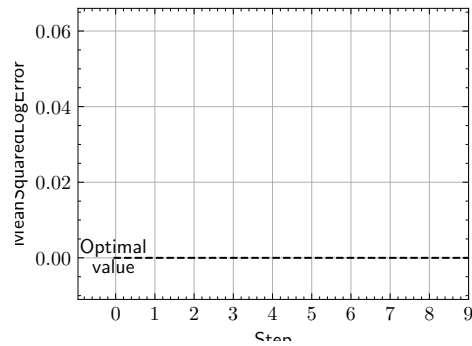
#### Raises

`ModuleNotFoundError` – If *matplotlib* is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import MeanSquaredLogError
>>> metric = MeanSquaredLogError()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import MeanSquaredLogError
>>> metric = MeanSquaredLogError()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```



## 1.94.2 Functional Interface

`torchmetrics.functional.mean_squared_log_error(preds, target)`

Compute mean squared log error.

### Parameters

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

### Return type

`Tensor`

### Returns

Tensor with RMSLE

### Example

```
>>> from torchmetrics.functional.regression import mean_squared_log_error
>>> x = torch.tensor([0., 1, 2, 3])
>>> y = torch.tensor([0., 1, 2, 2])
>>> mean_squared_log_error(x, y)
tensor(0.0207)
```

---

**Note:** Half precision is only support on GPU for this metric

---

## 1.95 Minkowski Distance

### 1.95.1 Module Interface

`class torchmetrics.MinkowskiDistance(p, **kwargs)`

Compute Minkowski Distance.

$$d_{\text{Minkowski}} = \left( \sum_i^N (|y_i - \hat{y}_i|^p)^{\frac{1}{p}} \right)$$

This metric can be seen as generalized version of the standard euclidean distance which corresponds to minkowski distance with  $p=2$ .

where

$y$  is a tensor of target values,

$\hat{y}$  is a tensor of predictions,  $p$  is a non-negative integer or floating-point number

#### Parameters

- **p** (`float`) – int or float larger than 1, exponent to which the difference between preds and target is to be raised
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torchmetrics.regression import MinkowskiDistance
>>> target = tensor([1.0, 2.8, 3.5, 4.5])
>>> preds = tensor([6.1, 2.11, 3.1, 5.6])
>>> minkowski_distance = MinkowskiDistance(3)
>>> minkowski_distance(preds, target)
tensor(5.1220)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

Figure and Axes object

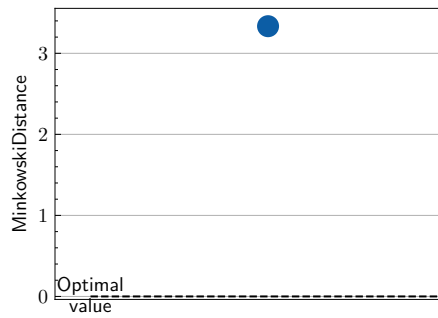
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import MinkowskiDistance
>>> metric = MinkowskiDistance(p=3)
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```

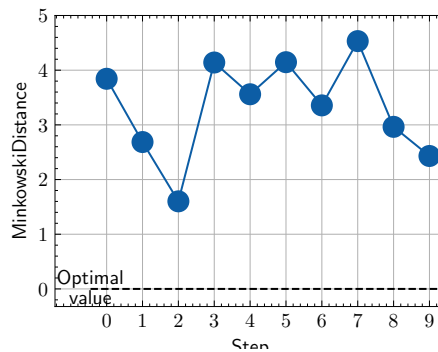
```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import MinkowskiDistance
>>> metric = MinkowskiDistance(p=3)
>>> values = []
>>> for _ in range(10):
```

(continues on next page)



(continued from previous page)

```
... values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```



## 1.95.2 Functional Interface

`torchmetrics.functional.minkowski_distance(preds, targets, p)`

Compute the [Minkowski distance](#).

$$d_{\text{Minkowski}} = \left( \sum_i^N (|y_i - \hat{y}_i|^p)^{\frac{1}{p}} \right)$$

This metric can be seen as generalized version of the standard euclidean distance which corresponds to minkowski distance with  $p=2$ .

### Parameters

- **preds** ([Tensor](#)) – estimated labels of type Tensor
- **targets** ([Tensor](#)) – ground truth labels of type Tensor
- **p** ([float](#)) – int or float larger than 1, exponent to which the difference between preds and target is to be raised

### Return type

[Tensor](#)

**Returns**

Tensor with the Minkowski distance

**Example**

```
>>> from torchmetrics.functional.regression import minkowski_distance
>>> x = torch.tensor([1.0, 2.8, 3.5, 4.5])
>>> y = torch.tensor([6.1, 2.11, 3.1, 5.6])
>>> minkowski_distance(x, y, p=3)
tensor(5.1220)
```

## 1.96 Pearson Corr. Coef.

### 1.96.1 Module Interface

**class** torchmetrics.PearsonCorrCoef(*num\_outputs=1, \*\*kwargs*)

Compute Pearson Correlation Coefficient.

$$P_{corr}(x, y) = \frac{cov(x, y)}{\sigma_x \sigma_y}$$

Where  $y$  is a tensor of target values, and  $x$  is a tensor of predictions.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): either single output float tensor with shape (N,) or multioutput float tensor of shape (N, d)
- **target** (**Tensor**): either single output tensor with shape (N,) or multioutput tensor of shape (N, d)

As output of forward and compute the metric returns the following output:

- **pearson** (**Tensor**): A tensor with the Pearson Correlation Coefficient

**Parameters**

- **num\_outputs** (**int**) – Number of outputs in multioutput setting
- **kwargs** (**Any**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (single output regression):**

```
>>> from torchmetrics.regression import PearsonCorrCoef
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> pearson = PearsonCorrCoef()
>>> pearson(preds, target)
tensor(0.9849)
```

**Example (multi output regression):**



```
>>> from torchmetrics.regression import PearsonCorrCoef
>>> target = torch.tensor([[3, -0.5], [2, 7]])
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> pearson = PearsonCorrCoef(num_outputs=2)
>>> pearson(preds, target)
tensor([1., 1.])
```

**plot**(*val=None*, *ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

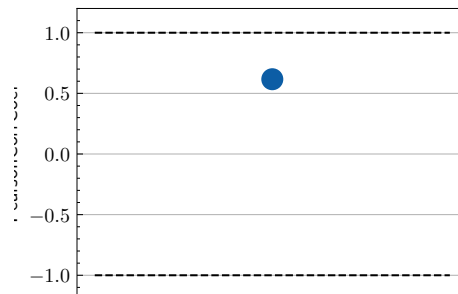
#### Returns

Figure and Axes object

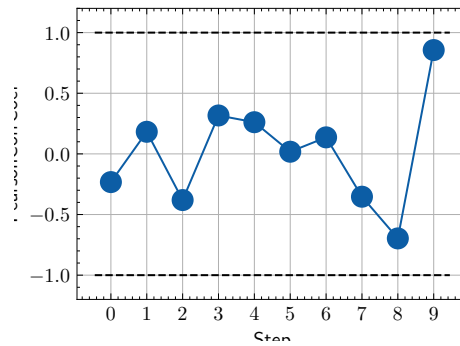
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import PearsonCorrCoef
>>> metric = PearsonCorrCoef()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import PearsonCorrCoef
>>> metric = PearsonCorrCoef()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```



## 1.96.2 Functional Interface

`torchmetrics.functional.pearson_corrcoef(preds, target)`

Compute pearson correlation coefficient.

### Parameters

- **preds** (`Tensor`) – estimated scores
- **target** (`Tensor`) – ground truth scores

### Return type

`Tensor`

Example (single output regression):

```
>>> from torchmetrics.functional.regression import pearson_corrcoef
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> pearson_corrcoef(preds, target)
tensor(0.9849)
```

Example (multi output regression):

```
>>> from torchmetrics.functional.regression import pearson_corrcoef
>>> target = torch.tensor([[3, -0.5], [2, 7]])
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> pearson_corrcoef(preds, target)
tensor([1., 1.])
```

## 1.97 R2 Score

### 1.97.1 Module Interface

`class torchmetrics.R2Score(num_outputs=1, adjusted=0, multioutput='uniform_average', **kwargs)`

Compute r2 score also known as `R2 Score_Coefficient Determination`.

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where  $SS_{res} = \sum_i (y_i - f(x_i))^2$  is the sum of residual squares, and  $SS_{tot} = \sum_i (y_i - \bar{y})^2$  is total sum of squares. Can also calculate adjusted r2 score given by

$$R_{adj}^2 = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

where the parameter  $k$  (the number of independent regressors) should be provided as the *adjusted* argument. The score is only proper defined when  $SS_{tot} \neq 0$ , which can happen for near constant targets. In this case a score of 0 is returned. By definition the score is bounded between 0 and 1, where 1 corresponds to the predictions exactly matching the targets.

As input to forward and update the metric accepts the following input:

- **preds** ([Tensor](#)): Predictions from model in float tensor with shape (N,) or (N, M) (multioutput)
- **target** ([Tensor](#)): Ground truth values in float tensor with shape (N,) or (N, M) (multioutput)

As output of forward and compute the metric returns the following output:

- **r2score** ([Tensor](#)): A tensor with the r2 score(s)

In the case of multioutput, as default the variances will be uniformly averaged over the additional dimensions. Please see argument `multioutput` for changing this behavior.

#### Parameters

- **num\_outputs** ([int](#)) – Number of outputs in multioutput setting
- **adjusted** ([int](#)) – number of independent regressors for calculating adjusted r2 score.
- **multioutput** ([str](#)) – Defines aggregation in the case of multiple output scores. Can be one of the following strings:
  - 'raw\_values' returns full set of scores
  - 'uniform\_average' scores are uniformly averaged
  - 'variance\_weighted' scores are weighted by their individual variances
- **kwargs** ([Any](#)) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Raises

- **ValueError** – If adjusted parameter is not an integer larger or equal to 0.
- **ValueError** – If multioutput is not one of "raw\_values", "uniform\_average" or "variance\_weighted".

#### Example

```
>>> from torchmetrics.regression import R2Score
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> r2score = R2Score()
>>> r2score(preds, target)
tensor(0.9486)
```

```
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> r2score = R2Score(num_outputs=2, multioutput='raw_values')
>>> r2score(preds, target)
tensor([0.9654, 0.9082])
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

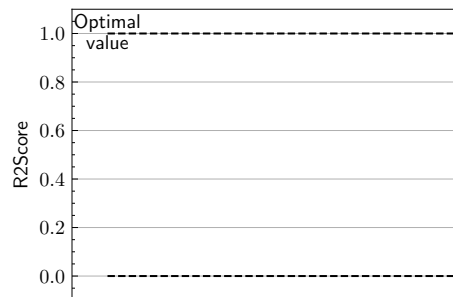
#### Returns

Figure and Axes object

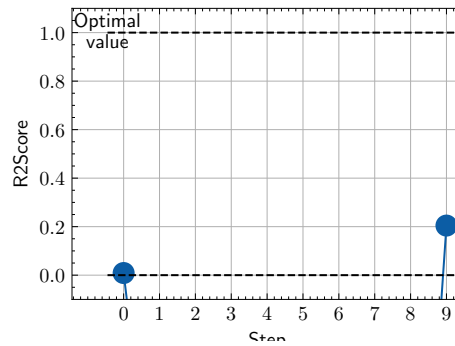
#### Raises

`ModuleNotFoundError` – If `matplotlib` is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import R2Score
>>> metric = R2Score()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import R2Score
>>> metric = R2Score()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```



### 1.97.2 Functional Interface

`torchmetrics.functional.r2_score(preds, target, adjusted=0, multioutput='uniform_average')`

Compute r2 score also known as **R2 Score\_Coefficient Determination**.

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where  $SS_{res} = \sum_i (y_i - f(x_i))^2$  is the sum of residual squares, and  $SS_{tot} = \sum_i (y_i - \bar{y})^2$  is total sum of squares. Can also calculate adjusted r2 score given by

$$R^2_{adj} = 1 - \frac{(1 - R^2)(n - 1)}{n - k - 1}$$

where the parameter  $k$  (the number of independent regressors) should be provided as the `adjusted` argument.

#### Parameters

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels
- **adjusted** (`int`) – number of independent regressors for calculating adjusted r2 score.
- **multioutput** (`str`) – Defines aggregation in the case of multiple output scores. Can be one of the following strings:
  - 'raw\_values' returns full set of scores
  - 'uniform\_average' scores are uniformly averaged
  - 'variance\_weighted' scores are weighted by their individual variances

#### Raises

- **ValueError** – If both preds and targets are not 1D or 2D tensors.
- **ValueError** – If `len(preds)` is less than 2 since at least 2 samples are needed to calculate r2 score.
- **ValueError** – If `multioutput` is not one of `raw_values`, `uniform_average` or `variance_weighted`.
- **ValueError** – If `adjusted` is not an integer greater than 0.

#### Return type

`Tensor`

### Example

```
>>> from torchmetrics.functional.regression import r2_score
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> r2_score(preds, target)
tensor(0.9486)
```

```
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> r2_score(preds, target, multioutput='raw_values')
tensor([0.9654, 0.9082])
```

## 1.98 Relative Squared Error (RSE)

### 1.98.1 Module Interface

**class** torchmetrics.RelativeSquaredError(num\_outputs=1, squared=True, \*\*kwargs)

Computes the relative squared error (RSE).

$$\text{RSE} = \frac{\sum_i^N (y_i - \hat{y}_i)^2}{\sum_i^N (y_i - \bar{y})^2}$$

Where  $y$  is a tensor of target values with mean  $\bar{y}$ , and  $\hat{y}$  is a tensor of predictions.

If num\_outputs > 1, the returned value is averaged over all the outputs.

As input to forward and update the metric accepts the following input:

- preds (**Tensor**): Predictions from model in float tensor with shape (N,) or (N, M) (multioutput)
- target (**Tensor**): Ground truth values in float tensor with shape (N,) or (N, M) (multioutput)

As output of forward and compute the metric returns the following output:

- rse (**Tensor**): A tensor with the RSE score(s)

#### Parameters

- num\_outputs (**int**) – Number of outputs in multioutput setting
- squared (**bool**) – If True returns RSE value, if False returns RRSE value.
- kwargs (**Any**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torchmetrics.regression import RelativeSquaredError
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> relative_squared_error = RelativeSquaredError()
>>> relative_squared_error(preds, target)
tensor(0.0514)
```

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

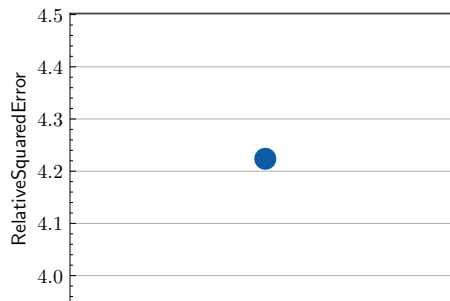
#### Returns

Figure and Axes object

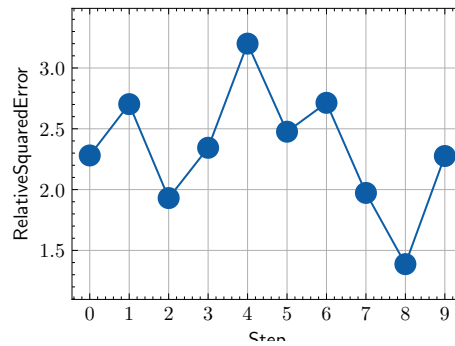
#### Raises

`ModuleNotFoundError` – If `matplotlib` is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import RelativeSquaredError
>>> metric = RelativeSquaredError()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import RelativeSquaredError
>>> metric = RelativeSquaredError()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```



## 1.98.2 Functional Interface

`torchmetrics.functional.relative_squared_error(preds, target, squared=True)`

Computes the relative squared error (RSE).

$$\text{RSE} = \frac{\sum_i^N (y_i - \hat{y}_i)^2}{\sum_i^N (y_i - \bar{y})^2}$$

Where  $y$  is a tensor of target values with mean  $\bar{y}$ , and  $\hat{y}$  is a tensor of predictions.

If *preds* and *targets* are 2D tensors, the RSE is averaged over the second dim.

### Parameters

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels
- **squared** (`bool`) – returns RRSE value if set to False

### Return type

`Tensor`

### Returns

Tensor with RSE

### Example

```
>>> from torchmetrics.functional.regression import relative_squared_error
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> relative_squared_error(preds, target)
tensor(0.0514)
```



## 1.99 Spearman Corr. Coef.

### 1.99.1 Module Interface

**class** torchmetrics.SpearmanCorrCoef(*num\_outputs=1, \*\*kwargs*)

Compute spearman's rank correlation coefficient.

where  $rg_x$  and  $rg_y$  are the rank associated to the variables  $x$  and  $y$ . Spearman's correlations coefficient corresponds to the standard pearson's correlation coefficient calculated on the rank variables.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): Predictions from model in float tensor with shape (N,d)
- **target** (**Tensor**): Ground truth values in float tensor with shape (N,d)

As output of forward and compute the metric returns the following output:

- **spearman** (**Tensor**): A tensor with the spearman correlation(s)

#### Parameters

- **num\_outputs** (**int**) – Number of outputs in multioutput setting
- **kwargs** (**Any**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

**Example (single output regression):**

```
>>> from torch import tensor
>>> from torchmetrics.regression import SpearmanCorrCoef
>>> target = tensor([3, -0.5, 2, 7])
>>> preds = tensor([2.5, 0.0, 2, 8])
>>> spearman = SpearmanCorrCoef()
>>> spearman(preds, target)
tensor(1.0000)
```

**Example (multi output regression):**

```
>>> from torchmetrics.regression import SpearmanCorrCoef
>>> target = tensor([[3, -0.5], [2, 7]])
>>> preds = tensor([[2.5, 0.0], [2, 8]])
>>> spearman = SpearmanCorrCoef(num_outputs=2)
>>> spearman(preds, target)
tensor([1.0000, 1.0000])
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (**Union**[**Tensor**, **Sequence**[**Tensor**], **None**]) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (**Optional**[**Axes**]) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

**Tuple**[**Figure**, **Union**[**Axes**, **ndarray**]]

**Returns**

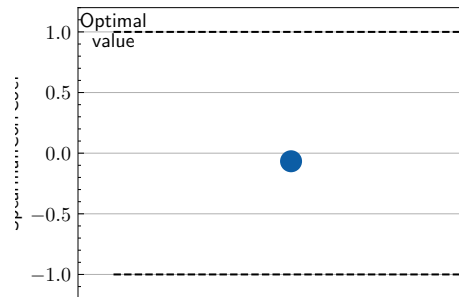
Figure and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```

>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import SpearmanCorrCoef
>>> metric = SpearmanCorrCoef()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()

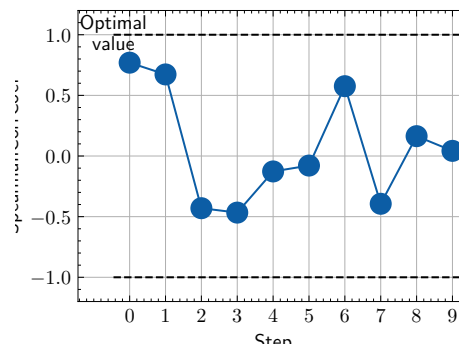
```



```

>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import SpearmanCorrCoef
>>> metric = SpearmanCorrCoef()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)

```



## 1.99.2 Functional Interface

`torchmetrics.functional.spearman_corrcoef(preds, target)`

Compute spearman's rank correlation coefficient.

where  $rg_x$  and  $rg_y$  are the rank associated to the variables  $x$  and  $y$ . Spearman's correlations coefficient corresponds to the standard pearson's correlation coefficient calculated on the rank variables.

### Parameters

- **preds** (`Tensor`) – estimated scores
- **target** (`Tensor`) – ground truth scores

### Return type

`Tensor`

Example (single output regression):

```
>>> from torchmetrics.functional.regression import spearman_corrcoef
>>> target = torch.tensor([3, -0.5, 2, 7])
>>> preds = torch.tensor([2.5, 0.0, 2, 8])
>>> spearman_corrcoef(preds, target)
tensor(1.0000)
```

Example (multi output regression):

```
>>> from torchmetrics.functional.regression import spearman_corrcoef
>>> target = torch.tensor([[3, -0.5], [2, 7]])
>>> preds = torch.tensor([[2.5, 0.0], [2, 8]])
>>> spearman_corrcoef(preds, target)
tensor([1.0000, 1.0000])
```

## 1.100 Symmetric Mean Absolute Percentage Error (SMAPE)

### 1.100.1 Module Interface

`class torchmetrics.SymmetricMeanAbsolutePercentageError(**kwargs)`

Compute symmetric mean absolute percentage error (SMAPE).

$$\text{SMAPE} = \frac{2}{n} \sum_1^n \frac{|y_i - \hat{y}_i|}{\max(|y_i| + |\hat{y}_i|, \epsilon)}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`Tensor`): Predictions from model
- **target** (`Tensor`): Ground truth values

As output of `forward` and `compute` the metric returns the following output:

- **smape** (`Tensor`): A tensor with non-negative floating point smaпе value between 0 and 1

### Parameters

**kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

## Example

```
>>> from torchmetrics.regression import SymmetricMeanAbsolutePercentageError
>>> target = tensor([1, 10, 1e6])
>>> preds = tensor([0.9, 15, 1.2e6])
>>> smape = SymmetricMeanAbsolutePercentageError()
>>> smape(preds, target)
tensor(0.2290)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

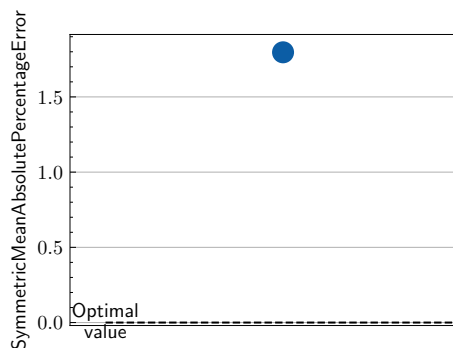
### Returns

Figure and Axes object

### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import SymmetricMeanAbsolutePercentageError
>>> metric = SymmetricMeanAbsolutePercentageError()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```

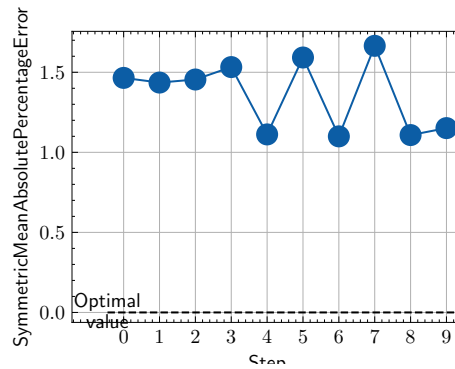


```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import SymmetricMeanAbsolutePercentageError
>>> metric = SymmetricMeanAbsolutePercentageError()
>>> values = []
>>> for _ in range(10):
```

(continues on next page)

(continued from previous page)

```
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```



### 1.100.2 Functional Interface

`torchmetrics.functional.symmetric_mean_absolute_percentage_error(preds, target)`

Compute symmetric mean absolute percentage error (SMAPE).

$$\text{SMAPE} = \frac{2}{n} \sum_1^n \frac{|y_i - \hat{y}_i|}{\max(|y_i| + |\hat{y}_i|, \epsilon)}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

#### Parameters

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

#### Return type

`Tensor`

#### Returns

Tensor with SMAPE.

#### Example

```
>>> from torchmetrics.functional.regression import symmetric_mean_absolute_
    percentage_error
>>> target = torch.tensor([1, 10, 1e6])
>>> preds = torch.tensor([0.9, 15, 1.2e6])
>>> symmetric_mean_absolute_percentage_error(preds, target)
tensor(0.2290)
```

## 1.101 Tweedie Deviance Score

### 1.101.1 Module Interface

**class** torchmetrics.**TweedieDevianceScore**(power=0.0, \*\*kwargs)

Compute the *Tweedie Deviance Score*.

$$\text{deviance\_score}(\hat{y}, y) = \begin{cases} (\hat{y} - y)^2, & \text{for } p = 0 \\ 2 * (y * \log(\frac{y}{\hat{y}}) + \hat{y} - y), & \text{for } p = 1 \\ 2 * (\log(\frac{\hat{y}}{y}) + \frac{y}{\hat{y}} - 1), & \text{for } p = 2 \\ 2 * (\frac{(\max(y, 0))^{2-p}}{(1-p)(2-p)} - \frac{y(\hat{y})^{1-p}}{1-p} + \frac{(\hat{y})^{2-p}}{2-p}), & \text{otherwise} \end{cases}$$

where  $y$  is a tensor of targets values,  $\hat{y}$  is a tensor of predictions, and  $p$  is the *power*.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (**Tensor**): Predicted float tensor with shape (N, ...)
- **target** (**Tensor**): Ground truth float tensor with shape (N, ...)

As output of `forward` and `compute` the metric returns the following output:

- **deviance\_score** (**Tensor**): A tensor with the deviance score

#### Parameters

- **power** (**float**) –
  - power < 0 : Extreme stable distribution. (Requires: preds > 0.)
  - power = 0 : Normal distribution. (Requires: targets and preds can be any real numbers.)
  - power = 1 : Poisson distribution. (Requires: targets >= 0 and y\_pred > 0.)
  - 1 < p < 2 : Compound Poisson distribution. (Requires: targets >= 0 and preds > 0.)
  - power = 2 : Gamma distribution. (Requires: targets > 0 and preds > 0.)
  - power = 3 : Inverse Gaussian distribution. (Requires: targets > 0 and preds > 0.)
  - otherwise : Positive stable distribution. (Requires: targets > 0 and preds > 0.)
- **kwargs** (**Any**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> from torchmetrics.regression import TweedieDevianceScore
>>> targets = torch.tensor([1.0, 2.0, 3.0, 4.0])
>>> preds = torch.tensor([4.0, 3.0, 2.0, 1.0])
>>> deviance_score = TweedieDevianceScore(power=2)
>>> deviance_score(preds, targets)
tensor(1.2083)
```

**plot**(val=None, ax=None)

Plot a single or multiple values from the metric.

#### Parameters

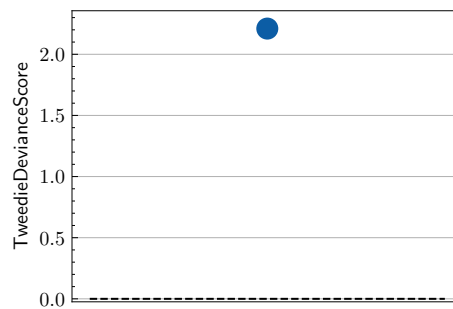
- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**`Tuple[Figure, Union\[Axes, ndarray\]]`**Returns**

Figure and Axes object

**Raises****`ModuleNotFoundError`** – If `matplotlib` is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import TweedieDevianceScore
>>> metric = TweedieDevianceScore()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```

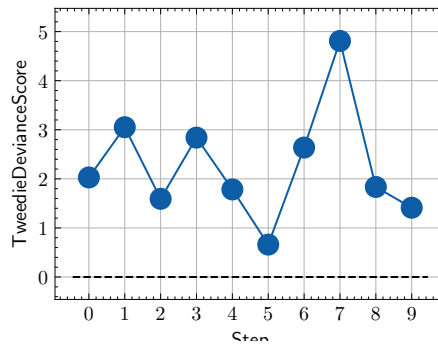


```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import TweedieDevianceScore
>>> metric = TweedieDevianceScore()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```

## 1.101.2 Functional Interface

`torchmetrics.functional.tweedie_deviance_score(preds, targets, power=0.0)`Compute the *Tweedie Deviance Score*.

$$\text{deviance\_score}(\hat{y}, y) = \begin{cases} (\hat{y} - y)^2, & \text{for } p = 0 \\ 2 * (y * \log(\frac{\hat{y}}{y}) + \hat{y} - y), & \text{for } p = 1 \\ 2 * (\log(\frac{\hat{y}}{y}) + \frac{\hat{y}}{y} - 1), & \text{for } p = 2 \\ 2 * (\frac{(\max(y, 0))^{2-p}}{(1-p)(2-p)} - \frac{y(\hat{y})^{1-p}}{1-p} + \frac{(\hat{y})^{2-p}}{2-p}), & \text{otherwise} \end{cases}$$



where  $y$  is a tensor of targets values,  $\hat{y}$  is a tensor of predictions, and  $p$  is the *power*.

#### Parameters

- **preds** ([Tensor](#)) – Predicted tensor with shape  $(N, \dots)$
- **targets** ([Tensor](#)) – Ground truth tensor with shape  $(N, \dots)$
- **power** ([float](#)) –
  - $power < 0$  : Extreme stable distribution. (Requires:  $preds > 0$ .)
  - $power = 0$  : Normal distribution. (Requires: targets and preds can be any real numbers.)
  - $power = 1$  : Poisson distribution. (Requires: targets  $\geq 0$  and  $y\_pred > 0$ .)
  - $1 < p < 2$  : Compound Poisson distribution. (Requires: targets  $\geq 0$  and  $preds > 0$ .)
  - $power = 2$  : Gamma distribution. (Requires: targets  $> 0$  and  $preds > 0$ .)
  - $power = 3$  : Inverse Gaussian distribution. (Requires: targets  $> 0$  and  $preds > 0$ .)
  - *otherwise* : Positive stable distribution. (Requires: targets  $> 0$  and  $preds > 0$ .)

#### Return type

[Tensor](#)

#### Example

```
>>> from torchmetrics.functional.regression import tweedie_deviance_score
>>> targets = torch.tensor([1.0, 2.0, 3.0, 4.0])
>>> preds = torch.tensor([4.0, 3.0, 2.0, 1.0])
>>> tweedie_deviance_score(preds, targets, power=2)
tensor(1.2083)
```



## 1.102 Weighted MAPE

### 1.102.1 Module Interface

**class** `torchmetrics.WeightedMeanAbsolutePercentageError(**kwargs)`

Compute weighted mean absolute percentage error (**WMAPE**).

The output of WMAPE metric is a non-negative floating point, where the optimal value is 0. It computes as:

$$\text{WMAPE} = \frac{\sum_{t=1}^n |y_t - \hat{y}_t|}{\sum_{t=1}^n |y_t|}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

As input to `forward` and `update` the metric accepts the following input:

- **preds** (`Tensor`): Predictions from model
- **target** (`Tensor`): Ground truth float tensor with shape (N, d)

As output of `forward` and `compute` the metric returns the following output:

- **wmape** (`Tensor`): A tensor with non-negative floating point wmape value between 0 and 1

#### Parameters

**kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> preds = torch.randn(20,)
>>> target = torch.randn(20,)
>>> wmape = WeightedMeanAbsolutePercentageError()
>>> wmape(preds, target)
tensor(1.3967)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

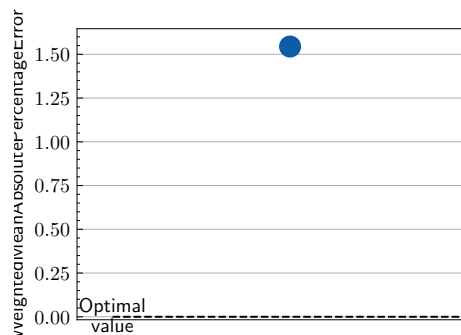
#### Returns

Figure and Axes object

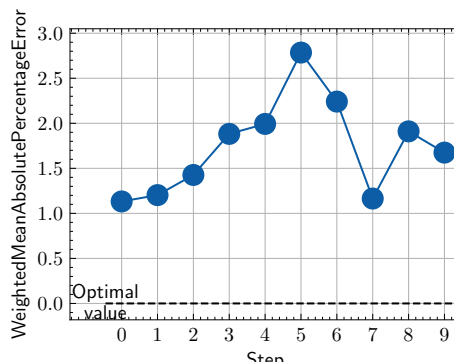
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> from torch import randn
>>> # Example plotting a single value
>>> from torchmetrics.regression import WeightedMeanAbsolutePercentageError
>>> metric = WeightedMeanAbsolutePercentageError()
>>> metric.update(randn(10,), randn(10,))
>>> fig_, ax_ = metric.plot()
```



```
>>> from torch import randn
>>> # Example plotting multiple values
>>> from torchmetrics.regression import WeightedMeanAbsolutePercentageError
>>> metric = WeightedMeanAbsolutePercentageError()
>>> values = []
>>> for _ in range(10):
...     values.append(metric(randn(10,), randn(10,)))
>>> fig, ax = metric.plot(values)
```



## 1.102.2 Functional Interface

`torchmetrics.functional.weighted_mean_absolute_percentage_error(preds, target)`

Compute weighted mean absolute percentage error (WMAPE).

The output of WMAPE metric is a non-negative floating point, where the optimal value is 0. It is computed as:

$$\text{WMAPE} = \frac{\sum_{t=1}^n |y_t - \hat{y}_t|}{\sum_{t=1}^n |y_t|}$$

Where  $y$  is a tensor of target values, and  $\hat{y}$  is a tensor of predictions.

**Parameters**

- **preds** (`Tensor`) – estimated labels
- **target** (`Tensor`) – ground truth labels

**Return type**`Tensor`**Returns**

Tensor with WMAPE.

**Example**

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> preds = torch.randn(20,)
>>> target = torch.randn(20,)
>>> weighted_mean_absolute_percentage_error(preds, target)
tensor(1.3967)
```

## 1.103 Retrieval Fall-Out

### 1.103.1 Module Interface

`torchmetrics.RetrievalFallOut`alias of `_RetrievalFallOut`

### 1.103.2 Functional Interface

`torchmetrics.functional.retrieval_fall_out(preds, target, top_k=None)`

Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _retrieval_fall_out(preds, target, top_k=2)
tensor(1.)
```

## 1.104 Retrieval Hit Rate

### 1.104.1 Module Interface

`torchmetrics.RetrievalHitRate`  
alias of `_RetrievalHitRate`

### 1.104.2 Functional Interface

`torchmetrics.functional.retrieval_hit_rate(preds, target, top_k=None)`  
Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _retrieval_hit_rate(preds, target, top_k=2)
tensor(1.)
```

## 1.105 Retrieval Mean Average Precision (MAP)

### 1.105.1 Module Interface

`torchmetrics.RetrievalMAP`  
alias of `_RetrievalMAP`

### 1.105.2 Functional Interface

`torchmetrics.functional.retrieval_average_precision(preds, target, top_k=None)`  
Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _retrieval_average_precision(preds, target)
tensor(0.8333)
```

## 1.106 Retrieval Mean Reciprocal Rank (MRR)

### 1.106.1 Module Interface

`torchmetrics.RetrievalMRR`  
alias of `_RetrievalMRR`

### 1.106.2 Functional Interface

`torchmetrics.functional.retrieval_reciprocal_rank(preds, target)`  
Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([False, True, False])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _retrieval_reciprocal_rank(preds, target)
tensor(0.5000)
```

## 1.107 Retrieval Normalized DCG

### 1.107.1 Module Interface

`torchmetrics.RetrievalNormalizedDCG`  
alias of `_RetrievalNormalizedDCG`

### 1.107.2 Functional Interface

`torchmetrics.functional.retrieval_normalized_dcg(preds, target, top_k=None)`  
Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([.1, .2, .3, 4, 70])
>>> target = tensor([10, 0, 0, 1, 5])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _retrieval_normalized_dcg(preds, target)
tensor(0.6957)
```

## 1.108 Retrieval Precision

### 1.108.1 Module Interface

`torchmetrics.RetrievalPrecision`  
alias of `_RetrievalPrecision`

### 1.108.2 Functional Interface

`torchmetrics.functional.retrieval_precision(preds, target, top_k=None, adaptive_k=False)`  
Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _retrieval_precision(preds, target, top_k=2)
tensor(0.5000)
```

## 1.109 Precision Recall Curve

### 1.109.1 Module Interface

`torchmetrics.RetrievalPrecisionRecallCurve`  
alias of `_RetrievalPrecisionRecallCurve`

### 1.109.2 Functional Interface

`torchmetrics.functional.retrieval_precision_recall_curve(preds, target, max_k=None, adaptive_k=False)`  
Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
>>> precisions, recalls, top_k = _retrieval_precision_recall_curve(preds, target,
↪ max_k=2)
>>> precisions
tensor([1.0000, 0.5000])
>>> recalls
tensor([0.5000, 0.5000])
:rtype: :py:data:`~typing.Tuple`[:py:class:`~torch.Tensor`, :py:class:`~torch.
↪ Tensor`, :py:class:`~torch.Tensor`]
```

```
>>> top_k
tensor([1, 2])
```

## 1.110 Retrieval R-Precision

### 1.110.1 Module Interface

`torchmetrics.RetrievalRPrecision`  
alias of `_RetrievalRPrecision`

### 1.110.2 Functional Interface

`torchmetrics.functional.retrieval_r_precision(preds, target)`  
Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _retrieval_r_precision(preds, target)
tensor(0.5000)
```

## 1.111 Retrieval Recall

### 1.111.1 Module Interface

`torchmetrics.RetrievalRecall`  
alias of `_RetrievalRecall`

### 1.111.2 Functional Interface

`torchmetrics.functional.retrieval_recall(preds, target, top_k=None)`  
Wrapper for deprecated import.

```
>>> from torch import tensor
>>> preds = tensor([0.2, 0.3, 0.5])
>>> target = tensor([True, False, True])
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _retrieval_recall(preds, target, top_k=2)
tensor(0.5000)
```

## 1.112 BERT Score

### 1.112.1 Module Interface

```
class torchmetrics.text.bert.BERTScore(model_name_or_path=None, num_layers=None,
                                       all_layers=False, model=None, user_tokenizer=None,
                                       user_forward_fn=None, verbose=False, idf=False, device=None,
                                       max_length=512, batch_size=64, num_threads=0,
                                       return_hash=False, lang='en', rescale_with_baseline=False,
                                       baseline_path=None, baseline_url=None, **kwargs)
```

[Bert\\_score Evaluating Text Generation](#) for measuring text similarity.

BERT leverages the pre-trained contextual embeddings from BERT and matches words in candidate and reference sentences by cosine similarity. It has been shown to correlate with human judgment on sentence-level and system-level evaluation. Moreover, BERTScore computes precision, recall, and F1 measure, which can be useful for evaluating different language generation tasks. This implementation follows the original implementation from [BERT\\_score](#).

As input to `forward` and `update` the metric accepts the following input:

- **preds** (List): An iterable of predicted sentences
- **target** (List): An iterable of reference sentences

As output of `forward` and `compute` the metric returns the following output:

- **score** (Dict): A dictionary containing the keys `precision`, `recall` and `f1` with corresponding values

#### Parameters

- **preds** – An iterable of predicted sentences.
- **target** – An iterable of target sentences.
- **model\_type** – A name or a model path used to load `transformers` pretrained model.
- **num\_layers** (Optional[int]) – A layer of representation to use.
- **all\_layers** (bool) – An indication of whether the representation from all model's layers should be used. If `all_layers=True`, the argument `num_layers` is ignored.
- **model** (Optional[Module]) – A user's own model. Must be of `torch.nn.Module` instance.
- **user\_tokenizer** (Optional[Any]) – A user's own tokenizer used with the own model. This must be an instance with the `__call__` method. This method must take an iterable of sentences (`List[str]`) and must return a python dictionary containing `"input_ids"` and `"attention_mask"` represented by `Tensor`. It is up to the user's model of whether `"input_ids"` is a `Tensor` of input ids or embedding vectors. This tokenizer must prepend an equivalent of [CLS] token and append an equivalent of [SEP] token as `transformers` tokenizer does.
- **user\_forward\_fn** (Optional[Callable[[Module, Dict[str, Tensor]], Tensor]]) – A user's own forward function used in a combination with `user_model`. This function must take `user_model` and a python dictionary of containing `"input_ids"` and `"attention_mask"` represented by `Tensor` as an input and return the model's output represented by the single `Tensor`.
- **verbose** (bool) – An indication of whether a progress bar to be displayed during the embeddings' calculation.
- **idf** (bool) – An indication whether normalization using inverse document frequencies should be used.



- **device** (`Union[str, device, None]`) – A device to be used for calculation.
- **max\_length** (`int`) – A maximum length of input sequences. Sequences longer than `max_length` are to be trimmed.
- **batch\_size** (`int`) – A batch size used for model processing.
- **num\_threads** (`int`) – A number of threads to use for a dataloader.
- **return\_hash** (`bool`) – An indication of whether the corresponding `hash_code` should be returned.
- **lang** (`str`) – A language of input sentences.
- **rescale\_with\_baseline** (`bool`) – An indication of whether `bertscore` should be rescaled with a pre-computed baseline. When a pretrained model from `transformers` model is used, the corresponding baseline is downloaded from the original `bert-score` package from `BERT_score` if available. In other cases, please specify a path to the baseline csv/tsv file, which must follow the formatting of the files from `BERT_score`.
- **baseline\_path** (`Optional[str]`) – A path to the user's own local csv/tsv file with the baseline scale.
- **baseline\_url** (`Optional[str]`) – A url path to the user's own csv/tsv file with the baseline scale.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from pprint import pprint
>>> from torchmetrics.text.bert import BERTScore
>>> preds = ["hello there", "general kenobi"]
>>> target = ["hello there", "master kenobi"]
>>> bertscore = BERTScore()
>>> pprint(bertscore(preds, target))
{'f1': tensor([1.0000, 0.9961]), 'precision': tensor([1.0000, 0.9961]), 'recall':
→ tensor([1.0000, 0.9961])}
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

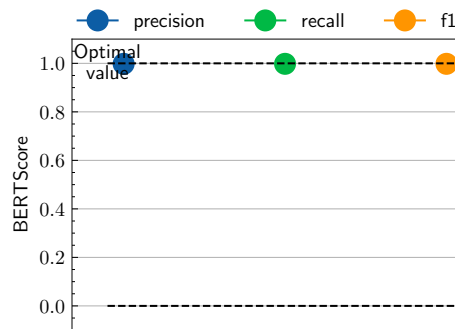
#### Returns

Figure and Axes object

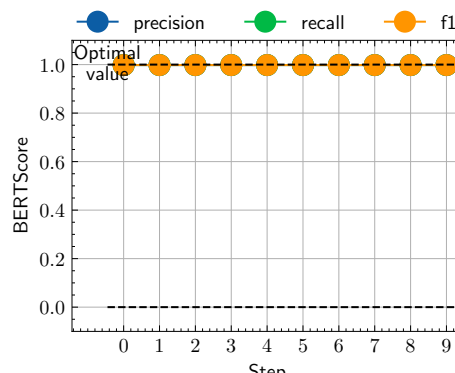
#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> # Example plotting a single value
>>> from torchmetrics.text.bert import BERTScore
>>> preds = ["hello there", "general kenobi"]
>>> target = ["hello there", "master kenobi"]
>>> metric = BERTScore()
>>> metric.update(preds, target)
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torch import tensor
>>> from torchmetrics.text.bert import BERTScore
>>> preds = ["hello there", "general kenobi"]
>>> target = ["hello there", "master kenobi"]
>>> metric = BERTScore()
>>> values = []
>>> for _ in range(10):
...     val = metric(preds, target)
...     val = {k: tensor(v).mean() for k,v in val.items()} # convert into
...     single value per key
...     values.append(val)
>>> fig_, ax_ = metric.plot(values)
```



### 1.112.2 Functional Interface

```
torchmetrics.functional.text.bert.bert_score(preds, target, model_name_or_path=None,
                                             num_layers=None, all_layers=False, model=None,
                                             user_tokenizer=None, user_forward_fn=None,
                                             verbose=False, idf=False, device=None,
                                             max_length=512, batch_size=64, num_threads=0,
                                             return_hash=False, lang='en',
                                             rescale_with_baseline=False, baseline_path=None,
                                             baseline_url=None)
```

[Bert\\_score](#) Evaluating Text Generation for text similarity matching.

This metric leverages the pre-trained contextual embeddings from BERT and matches words in candidate and reference sentences by cosine similarity. It has been shown to correlate with human judgment on sentence-level and system-level evaluation. Moreover, BERTScore computes precision, recall, and F1 measure, which can be useful for evaluating different language generation tasks.

This implementation follows the original implementation from [BERT\\_score](#).

#### Parameters

- **preds** (`Union[str, Sequence[str], Dict[str, Tensor]]`) – Either an iterable of predicted sentences or a `Dict[input_ids, attention_mask]`.
- **target** (`Union[str, Sequence[str], Dict[str, Tensor]]`) – Either an iterable of target sentences or a `Dict[input_ids, attention_mask]`.
- **model\_name\_or\_path** (`Optional[str]`) – A name or a model path used to load transformers pretrained model.
- **num\_layers** (`Optional[int]`) – A layer of representation to use.
- **all\_layers** (`bool`) – An indication of whether the representation from all model's layers should be used. If `all_layers = True`, the argument `num_layers` is ignored.
- **model** (`Optional[Module]`) – A user's own model.
- **user\_tokenizer** (`Optional[Any]`) – A user's own tokenizer used with the own model. This must be an instance with the `__call__` method. This method must take an iterable of sentences (`List[str]`) and must return a python dictionary containing "input\_ids" and "attention\_mask" represented by `Tensor`. It is up to the user's model of whether "input\_ids" is a `Tensor` of input ids or embedding vectors. his tokenizer must prepend an equivalent of [CLS] token and append an equivalent of [SEP] token as *transformers* tokenizer does.
- **user\_forward\_fn** (`Optional[Callable[[Module, Dict[str, Tensor]], Tensor]]`) – A user's own forward function used in a combination with `user_model`. This function must take `user_model` and a python dictionary of containing "input\_ids" and "attention\_mask" represented by `Tensor` as an input and return the model's output represented by the single `Tensor`.
- **verbose** (`bool`) – An indication of whether a progress bar to be displayed during the embeddings' calculation.
- **idf** (`bool`) – An indication of whether normalization using inverse document frequencies should be used.
- **device** (`Union[str, device, None]`) – A device to be used for calculation.
- **max\_length** (`int`) – A maximum length of input sequences. Sequences longer than `max_length` are to be trimmed.

- **batch\_size** (`int`) – A batch size used for model processing.
- **num\_threads** (`int`) – A number of threads to use for a dataloader.
- **return\_hash** (`bool`) – An indication of whether the corresponding hash\_code should be returned.
- **lang** (`str`) – A language of input sentences. It is used when the scores are rescaled with a baseline.
- **rescale\_with\_baseline** (`bool`) – An indication of whether bertscore should be rescaled with a pre-computed baseline. When a pretrained model from `transformers` model is used, the corresponding baseline is downloaded from the original `bert-score` package from `BERT_score` if available. In other cases, please specify a path to the baseline csv/tsv file, which must follow the formatting of the files from `BERT_score`
- **baseline\_path** (`Optional[str]`) – A path to the user's own local csv/tsv file with the baseline scale.
- **baseline\_url** (`Optional[str]`) – A url path to the user's own csv/tsv file with the baseline scale.

**Return type**`Dict[str, Union[Tensor, List[float], str]]`**Returns**

Python dictionary containing the keys `precision`, `recall` and `f1` with corresponding values.

**Raises**

- **ValueError** – If `len(preds) != len(target)`.
- **ModuleNotFoundError** – If `tqdm` package is required and not installed.
- **ModuleNotFoundError** – If `transformers` package is required and not installed.
- **ValueError** – If `num_layer` is larger than the number of the model layers.
- **ValueError** – If invalid input is provided.

**Example**

```
>>> from pprint import pprint
>>> from torchmetrics.functional.text.bert import bert_score
>>> preds = ["hello there", "general kenobi"]
>>> target = ["hello there", "master kenobi"]
>>> pprint(bert_score(preds, target))
{'f1': tensor([1.0000, 0.9961]), 'precision': tensor([1.0000, 0.9961]), 'recall':
→ tensor([1.0000, 0.9961])}
```

## 1.113 BLEU Score

### 1.113.1 Module Interface

`torchmetrics.BLEUScore`  
alias of `_BLEUScore`

### 1.113.2 Functional Interface

`torchmetrics.functional.bleu_score(preds, target, n_gram=4, smooth=False, weights=None)`  
Wrapper for deprecated import.

```
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _bleu_score(preds, target)
tensor(0.7598)
```

## 1.114 Char Error Rate

### 1.114.1 Module Interface

`torchmetrics.CharErrorRate`  
alias of `_CharErrorRate`

### 1.114.2 Functional Interface

`torchmetrics.functional.char_error_rate(preds, target)`  
Wrapper for deprecated import.

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _char_error_rate(preds=preds, target=target)
tensor(0.3415)
```

## 1.115 ChrF Score

### 1.115.1 Module Interface

`torchmetrics.CHRFScore`  
alias of `_CHRFScore`

### 1.115.2 Functional Interface

`torchmetrics.functional.chrf_score(preds, target, n_char_order=6, n_word_order=2, beta=2.0, lowercase=False, whitespace=False, return_sentence_level_score=False)`

Wrapper for deprecated import.

```
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
:rtype: :py:data:~typing.Union\[:py:class:`~torch.Tensor`, :py:data:~typing.
↳ Tuple\[:py:class:`~torch.Tensor`, :py:class:`~torch.Tensor`]
```

```
>>> _chrf_score(preds, target)
tensor(0.8640)
```

## 1.116 Extended Edit Distance

### 1.116.1 Module Interface

`torchmetrics.ExtendedEditDistance`  
alias of `_ExtendedEditDistance`

### 1.116.2 Functional Interface

`torchmetrics.functional.extended_edit_distance(preds, target, language='en', return_sentence_level_score=False, alpha=2.0, rho=0.3, deletion=0.2, insertion=1.0)`

Wrapper for deprecated import.

```
>>> preds = ["this is the prediction", "here is an other sample"]
>>> target = ["this is the reference", "here is another one"]
:rtype: :py:data:~typing.Union\[:py:class:`~torch.Tensor`, :py:data:~typing.
↳ Tuple\[:py:class:`~torch.Tensor`, :py:class:`~torch.Tensor`]
```

```
>>> _extended_edit_distance(preds=preds, target=target)
tensor(0.3078)
```

## 1.117 InfoLM

### 1.117.1 Module Interface

```
class torchmetrics.text.infolm.InfoLM(model_name_or_path='bert-base-uncased', temperature=0.25,
                                       information_measure='kl_divergence', idf=True, alpha=None,
                                       beta=None, device=None, max_length=None, batch_size=64,
                                       num_threads=0, verbose=True,
                                       return_sentence_level_score=False, **kwargs)
```

Calculate [InfoLM](#).

InfoLM measures a distance/divergence between predicted and reference sentence discrete distribution using one of the following information measures:

- [KL divergence](#)
- [alpha divergence](#)
- [beta divergence](#)
- [AB divergence](#)
- [Rényi divergence](#)
- [L1 distance](#)
- [L2 distance](#)
- [L-infinity distance](#)
- [Fisher-Rao distance](#)

[InfoLM](#) is a family of untrained embedding-based metrics which addresses some famous flaws of standard string-based metrics thanks to the usage of pre-trained masked language models. This family of metrics is mainly designed for summarization and data-to-text tasks.

The implementation of this metric is fully based HuggingFace `transformers`’ package.

As input to `forward` and `update` the metric accepts the following input:

- `preds` ([Sequence](#)): An iterable of hypothesis corpus
- `target` ([Sequence](#)): An iterable of reference corpus

As output of `forward` and `compute` the metric returns the following output:

- `infolm` ([Tensor](#)): If `return_sentence_level_score=True` return a tuple with a tensor with the corpus-level InfoLM score and a list of sentence-level InfoLM scores, else return a corpus-level InfoLM score

#### Parameters

- **model\_name\_or\_path** ([Union](#)[[str](#), [PathLike](#)]) – A name or a model path used to load `transformers` pretrained model. By default the “*bert-base-uncased*” model is used.
- **temperature** ([float](#)) – A temperature for calibrating language modelling. For more information, please reference [InfoLM](#) paper.
- **information\_measure** ([Literal](#)['kl\_divergence', 'alpha\_divergence', 'beta\_divergence', 'ab\_divergence', 'renyi\_divergence', 'l1\_distance', 'l2\_distance', 'l\_infinity\_distance', 'fisher\_rao\_distance']) – A name of information measure to be used. Please use one of: ['kl\_divergence', 'alpha\_divergence', 'beta\_divergence', 'ab\_divergence', 'renyi\_divergence', 'l1\_distance', 'l2\_distance', 'l\_infinity\_distance', 'fisher\_rao\_distance']

- **idf** (`bool`) – An indication of whether normalization using inverse document frequencies should be used.
- **alpha** (`Optional[float]`) – Alpha parameter of the divergence used for alpha, AB and Rényi divergence measures.
- **beta** (`Optional[float]`) – Beta parameter of the divergence used for beta and AB divergence measures.
- **device** (`Union[str, device, None]`) – A device to be used for calculation.
- **max\_length** (`Optional[int]`) – A maximum length of input sequences. Sequences longer than `max_length` are to be trimmed.
- **batch\_size** (`int`) – A batch size used for model processing.
- **num\_threads** (`int`) – A number of threads to use for a dataloader.
- **verbose** (`bool`) – An indication of whether a progress bar to be displayed during the embeddings calculation.
- **return\_sentence\_level\_score** (`bool`) – An indication whether a sentence-level InfoLM score to be returned.

### Example

```
>>> from torchmetrics.text.infofm import InfoLM
>>> preds = ['he read the book because he was interested in world history']
>>> target = ['he was interested in world history because he read the book']
>>> infofm = InfoLM('google/bert_uncased_L-2_H-128_A-2', idf=False)
>>> infofm(preds, target)
tensor(-0.1784)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

#### Returns

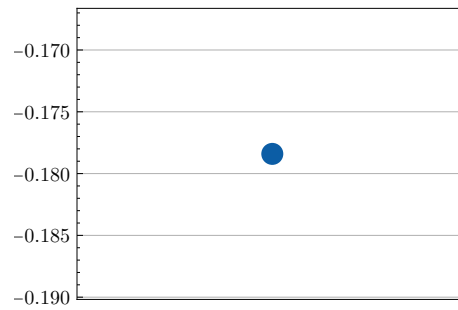
Figure and Axes object

#### Raises

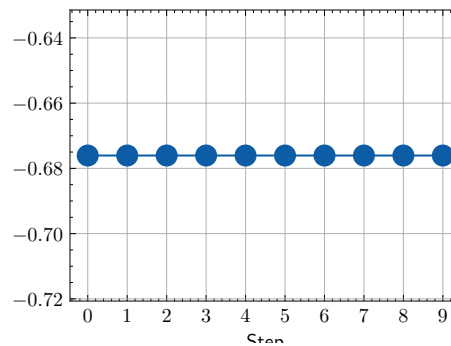
**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> from torchmetrics.text.infofm import InfoLM
>>> metric = InfoLM('google/bert_uncased_L-2_H-128_A-2', idf=False)
>>> preds = ['he read the book because he was interested in world history']
>>> target = ['he was interested in world history because he read the book']
>>> metric.update(preds, target)
>>> fig_, ax_ = metric.plot()
```





```
>>> # Example plotting multiple values
>>> from torchmetrics.text.infolm import InfoLM
>>> metric = InfoLM('google/bert_uncased_L-2_H-128_A-2', idf=False)
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(preds, target))
>>> fig_, ax_ = metric.plot(values)
```



### 1.117.2 Functional Interface

`torchmetrics.functional.text.infolm.infolm(preds, target, model_name_or_path='bert-base-uncased', temperature=0.25, information_measure='kl_divergence', idf=True, alpha=None, beta=None, device=None, max_length=None, batch_size=64, num_threads=0, verbose=True, return_sentence_level_score=False)`

Calculate InfoLM [1].

InfoML corresponds to distance/divergence between predicted and reference sentence discrete distribution using one of the following information measures:

- KL divergence
- alpha divergence
- beta divergence

- [AB divergence](#)
- [Rényi divergence](#)
- L1 distance
- L2 distance
- L-infinity distance
- [Fisher-Rao distance](#)

**InfoLM** is a family of untrained embedding-based metrics which addresses some famous flaws of standard string-based metrics thanks to the usage of pre-trained masked language models. This family of metrics is mainly designed for summarization and data-to-text tasks.

If you want to use IDF scaling over the whole dataset, please use the class metric.

The implementation of this metric is fully based HuggingFace *transformers*’ package.

#### Parameters

- **preds** (`Union[str, Sequence[str]]`) – An iterable of hypothesis corpus.
- **target** (`Union[str, Sequence[str]]`) – An iterable of reference corpus.
- **model\_name\_or\_path** (`Union[str, PathLike]`) – A name or a model path used to load *transformers* pretrained model.
- **temperature** (`float`) – A temperature for calibrating language modelling. For more information, please reference [InfoLM](#) paper.
- **information\_measure** (`Literal['kl_divergence', 'alpha_divergence', 'beta_divergence', 'ab_divergence', 'renyi_divergence', 'l1_distance', 'l2_distance', 'l_infinity_distance', 'fisher_rao_distance']`) – A name of information measure to be used. Please use one of: ['kl\_divergence', 'alpha\_divergence', 'beta\_divergence', 'ab\_divergence', 'renyi\_divergence', 'l1\_distance', 'l2\_distance', 'l\_infinity\_distance', 'fisher\_rao\_distance']
- **idf** (`bool`) – An indication of whether normalization using inverse document frequencies should be used.
- **alpha** (`Optional[float]`) – Alpha parameter of the divergence used for alpha, AB and Rényi divergence measures.
- **beta** (`Optional[float]`) – Beta parameter of the divergence used for beta and AB divergence measures.
- **device** (`Union[str, device, None]`) – A device to be used for calculation.
- **max\_length** (`Optional[int]`) – A maximum length of input sequences. Sequences longer than *max\_length* are to be trimmed.
- **batch\_size** (`int`) – A batch size used for model processing.
- **num\_threads** (`int`) – A number of threads to use for a dataloader.
- **verbose** (`bool`) – An indication of whether a progress bar to be displayed during the embeddings calculation.
- **return\_sentence\_level\_score** (`bool`) – An indication whether a sentence-level InfoLM score to be returned.

#### Return type

`Union[Tensor, Tuple[Tensor, Tensor]]`

**Returns**

A corpus-level InfoLM score. (Optionally) A list of sentence-level InfoLM scores if *return\_sentence\_level\_score=True*.

**Example**

```
>>> from torchmetrics.functional.text.infolm import infolm
>>> preds = ['he read the book because he was interested in world history']
>>> target = ['he was interested in world history because he read the book']
>>> infolm(preds, target, model_name_or_path='google/bert_uncased_L-2_H-128_A-2',
↳idf=False)
tensor(-0.1784)
```

**References**

[1] InfoLM: A New Metric to Evaluate Summarization & Data2Text Generation by Pierre Colombo, Chloé Clavel and Pablo Piantanida [InfoLM](#)

## 1.118 Match Error Rate

### 1.118.1 Module Interface

`torchmetrics.MatchErrorRate`  
alias of `_MatchErrorRate`

### 1.118.2 Functional Interface

`torchmetrics.functional.match_error_rate(preds, target)`  
Wrapper for deprecated import.

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _match_error_rate(preds=preds, target=target)
tensor(0.4444)
```

## 1.119 Perplexity

### 1.119.1 Module Interface

**class** `torchmetrics.text.perplexity.Perplexity(ignore_index=None, **kwargs)`

Perplexity measures how well a language model predicts a text sample.

It's calculated as the average number of bits per word a model needs to represent the sample.

As input to forward and update the metric accepts the following input:

- **preds** (**Tensor**): Logits or a unnormalized score assigned to each token in a sequence with shape `[batch_size, seq_len, vocab_size]`, which is the output of a language model. Scores will be normalized internally using softmax.
- **target** (**Tensor**): Ground truth values with a shape `[batch_size, seq_len]`

As output of `forward` and `compute` the metric returns the following output:

- **perp** (**Tensor**): A tensor with the perplexity score

#### Parameters

- **ignore\_index** (**Optional[int]**) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score.
- **kwargs** (**Dict[str, Any]**) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Examples

```
>>> from torchmetrics.text import Perplexity
>>> import torch
>>> preds = torch.rand(2, 8, 5, generator=torch.manual_seed(22))
>>> target = torch.randint(5, (2, 8), generator=torch.manual_seed(22))
>>> target[0, 6:] = -100
>>> perp = Perplexity(ignore_index=-100)
>>> perp(preds, target)
tensor(5.2545)
```

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

#### Parameters

- **val** (**Union[Tensor, Sequence[Tensor], None]**) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (**Optional[Axes]**) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

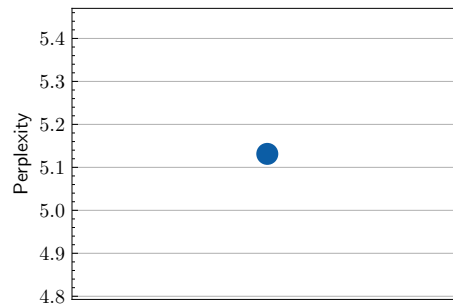
#### Returns

Figure and Axes object

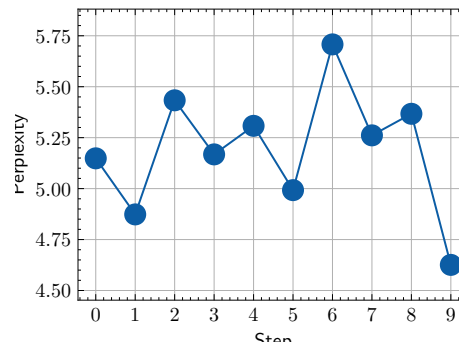
#### Raises

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.text import Perplexity
>>> metric = Perplexity()
>>> metric.update(torch.rand(2, 8, 5), torch.randint(5, (2, 8)))
>>> fig, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.text import Perplexity
>>> metric = Perplexity()
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(torch.rand(2, 8, 5), torch.randint(5, (2, 8))))
>>> fig_, ax_ = metric.plot(values)
```



## 1.119.2 Functional Interface

`torchmetrics.functional.text.perplexity.perplexity(preds, target, ignore_index=None)`

Perplexity measures how well a language model predicts a text sample.

This metric is calculated as the average number of bits per word a model needs to represent the sample.

### Parameters

- **preds** (`Tensor`) – Logits or a unnormalized score assigned to each token in a sequence with shape `[batch_size, seq_len, vocab_size]`, which is the output of a language model. Scores will be normalized internally using softmax.
- **target** (`Tensor`) – Ground truth values with a shape `[batch_size, seq_len]`.
- **ignore\_index** (`Optional[int]`) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score.

**Return type**

Tensor

**Returns**

Perplexity value

**Examples**

```
>>> import torch
>>> preds = torch.rand(2, 8, 5, generator=torch.manual_seed(22))
>>> target = torch.randint(5, (2, 8), generator=torch.manual_seed(22))
>>> target[0, 6:] = -100
>>> perplexity(preds, target, ignore_index=-100)
tensor(5.2545)
```

## 1.120 ROUGE Score

### 1.120.1 Module Interface

```
class torchmetrics.text.rouge.ROUGEScore(use_stemmer=False, normalizer=None, tokenizer=None,
                                          accumulate='best', rouge_keys=('rouge1', 'rouge2', 'rougeL',
                                          'rougeLsum'), **kwargs)
```

Calculate Rouge Score, used for automatic summarization.

This implementation should imitate the behaviour of the `rouge-score` package *Python ROUGE Implementation*

As input to `forward` and `update` the metric accepts the following input:

- **preds** (Sequence): An iterable of predicted sentences or a single predicted sentence
- **target** (Sequence): An iterable of target sentences or an iterable of interables of target sentences or a single target sentence

As output of `forward` and `compute` the metric returns the following output:

- **rouge** (Dict): A dictionary of tensor rouge scores for each input str rouge key

**Parameters**

- **use\_stemmer** (bool) – Use Porter stemmer to strip word suffixes to improve matching.
- **normalizer** (Optional[Callable[[str], str]]) – A user's own normalizer function. If this is None, replacing any non-alpha-numeric characters with spaces is default. This function must take a `str` and return a `str`.
- **tokenizer** (Optional[Callable[[str], Sequence[str]]]) – A user's own tokenizer function. If this is None, splitting by spaces is default This function must take a `str` and return `Sequence[str]`
- **accumulate** (Literal['avg', 'best']) – Useful in case of multi-reference rouge score.
  - `avg` takes the avg of all references with respect to predictions
  - `best` takes the best fmeasure score obtained between prediction and multiple corresponding references.

- **rouge\_keys** (`Union[str, Tuple[str, ...]]`) – A list of rouge types to calculate. Keys that are allowed are rougeL, rougeLsum, and rouge1 through rouge9.
- **kwargs** (`Any`) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

### Example

```
>>> from torchmetrics.text.rouge import ROUGEScore
>>> preds = "My name is John"
>>> target = "Is your name John"
>>> rouge = ROUGEScore()
>>> from pprint import pprint
>>> pprint(rouge(preds, target))
{'rouge1_fmeasure': tensor(0.7500),
 'rouge1_precision': tensor(0.7500),
 'rouge1_recall': tensor(0.7500),
 'rouge2_fmeasure': tensor(0.),
 'rouge2_precision': tensor(0.),
 'rouge2_recall': tensor(0.),
 'rougeL_fmeasure': tensor(0.5000),
 'rougeL_precision': tensor(0.5000),
 'rougeL_recall': tensor(0.5000),
 'rougeLsum_fmeasure': tensor(0.5000),
 'rougeLsum_precision': tensor(0.5000),
 'rougeLsum_recall': tensor(0.5000)}
```

### Raises

- **ValueError** – If the python packages `nltk` is not installed.
- **ValueError** – If any of the rouge\_keys does not belong to the allowed set of keys.

**plot**(*val=None, ax=None*)

Plot a single or multiple values from the metric.

### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

### Returns

Figure and Axes object

### Raises

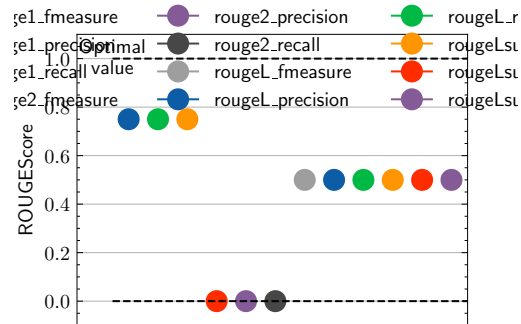
**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> from torchmetrics.text.rouge import ROUGEScore
>>> metric = ROUGEScore()
>>> preds = "My name is John"
>>> target = "Is your name John"
```

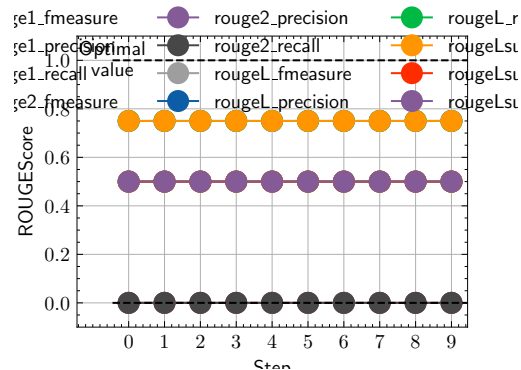
(continues on next page)

(continued from previous page)

```
>>> metric.update(preds, target)
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> from torchmetrics.text.rouge import ROUGEScore
>>> metric = ROUGEScore()
>>> preds = "My name is John"
>>> target = "Is your name John"
>>> values = [ ]
>>> for _ in range(10):
...     values.append(metric(preds, target))
>>> fig_, ax_ = metric.plot(values)
```



## 1.120.2 Functional Interface

`torchmetrics.functional.text.rouge.rouge_score(preds, target, accumulate='best', use_stemmer=False, normalizer=None, tokenizer=None, rouge_keys=('rouge1', 'rouge2', 'rougeL', 'rougeLsum'))`

Calculate [Calculate Rouge Score](#), used for automatic summarization.

### Parameters

- **preds** (`Union[str, Sequence[str]]`) – An iterable of predicted sentences or a single predicted sentence.



- **target** (`Union[str, Sequence[str], Sequence[Sequence[str]]]`) – An iterable of iterables of target sentences or an iterable of target sentences or a single target sentence.
- **accumulate** (`Literal['avg', 'best']`) – Useful incase of multi-reference rouge score.
  - `avg` takes the avg of all references with respect to predictions
  - `best` takes the best fmeasure score obtained between prediction and multiple corresponding references.
- **use\_stemmer** (`bool`) – Use Porter stemmer to strip word suffixes to improve matching.
- **normalizer** (`Optional[Callable[[str], str]]`) – A user’s own normalizer function. If this is `None`, replacing any non-alpha-numeric characters with spaces is default. This function must take a `str` and return a `str`.
- **tokenizer** (`Optional[Callable[[str], Sequence[str]]]`) – A user’s own tokenizer function. If this is `None`, splitting by spaces is default This function must take a `str` and return `Sequence[str]`
- **rouge\_keys** (`Union[str, Tuple[str, ...]]`) – A list of rouge types to calculate. Keys that are allowed are `rougeL`, `rougeLsum`, and `rouge1` through `rouge9`.

**Return type**`Dict[str, Tensor]`**Returns**

Python dictionary of rouge scores for each input rouge key.

**Example**

```
>>> from torchmetrics.functional.text.rouge import rouge_score
>>> preds = "My name is John"
>>> target = "Is your name John"
>>> from pprint import pprint
>>> pprint(rouge_score(preds, target))
{'rouge1_fmeasure': tensor(0.7500),
 'rouge1_precision': tensor(0.7500),
 'rouge1_recall': tensor(0.7500),
 'rouge2_fmeasure': tensor(0.),
 'rouge2_precision': tensor(0.),
 'rouge2_recall': tensor(0.),
 'rougeL_fmeasure': tensor(0.5000),
 'rougeL_precision': tensor(0.5000),
 'rougeL_recall': tensor(0.5000),
 'rougeLsum_fmeasure': tensor(0.5000),
 'rougeLsum_precision': tensor(0.5000),
 'rougeLsum_recall': tensor(0.5000)}
```

**Raises**

- **ModuleNotFoundError** – If the python package `nltk` is not installed.
- **ValueError** – If any of the `rouge_keys` does not belong to the allowed set of keys.

## References

[1] ROUGE: A Package for Automatic Evaluation of Summaries by Chin-Yew Lin. <https://aclanthology.org/W04-1013/>

## 1.121 Sacre BLEU Score

### 1.121.1 Module Interface

`torchmetrics.SacreBLEUScore`  
alias of `_SacreBLEUScore`

### 1.121.2 Functional Interface

`torchmetrics.functional.sacre_bleu_score(preds, target, n_gram=4, smooth=False, tokenize='l3a', lowercase=False, weights=None)`

Wrapper for deprecated import.

```
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _sacre_bleu_score(preds, target)
tensor(0.7598)
```

## 1.122 SQuAD

### 1.122.1 Module Interface

`torchmetrics.SQuAD`  
alias of `_SQuAD`

### 1.122.2 Functional Interface

`torchmetrics.functional.squad(preds, target)`

Wrapper for deprecated import.

```
>>> preds = [{"prediction_text": "1976", "id": "56e10a3be3433e1400422b22"}]
>>> target = [{"answers": {"answer_start": [97], "text": ["1976"]}, "id":
↪ "56e10a3be3433e1400422b22"}]
:rtype: :py:class:`~typing.Dict`[:py:class:`str`, :py:class:`~torch.Tensor`]
```

```
>>> _squad(preds, target)
{'exact_match': tensor(100.), 'f1': tensor(100.)}
```

## 1.123 Translation Edit Rate (TER)

### 1.123.1 Module Interface

`torchmetrics.TranslationEditRate`  
alias of `_TranslationEditRate`

### 1.123.2 Functional Interface

`torchmetrics.functional.translation_edit_rate(preds, target, normalize=False, no_punctuation=False, lowercase=True, asian_support=False, return_sentence_level_score=False)`

Wrapper for deprecated import.

```
>>> preds = ['the cat is on the mat']
>>> target = [['there is a cat on the mat', 'a cat is on the mat']]
:rtype: :py:data:`~typing.Union`[:py:class:`~torch.Tensor`, :py:data:`~typing.
↳ Tuple`[:py:class:`~torch.Tensor`, :py:class:`~typing.List`[:py:class:`~torch.
↳ Tensor`]]]
```

```
>>> _translation_edit_rate(preds, target)
tensor(0.1538)
```

## 1.124 Word Error Rate

### 1.124.1 Module Interface

`torchmetrics.WordErrorRate`  
alias of `_WordErrorRate`

### 1.124.2 Functional Interface

`torchmetrics.functional.word_error_rate(preds, target)`

Wrapper for deprecated import.

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _word_error_rate(preds=preds, target=target)
tensor(0.5000)
```

## 1.125 Word Info. Lost

### 1.125.1 Module Interface

`torchmetrics.WordInfoLost`

alias of `_WordInfoLost`

### 1.125.2 Functional Interface

`torchmetrics.functional.word_information_lost(preds, target)`

Wrapper for deprecated import.

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _word_information_lost(preds, target)
tensor(0.6528)
```

## 1.126 Word Info. Preserved

### 1.126.1 Module Interface

`torchmetrics.WordInfoPreserved`

alias of `_WordInfoPreserved`

### 1.126.2 Functional Interface

`torchmetrics.functional.word_information_preserved(preds, target)`

Wrapper for deprecated import.

```
>>> preds = ["this is the prediction", "there is an other sample"]
>>> target = ["this is the reference", "there is another one"]
:rtype: :py:class:`~torch.Tensor`
```

```
>>> _word_information_preserved(preds, target)
tensor(0.3472)
```

## 1.127 Bootstrapper

### 1.127.1 Module Interface

**class** torchmetrics.**BootStrapper**(*base\_metric*, *num\_bootstraps*=10, *mean*=True, *std*=True, *quantile*=None, *raw*=False, *sampling\_strategy*='poisson', *\*\*kwargs*)

Using [Turn a Metric into a Bootstrapped](#).

That can automate the process of getting confidence intervals for metric values. This wrapper class basically keeps multiple copies of the same base metric in memory and whenever `update` or `forward` is called, all input tensors are resampled (with replacement) along the first dimension.

#### Parameters

- **base\_metric** (*Metric*) – base metric class to wrap
- **num\_bootstraps** (*int*) – number of copies to make of the base metric for bootstrapping
- **mean** (*bool*) – if True return the mean of the bootstraps
- **std** (*bool*) – if True return the standard deviation of the bootstraps
- **quantile** (*Union[float, Tensor, None]*) – if given, returns the quantile of the bootstraps. Can only be used with pytorch version 1.6 or higher
- **raw** (*bool*) – if True, return all bootstrapped values
- **sampling\_strategy** (*str*) – Determines how to produce bootstrapped samplings. Either 'poisson' or multinomial. If 'poisson' is chosen, the number of times each sample will be included in the bootstrap will be given by  $n \sim \text{Poisson}(\lambda = 1)$ , which approximates the true bootstrap distribution when the number of samples is large. If 'multinomial' is chosen, we will apply true bootstrapping at the batch level to approximate bootstrapping over the whole dataset.
- **kwargs** (*Any*) – Additional keyword arguments, see [Advanced metric settings](#) for more info.

#### Example::

```
>>> from pprint import pprint
>>> from torchmetrics.wrappers import BootStrapper
>>> from torchmetrics.classification import MulticlassAccuracy
>>> _ = torch.manual_seed(123)
>>> base_metric = MulticlassAccuracy(num_classes=5, average='micro')
>>> bootstrap = BootStrapper(base_metric, num_bootstraps=20)
>>> bootstrap.update(torch.randint(5, (20,)), torch.randint(5, (20,)))
>>> output = bootstrap.compute()
>>> pprint(output)
{'mean': tensor(0.2205), 'std': tensor(0.0859)}
```

#### compute()

Compute the bootstrapped metric values.

Always returns a dict of tensors, which can contain the following keys: `mean`, `std`, `quantile` and `raw` depending on how the class was initialized.

#### Return type

`Dict[str, Tensor]`

`plot(val=None, ax=None)`

Plot a single or multiple values from the metric.

#### Parameters

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling `metric.forward` or `metric.compute` or a list of these results. If no value is provided, will automatically call `metric.compute` and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### Return type

`Tuple[Figure, Union[Axes, ndarray]]`

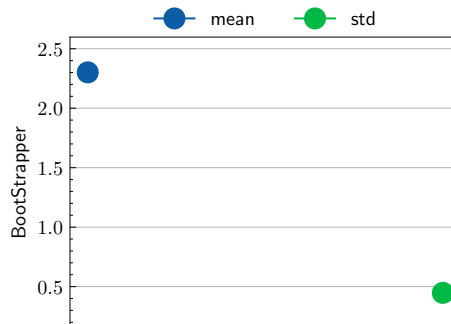
#### Returns

Figure and Axes object

#### Raises

**ModuleNotFoundError** – If `matplotlib` is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.wrappers import BootStrapper
>>> from torchmetrics.regression import MeanSquaredError
>>> metric = BootStrapper(MeanSquaredError(), num_bootstraps=20)
>>> metric.update(torch.randn(100,), torch.randn(100,))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.wrappers import BootStrapper
>>> from torchmetrics.regression import MeanSquaredError
>>> metric = BootStrapper(MeanSquaredError(), num_bootstraps=20)
>>> values = [ ]
>>> for _ in range(3):
...     values.append(metric(torch.randn(100,), torch.randn(100,)))
>>> fig_, ax_ = metric.plot(values)
```

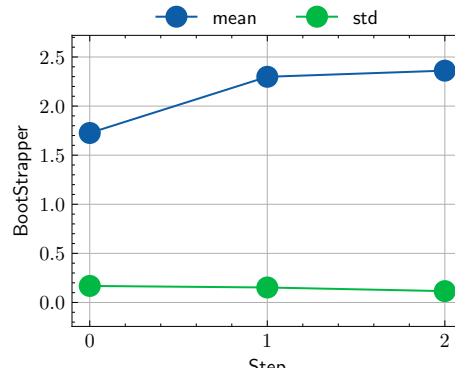
`update(*args, **kwargs)`

Update the state of the base metric.

Any tensor passed in will be bootstrapped along dimension 0.

#### Return type

`None`



## 1.128 Classwise Wrapper

### 1.128.1 Module Interface

**class** `torchmetrics.ClasswiseWrapper`(*metric*, *labels=None*, *prefix=None*, *postfix=None*)

Wrapper metric for altering the output of classification metrics.

This metric works together with classification metrics that returns multiple values (one value per class) such that label information can be automatically included in the output.

#### Parameters

- **metric** (*Metric*) – base metric that should be wrapped. It is assumed that the metric outputs a single tensor that is split along the first dimension.
- **labels** (*Optional[List[str]]*) – list of strings indicating the different classes.
- **prefix** (*Optional[str]*) – string that is prepended to the metric names.
- **postfix** (*Optional[str]*) – string that is appended to the metric names.

#### Example::

Basic example where the output of a metric is unwrapped into a dictionary with the class index as keys:

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> from torchmetrics.wrappers import ClasswiseWrapper
>>> from torchmetrics.classification import MulticlassAccuracy
>>> metric = ClasswiseWrapper(MulticlassAccuracy(num_classes=3, average=None))
>>> preds = torch.randn(10, 3).softmax(dim=-1)
>>> target = torch.randint(3, (10,))
>>> metric(preds, target)
{'multiclassaccuracy_0': tensor(0.5000),
 'multiclassaccuracy_1': tensor(0.7500),
 'multiclassaccuracy_2': tensor(0.)}
```

#### Example::

Using custom name via prefix and postfix:

```
>>> import torch
>>> _ = torch.manual_seed(42)
>>> from torchmetrics.wrappers import ClasswiseWrapper
```

(continues on next page)

(continued from previous page)

```
>>> from torchmetrics.classification import MulticlassAccuracy
>>> metric_pre = ClasswiseWrapper(MulticlassAccuracy(num_classes=3,
↳ average=None), prefix="acc-")
>>> metric_post = ClasswiseWrapper(MulticlassAccuracy(num_classes=3,
↳ average=None), postfix="-acc")
>>> preds = torch.randn(10, 3).softmax(dim=-1)
>>> target = torch.randint(3, (10,))
>>> metric_pre(preds, target)
{'acc-0': tensor(0.5000),
 'acc-1': tensor(0.7500),
 'acc-2': tensor(0.)}
>>> metric_post(preds, target)
{'0-acc': tensor(0.5000),
 '1-acc': tensor(0.7500),
 '2-acc': tensor(0.)}
```

**Example::**

Providing labels as a list of strings:

```
>>> from torchmetrics.wrappers import ClasswiseWrapper
>>> from torchmetrics.classification import MulticlassAccuracy
>>> metric = ClasswiseWrapper(
...     MulticlassAccuracy(num_classes=3, average=None),
...     labels=["horse", "fish", "dog"]
... )
>>> preds = torch.randn(10, 3).softmax(dim=-1)
>>> target = torch.randint(3, (10,))
>>> metric(preds, target)
{'multiclassaccuracy_horse': tensor(0.3333),
 'multiclassaccuracy_fish': tensor(0.6667),
 'multiclassaccuracy_dog': tensor(0.)}
```

**Example::**

Classwise can also be used in combination with MetricCollection. In this case, everything will be flattened into a single dictionary:

```
>>> from torchmetrics import MetricCollection
>>> from torchmetrics.wrappers import ClasswiseWrapper
>>> from torchmetrics.classification import MulticlassAccuracy, MulticlassRecall
>>> labels = ["horse", "fish", "dog"]
>>> metric = MetricCollection(
...     {'multiclassaccuracy': ClasswiseWrapper(MulticlassAccuracy(num_
↳ classes=3, average=None), labels),
...     'multiclassrecall': ClasswiseWrapper(MulticlassRecall(num_classes=3,
↳ average=None), labels)}
... )
>>> preds = torch.randn(10, 3).softmax(dim=-1)
>>> target = torch.randint(3, (10,))
>>> metric(preds, target)
{'multiclassaccuracy_horse': tensor(0.),
 'multiclassaccuracy_fish': tensor(0.3333),
 'multiclassaccuracy_dog': tensor(0.4000),
```

(continues on next page)



(continued from previous page)

```
'multiclassrecall_horse': tensor(0.),
'multiclassrecall_fish': tensor(0.3333),
'multiclassrecall_dog': tensor(0.4000)}
```

**compute()**

Compute metric.

**Return type**

`Dict[str, Tensor]`

**forward(\*args, \*\*kwargs)**

Calculate on batch and accumulate to global state.

**Return type**

`Any`

**plot(val=None, ax=None)**

Plot a single or multiple values from the metric.

**Parameters**

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

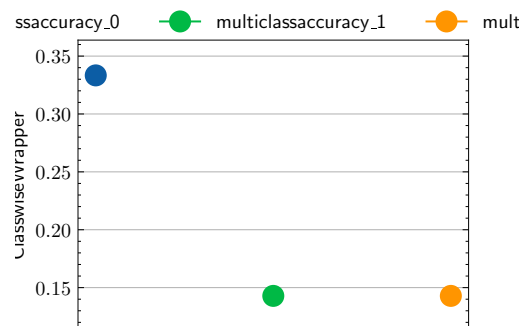
**Returns**

Figure and Axes object

**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

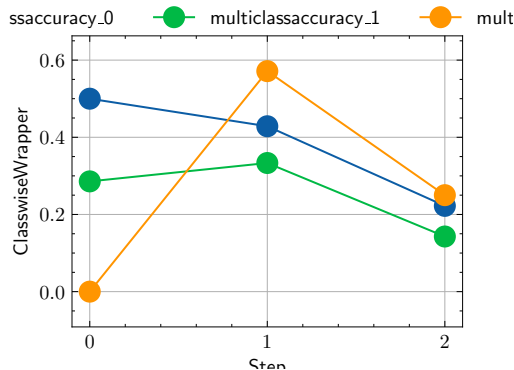
```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.wrappers import ClasswiseWrapper
>>> from torchmetrics.classification import MulticlassAccuracy
>>> metric = ClasswiseWrapper(MulticlassAccuracy(num_classes=3, average=None))
>>> metric.update(torch.randint(3, (20,)), torch.randint(3, (20,)))
>>> fig, ax_ = metric.plot()
```



```

>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.wrappers import ClasswiseWrapper
>>> from torchmetrics.classification import MulticlassAccuracy
>>> metric = ClasswiseWrapper(MulticlassAccuracy(num_classes=3, average=None))
>>> values = [ ]
>>> for _ in range(3):
...     values.append(metric(torch.randint(3, (20,)), torch.randint(3, (20,))))
>>> fig_, ax_ = metric.plot(values)

```



**reset()**

Reset metric.

**Return type**

`None`

**update(\*args, \*\*kwargs)**

Update state.

**Return type**

`None`

## 1.129 Metric Tracker

### 1.129.1 Module Interface

**class** torchmetrics.**MetricTracker**(metric, maximize=True)

A wrapper class that can help keeping track of a metric or metric collection over time.

The wrapper implements the standard `.update()`, `.compute()`, `.reset()` methods that just calls corresponding method of the currently tracked metric. However, the following additional methods are provided:

- MetricTracker.n\_steps: number of metrics being tracked
- MetricTracker.increment(): initialize a new metric for being tracked
- MetricTracker.compute\_all(): get the metric value for all steps
- MetricTracker.best\_metric(): returns the best value

Out of the box, this wrapper class fully supports that the base metric being tracked is a single *Metric*, a *MetricCollection* or another *MetricWrapper* wrapped around a metric. However, multiple layers of nesting, such as using a *Metric* inside a *MetricWrapper* inside a *MetricCollection* is not fully supported, especially the `.best_metric` method that cannot auto compute the best metric and index for such nested structures.

**Parameters**

- **metric** (`Union[Metric, MetricCollection]`) – instance of a `torchmetrics.Metric` or `torchmetrics.MetricCollection` to keep track of at each timestep.
- **maximize** (`Union[bool, List[bool]]`) – either single bool or list of bool indicating if higher metric values are better (`True`) or lower is better (`False`).

**Example (single metric):**

```
>>> from torchmetrics.wrappers import MetricTracker
>>> from torchmetrics.classification import MulticlassAccuracy
>>> _ = torch.manual_seed(42)
>>> tracker = MetricTracker(MulticlassAccuracy(num_classes=10, average='micro'))
>>> for epoch in range(5):
...     tracker.increment()
...     for batch_idx in range(5):
...         preds, target = torch.randint(10, (100,)), torch.randint(10, (100,))
...         tracker.update(preds, target)
...         print(f"current acc={tracker.compute()}")
current acc=0.11200000034570694
current acc=0.08799999952316284
current acc=0.126000000202655792
current acc=0.07999999821186066
current acc=0.10199999809265137
>>> best_acc, which_epoch = tracker.best_metric(return_step=True)
>>> best_acc
0.1260...
>>> which_epoch
2
>>> tracker.compute_all()
tensor([0.1120, 0.0880, 0.1260, 0.0800, 0.1020])
```

**Example (multiple metrics using MetricCollection):**

```
>>> from torchmetrics.wrappers import MetricTracker
>>> from torchmetrics import MetricCollection
>>> from torchmetrics.regression import MeanSquaredError, ExplainedVariance
>>> _ = torch.manual_seed(42)
>>> tracker = MetricTracker(MetricCollection([MeanSquaredError(),
↳ ExplainedVariance()])), maximize=[False, True])
>>> for epoch in range(5):
...     tracker.increment()
...     for batch_idx in range(5):
...         preds, target = torch.randn(100), torch.randn(100)
...         tracker.update(preds, target)
...         print(f"current stats={tracker.compute()}")
current stats={'MeanSquaredError': tensor(1.8218), 'ExplainedVariance': tensor(-
↳ 0.8969)}
current stats={'MeanSquaredError': tensor(2.0268), 'ExplainedVariance': tensor(-
↳ 1.0206)}
current stats={'MeanSquaredError': tensor(1.9491), 'ExplainedVariance': tensor(-
↳ 0.8298)}
current stats={'MeanSquaredError': tensor(1.9800), 'ExplainedVariance': tensor(-
↳ 0.9199)}
```

(continues on next page)

(continued from previous page)

```

current stats={'MeanSquaredError': tensor(2.2481), 'ExplainedVariance': tensor(-
  ↪1.1622)}
>>> from pprint import pprint
>>> best_res, which_epoch = tracker.best_metric(return_step=True)
>>> pprint(best_res)
{'ExplainedVariance': -0.829...,
 'MeanSquaredError': 1.821...}
>>> which_epoch
{'MeanSquaredError': 0, 'ExplainedVariance': 2}
>>> pprint(tracker.compute_all())
{'ExplainedVariance': tensor([-0.8969, -1.0206, -0.8298, -0.9199, -1.1622]),
 'MeanSquaredError': tensor([1.8218, 2.0268, 1.9491, 1.9800, 2.2481])}

```

**best\_metric**(*return\_step=False*)

Return the highest metric out of all tracked.

#### Parameters

**return\_step** (*bool*) – If True will also return the step with the highest metric value.

#### Return type

`Union[None, float, Tuple[float, int], Tuple[None, None], Dict[str, Optional[float]], Tuple[Dict[str, Optional[float]], Dict[str, Optional[int]]]`

#### Returns

Either a single value or a tuple, depends on the value of `return_step` and the object being tracked.

- If a single metric is being tracked and `return_step=False` then a single tensor will be returned
- If a single metric is being tracked and `return_step=True` then a 2-element tuple will be returned, where the first value is optimal value and second value is the corresponding optimal step
- If a metric collection is being tracked and `return_step=False` then a single dict will be returned, where keys correspond to the different values of the collection and the values are the optimal metric value
- If a metric collection is being tracked and `return_step=True` then a 2-element tuple will be returned where each is a dict, with keys corresponding to the different values of the collection and the values of the first dict being the optimal values and the values of the second dict being the optimal step

In addition the value in all cases may be `None` if the underlying metric does not have a proper defined way of being optimal or in the case where a nested structure of metrics are being tracked.

**compute\_all**()

Compute the metric value for all tracked metrics.

#### Return type

`Any`

#### Returns

By default will try stacking the results from all increments into a single tensor if the tracked base object is a single metric. If a metric collection is provided a dict of stacked tensors will be returned. If the stacking process fails a list of the computed results will be returned.

**Raises**

**ValueError** – If *self.increment* have not been called before this method is called.

**forward**(\*args, \*\*kwargs)

Call forward of the current metric being tracked.

**Return type**

*None*

**increment**()

Create a new instance of the input metric that will be updated next.

**Return type**

*None*

**plot**(val=*None*, ax=*None*)

Plot a single or multiple values from the metric.

**Parameters**

- **val** (*Union*[*Tensor*, *Sequence*[*Tensor*], *None*]) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (*Optional*[*Axes*]) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

*Tuple*[*Figure*, *Union*[*Axes*, *ndarray*]]

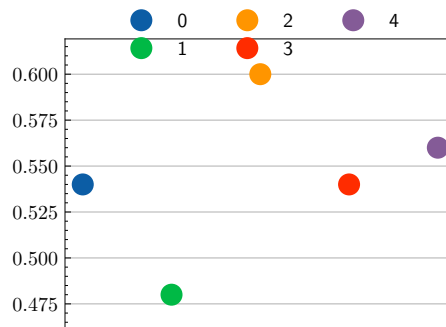
**Returns**

Figure and Axes object

**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.wrappers import MetricTracker
>>> from torchmetrics.classification import BinaryAccuracy
>>> tracker = MetricTracker(BinaryAccuracy())
>>> for epoch in range(5):
...     tracker.increment()
...     for batch_idx in range(5):
...         tracker.update(torch.randint(2, (10,)), torch.randint(2, (10,)))
>>> fig_, ax_ = tracker.plot() # plot all epochs
```



**reset()**

Reset the current metric being tracked.

**Return type**`None`**reset\_all()**

Reset all metrics being tracked.

**Return type**`None`**property n\_steps: int**

Returns the number of times the tracker has been incremented.

## 1.130 Min / Max

### 1.130.1 Module Interface

**class** `torchmetrics.MinMaxMetric`(*base\_metric*, *\*\*kwargs*)

Wrapper metric that tracks both the minimum and maximum of a scalar/tensor across an experiment.

The min/max value will be updated each time `.compute` is called.**Parameters**

- **base\_metric** (*Metric*) – The metric of which you want to keep track of its maximum and minimum values.
- **kwargs** (*Any*) – Additional keyword arguments, see *Advanced metric settings* for more info.

**Raises**

**ValueError** – If `base_metric` argument is not a subclasses instance of `torchmetrics.Metric`

**Example::**

```
>>> import torch
>>> from torchmetrics.wrappers import MinMaxMetric
>>> from torchmetrics.classification import BinaryAccuracy
>>> from pprint import pprint
>>> base_metric = BinaryAccuracy()
>>> minmax_metric = MinMaxMetric(base_metric)
>>> preds_1 = torch.Tensor([[0.1, 0.9], [0.2, 0.8]])
>>> preds_2 = torch.Tensor([[0.9, 0.1], [0.2, 0.8]])
>>> labels = torch.Tensor([[0, 1], [0, 1]]).long()
>>> pprint(minmax_metric(preds_1, labels))
{'max': tensor(1.), 'min': tensor(1.), 'raw': tensor(1.)}
>>> pprint(minmax_metric.compute())
{'max': tensor(1.), 'min': tensor(1.), 'raw': tensor(1.)}
>>> minmax_metric.update(preds_2, labels)
>>> pprint(minmax_metric.compute())
{'max': tensor(1.), 'min': tensor(0.7500), 'raw': tensor(0.7500)}
```

**compute()**

Compute the underlying metric as well as max and min values for this metric.

Returns a dictionary that consists of the computed value (`raw`), as well as the minimum (`min`) and maximum (`max`) values.

**Return type**

`Dict[str, Tensor]`

**plot(val=None, ax=None)**

Plot a single or multiple values from the metric.

**Parameters**

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

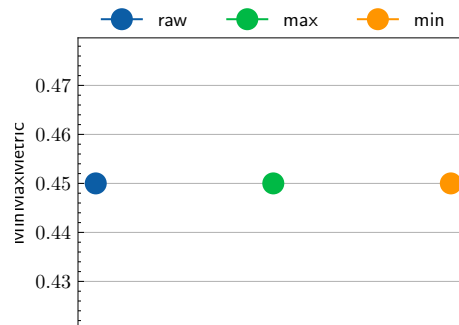
**Returns**

Figure and Axes object

**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.wrappers import MinMaxMetric
>>> from torchmetrics.classification import BinaryAccuracy
>>> metric = MinMaxMetric(BinaryAccuracy())
>>> metric.update(torch.randint(2, (20,)), torch.randint(2, (20,)))
>>> fig_, ax_ = metric.plot()
```

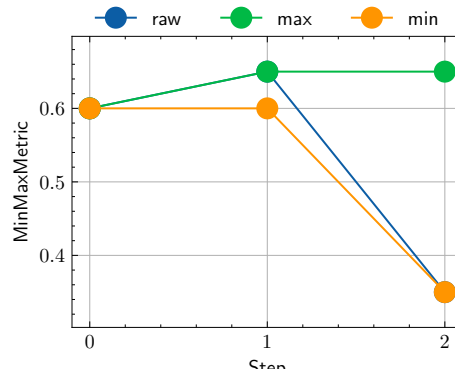


```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.wrappers import MinMaxMetric
>>> from torchmetrics.classification import BinaryAccuracy
>>> metric = MinMaxMetric(BinaryAccuracy())
>>> values = [ ]
>>> for _ in range(3):
```

(continues on next page)

(continued from previous page)

```
... values.append(metric(torch.randint(2, (20,)), torch.randint(2, (20,))))
>>> fig_, ax_ = metric.plot(values)
```

**reset()**

Set max\_val and min\_val to the initialization bounds and resets the base metric.

**Return type**

`None`

**update(\*args, \*\*kwargs)**

Update the underlying metric.

**Return type**

`None`

## 1.131 Multi-output Wrapper

### 1.131.1 Module Interface

```
class torchmetrics.MultioutputWrapper(base_metric, num_outputs, output_dim=-1, remove_nans=True,
                                     squeeze_outputs=True)
```

Wrap a base metric to enable it to support multiple outputs.

Several torchmetrics metrics, such as `torchmetrics.regression.spearman.SpearmanCorrcoef` lack support for multioutput mode. This class wraps such metrics to support computing one metric per output. Unlike specific torchmetric metrics, it doesn't support any aggregation across outputs. This means if you set `num_outputs` to 2, `.compute()` will return a Tensor of dimension `(2, ...)` where `...` represents the dimensions the metric returns when not wrapped.

In addition to enabling multioutput support for metrics that lack it, this class also supports, albeit in a crude fashion, dealing with missing labels (or other data). When `remove_nans` is passed, the class will remove the intersection of NaN containing "rows" upon each update for each output. For example, suppose a user uses `MultioutputWrapper` to wrap `torchmetrics.regression.r2.R2Score` with 2 outputs, one of which occasionally has missing labels for classes like `R2Score` is that this class supports removing NaN values (parameter `remove_nans`) on a per-output basis. When `remove_nans` is passed the wrapper will remove all rows

**Parameters**

- **base\_metric** (*Metric*) – Metric being wrapped.



- **num\_outputs** (`int`) – Expected dimensionality of the output dimension. This parameter is used to determine the number of distinct metrics we need to track.
- **output\_dim** (`int`) – Dimension on which output is expected. Note that while this provides some flexibility, the output dimension must be the same for all inputs to update. This applies even for metrics such as *Accuracy* where the labels can have a different number of dimensions than the predictions. This can be worked around if the output dimension can be set to -1 for both, even if -1 corresponds to different dimensions in different inputs.
- **remove\_nans** (`bool`) – Whether to remove the intersection of rows containing NaNs from the values passed through to each underlying metric. Proper operation requires all tensors passed to update to have dimension (N, ...) where N represents the length of the batch or dataset being passed in.
- **squeeze\_outputs** (`bool`) – If True, will squeeze the 1-item dimensions left after `index_select` is applied. This is sometimes unnecessary but harmless for metrics such as *R2Score* but useful for certain classification metrics that can't handle additional 1-item dimensions.

### Example

```
>>> # Mimic R2Score in `multioutput`, `raw_values` mode:
>>> import torch
>>> from torchmetrics.wrappers import MultioutputWrapper
>>> from torchmetrics.regression import R2Score
>>> target = torch.tensor([[0.5, 1], [-1, 1], [7, -6]])
>>> preds = torch.tensor([[0, 2], [-1, 2], [8, -5]])
>>> r2score = MultioutputWrapper(R2Score(), 2)
>>> r2score(preds, target)
tensor([0.9654, 0.9082])
```

#### **compute()**

Compute metrics.

#### **Return type**

`Tensor`

#### **forward(\*args, \*\*kwargs)**

Call underlying forward methods and aggregate the results if they're non-null.

We override this method to ensure that state variables get copied over on the underlying metrics.

#### **Return type**

`Any`

#### **plot(val=None, ax=None)**

Plot a single or multiple values from the metric.

#### **Parameters**

- **val** (`Union[Tensor, Sequence[Tensor], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

#### **Return type**

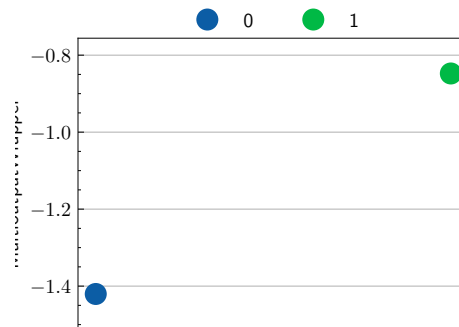
`Tuple[Figure, Union[Axes, ndarray]]`

**Returns**

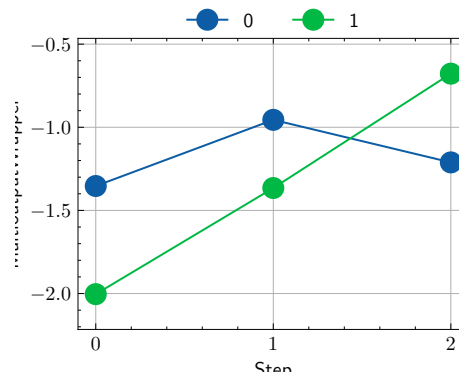
Figure and Axes object

**Raises****ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.wrappers import MultioutputWrapper
>>> from torchmetrics.regression import R2Score
>>> metric = MultioutputWrapper(R2Score(), 2)
>>> metric.update(torch.randn(20, 2), torch.randn(20, 2))
>>> fig_, ax_ = metric.plot()
```



```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.wrappers import MultioutputWrapper
>>> from torchmetrics.regression import R2Score
>>> metric = MultioutputWrapper(R2Score(), 2)
>>> values = [ ]
>>> for _ in range(3):
...     values.append(metric(torch.randn(20, 2), torch.randn(20, 2)))
>>> fig_, ax_ = metric.plot(values)
```

**reset()**

Reset all underlying metrics.

**Return type**

None

**update**(\*args, \*\*kwargs)

Update each underlying metric with the corresponding output.

**Return type**

None

## 1.132 Multi-task Wrapper

### 1.132.1 Module Interface

**class** torchmetrics.MultitaskWrapper(task\_metrics)

Wrapper class for computing different metrics on different tasks in the context of multitask learning.

In multitask learning the different tasks requires different metrics to be evaluated. This wrapper allows for easy evaluation in such cases by supporting multiple predictions and targets through a dictionary. Note that only metrics where the signature of *update* follows the standard *preds*, *target* is supported.

#### Parameters

**task\_metrics** (Dict[str, Union[Metric, MetricCollection]]) – Dictionary associating each task to a Metric or a MetricCollection. The keys of the dictionary represent the names of the tasks, and the values represent the metrics to use for each task.

#### Raises

- **TypeError** – If argument *task\_metrics* is not an dictionary
- **TypeError** – If not all values in the *task\_metrics* dictionary is instances of *Metric* or *MetricCollection*

**Example (with a single metric per class):**

```
>>> import torch
>>> from torchmetrics.wrappers import MultitaskWrapper
>>> from torchmetrics.regression import MeanSquaredError
>>> from torchmetrics.classification import BinaryAccuracy
>>>
>>> classification_target = torch.tensor([0, 1, 0])
>>> regression_target = torch.tensor([2.5, 5.0, 4.0])
>>> targets = {"Classification": classification_target, "Regression":
↳ regression_target}
>>>
>>> classification_preds = torch.tensor([0, 0, 1])
>>> regression_preds = torch.tensor([3.0, 5.0, 2.5])
>>> preds = {"Classification": classification_preds, "Regression": regression_
↳ preds}
>>>
>>> metrics = MultitaskWrapper({
...     "Classification": BinaryAccuracy(),
...     "Regression": MeanSquaredError()
... })
>>> metrics.update(preds, targets)
>>> metrics.compute()
{'Classification': tensor(0.3333), 'Regression': tensor(0.8333)}
```

**Example (with several metrics per task):**

```

>>> import torch
>>> from torchmetrics import MetricCollection
>>> from torchmetrics.wrappers import MultitaskWrapper
>>> from torchmetrics.regression import MeanSquaredError, MeanAbsoluteError
>>> from torchmetrics.classification import BinaryAccuracy, BinaryF1Score
>>>
>>> classification_target = torch.tensor([0, 1, 0])
>>> regression_target = torch.tensor([2.5, 5.0, 4.0])
>>> targets = {"Classification": classification_target, "Regression":
↳ regression_target}
>>>
>>> classification_preds = torch.tensor([0, 0, 1])
>>> regression_preds = torch.tensor([3.0, 5.0, 2.5])
>>> preds = {"Classification": classification_preds, "Regression": regression_
↳ preds}
>>>
>>> metrics = MultitaskWrapper({
...     "Classification": MetricCollection(BinaryAccuracy(), BinaryF1Score()),
...     "Regression": MetricCollection(MeanSquaredError(), MeanAbsoluteError())
... })
>>> metrics.update(preds, targets)
>>> metrics.compute()
{'Classification': {'BinaryAccuracy': tensor(0.3333), 'BinaryF1Score': tensor(0.
↳ )},
 'Regression': {'MeanSquaredError': tensor(0.8333), 'MeanAbsoluteError':
↳ tensor(0.6667)}}

```

**compute()**

Compute metrics for all tasks.

**Return type**

`Dict[str, Any]`

**forward(task\_preds, task\_targets)**

Call underlying forward methods for all tasks and return the result as a dictionary.

**Return type**

`Dict[str, Any]`

**plot(val=None, axes=None)**

Plot a single or multiple values from the metric.

All tasks' results are plotted on individual axes.

**Parameters**

- **val** (`Union[Dict, Sequence[Dict], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **axes** (`Optional[Sequence[Axes]]`) – Sequence of matplotlib axis objects. If provided, will add the plots to the provided axis objects. If not provided, will create them.

**Return type**

`Sequence[Tuple[Figure, Union[Axes, ndarray]]]`

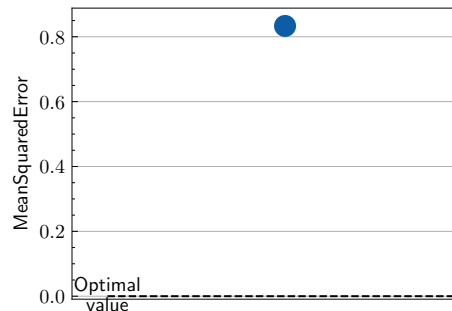
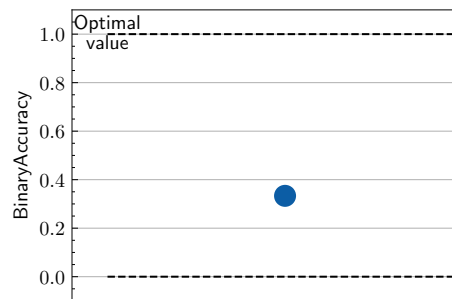
**Returns**

Sequence of tuples with Figure and Axes object for each task.

```

>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.wrappers import MultitaskWrapper
>>> from torchmetrics.regression import MeanSquaredError
>>> from torchmetrics.classification import BinaryAccuracy
>>>
>>> classification_target = torch.tensor([0, 1, 0])
>>> regression_target = torch.tensor([2.5, 5.0, 4.0])
>>> targets = {"Classification": classification_target, "Regression":
↳ regression_target}
>>>
>>> classification_preds = torch.tensor([0, 0, 1])
>>> regression_preds = torch.tensor([3.0, 5.0, 2.5])
>>> preds = {"Classification": classification_preds, "Regression": regression_
↳ preds}
>>>
>>> metrics = MultitaskWrapper({
...     "Classification": BinaryAccuracy(),
...     "Regression": MeanSquaredError()
... })
>>> metrics.update(preds, targets)
>>> value = metrics.compute()
>>> fig_, ax_ = metrics.plot(value)

```



```

>>> # Example plotting multiple values
>>> import torch

```

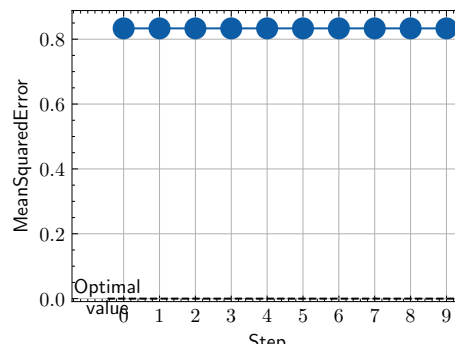
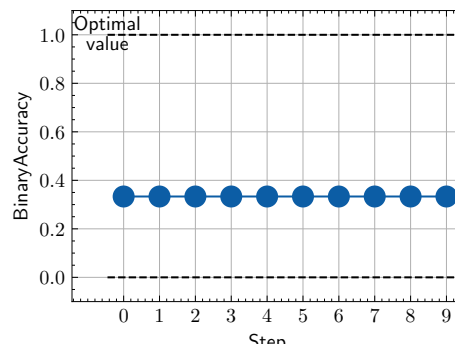
(continues on next page)

(continued from previous page)

```

>>> from torchmetrics.wrappers import MultitaskWrapper
>>> from torchmetrics.regression import MeanSquaredError
>>> from torchmetrics.classification import BinaryAccuracy
>>>
>>> classification_target = torch.tensor([0, 1, 0])
>>> regression_target = torch.tensor([2.5, 5.0, 4.0])
>>> targets = {"Classification": classification_target, "Regression":
↳ regression_target}
>>>
>>> classification_preds = torch.tensor([0, 0, 1])
>>> regression_preds = torch.tensor([3.0, 5.0, 2.5])
>>> preds = {"Classification": classification_preds, "Regression": regression_
↳ preds}
>>>
>>> metrics = MultitaskWrapper({
...     "Classification": BinaryAccuracy(),
...     "Regression": MeanSquaredError()
... })
>>> values = []
>>> for _ in range(10):
...     values.append(metrics(preds, targets))
>>> fig_, ax_ = metrics.plot(values)

```

**reset()**

Reset all underlying metrics.

**Return type**

None

**update**(*task\_preds*, *task\_targets*)

Update each task's metric with its corresponding pred and target.

#### Parameters

- **task\_preds** (`Dict[str, Tensor]`) – Dictionary associating each task to a Tensor of pred.
- **task\_targets** (`Dict[str, Tensor]`) – Dictionary associating each task to a Tensor of target.

#### Return type

None

## 1.133 Running

### 1.133.1 Module Interface

**class** `torchmetrics.wrappers.Running`(*base\_metric*, *window=5*)

Running wrapper for metrics.

Using this wrapper allows for calculating metrics over a running window of values, instead of the whole history of values. This is beneficial when you want to get a better estimate of the metric during training and don't want to wait for the whole training to finish to get epoch level estimates.

The running window is defined by the *window* argument. The window is a fixed size and this wrapper will store a duplicate of the underlying metric state for each value in the window. Thus memory usage will increase linearly with window size. Use accordingly. Also note that the running only works with metrics that have the *full\_state\_update* set to *False*.

Importantly, the wrapper does not alter the value of the *forward* method of the underlying metric. Thus, forward will still return the value on the current batch. To get the running value call *compute* instead.

#### Parameters

- **base\_metric** (*Metric*) – The metric to wrap.
- **window** (`int`) – The size of the running window.

#### Example

```
# Single metric >>> from torch import tensor >>> from torchmetrics.wrappers import Running >>>
from torchmetrics.aggregation import SumMetric >>> metric = Running(SumMetric(), window=3)
>>> for i in range(6): ... current_val = metric(tensor([i])) ... running_val = metric.compute()
... total_val = tensor(sum(list(range(i+1)))) # value we would get from compute without running ...
print(f'{current_val=}, {running_val=}, {total_val=}') current_val=tensor(0.), running_val=tensor(0.), to-
tal_val=tensor(0) current_val=tensor(1.), running_val=tensor(1.), total_val=tensor(1) current_val=tensor(2.),
running_val=tensor(3.), total_val=tensor(3) current_val=tensor(3.), running_val=tensor(6.), total_val=tensor(6)
current_val=tensor(4.), running_val=tensor(9.), total_val=tensor(10) current_val=tensor(5.), run-
ning_val=tensor(12.), total_val=tensor(15)
```

## Example

```
# Metric collection >>> from torch import tensor >>> from torchmetrics.wrappers import Running >>>
from torchmetrics import MetricCollection >>> from torchmetrics.aggregation import SumMetric, MeanMetric
>>> # note that running is input to collection, not the other way >>> metric = MetricCollection({"sum":
Running(SumMetric(), 3), "mean": Running(MeanMetric(), 3)}) >>> for i in range(6): ... current_val =
metric(tensor([i])) ... running_val = metric.compute() ... print(f'{current_val=}, {running_val=}') cur-
rent_val={'mean': tensor(0.), 'sum': tensor(0.)}, running_val={'mean': tensor(0.), 'sum': tensor(0.)} cur-
rent_val={'mean': tensor(1.), 'sum': tensor(1.)}, running_val={'mean': tensor(0.5000), 'sum': tensor(1.)}
current_val={'mean': tensor(2.), 'sum': tensor(2.)}, running_val={'mean': tensor(1.), 'sum': tensor(3.)} cur-
rent_val={'mean': tensor(3.), 'sum': tensor(3.)}, running_val={'mean': tensor(2.), 'sum': tensor(6.)} cur-
rent_val={'mean': tensor(4.), 'sum': tensor(4.)}, running_val={'mean': tensor(3.), 'sum': tensor(9.)} cur-
rent_val={'mean': tensor(5.), 'sum': tensor(5.)}, running_val={'mean': tensor(4.), 'sum': tensor(12.)}
```

**forward**(\*args, \*\*kwargs)

Forward input to the underlying metric and save state afterwards.

**Return type**

Any

**plot**(val=None, ax=None)

Plot a single or multiple values from the metric.

**Parameters**

- **val** (`Union[Tensor, Sequence\[Tensor\], None]`) – Either a single result from calling *metric.forward* or *metric.compute* or a list of these results. If no value is provided, will automatically call *metric.compute* and plot that result.
- **ax** (`Optional[Axes]`) – An matplotlib axis object. If provided will add plot to that axis

**Return type**

`Tuple[Figure, Union[Axes, ndarray]]`

**Returns**

Figure and Axes object

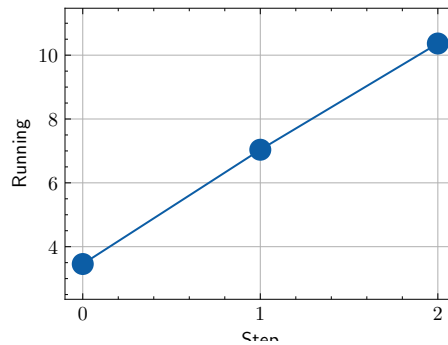
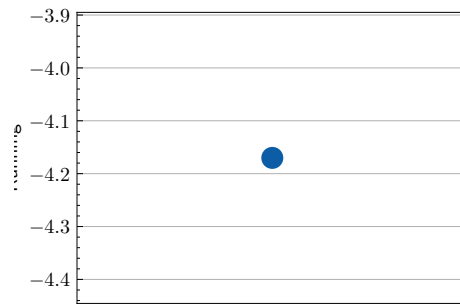
**Raises**

**ModuleNotFoundError** – If *matplotlib* is not installed

```
>>> # Example plotting a single value
>>> import torch
>>> from torchmetrics.wrappers import Running
>>> from torchmetrics.aggregation import SumMetric
>>> metric = Running(SumMetric(), 2)
>>> metric.update(torch.randn(20, 2))
>>> fig_, ax_ = metric.plot()
```

```
>>> # Example plotting multiple values
>>> import torch
>>> from torchmetrics.wrappers import Running
>>> from torchmetrics.aggregation import SumMetric
>>> metric = Running(SumMetric(), 2)
>>> values = [ ]
>>> for _ in range(3):
...     values.append(metric(torch.randn(20, 2)))
>>> fig_, ax_ = metric.plot(values)
```





**reset()**

Reset metric.

**Return type**

`None`

## 1.134 torchmetrics.Metric

The base `Metric` class is an abstract base class that are used as the building block for all other Module metrics.

**class** `torchmetrics.Metric(**kwargs)`

Base class for all metrics present in the Metrics API.

This class is inherited by all metrics and implements the following functionality: 1. Handles the transfer of metric states to correct device 2. Handles the synchronization of metric states across processes

The three core methods of the base class are \* `add_state()` \* `forward()` \* `reset()`

which should almost never be overwritten by child classes. Instead, the following methods should be overwritten \* `update()` \* `compute()`

**Parameters**

**kwargs** (`Any`) – additional keyword arguments, see [Advanced metric settings](#) for more info.

- `compute_on_cpu`: If metric state should be stored on CPU during computations. Only works for list states.
- `dist_sync_on_step`: If metric state should synchronize on `forward()`. Default is `False`

- `process_group`: The process group on which the synchronization is called. Default is the world.
- `dist_sync_fn`: Function that performs the allgather option on the metric state. Default is a custom implementation that calls `torch.distributed.all_gather` internally.
- `distributed_available_fn`: Function that checks if the distributed backend is available. Defaults to a check of `torch.distributed.is_available()` and `torch.distributed.is_initialized()`.
- `sync_on_compute`: If metric state should synchronize when `compute` is called. Default is `True`
- `compute_with_cache`: If results from `compute` should be cached. Default is `False`

**add\_state**(*name, default, dist\_reduce\_fx=None, persistent=False*)

Add metric state variable. Only used by subclasses.

Metric state variables are either `:class:`~torch.Tensor`` or an empty list, which can be appended to by the metric. Each state variable must have a unique name associated with it. State variables are accessible as attributes of the metric i.e, if name is "my\_state" then its value can be accessed from an instance `metric` as `metric.my_state`. Metric states behave like buffers and parameters of `Module` as they are also updated when `.to()` is called. Unlike parameters and buffers, metric states are not by default saved in the modules `state_dict`.

#### Parameters

- **name** (`str`) – The name of the state variable. The variable will then be accessible at `self.name`.
- **default** (`Union[list, Tensor]`) – Default value of the state; can either be a `Tensor` or an empty list. The state will be reset to this value when `self.reset()` is called.
- **dist\_reduce\_fx** (`Optional`) – Function to reduce state across multiple processes in distributed mode. If value is "sum", "mean", "cat", "min" or "max" we will use `torch.sum`, `torch.mean`, `torch.cat`, `torch.min` and `torch.max` respectively, each with argument `dim=0`. Note that the "cat" reduction only makes sense if the state is a list, and not a tensor. The user can also pass a custom function in this parameter.
- **persistent** (`Optional`) – whether the state will be saved as part of the modules `state_dict`. Default is `False`.

#### Return type

`None`

---

**Note:** Setting `dist_reduce_fx` to `None` will return the metric state synchronized across different processes. However, there won't be any reduction function applied to the synchronized metric state.

The metric states would be synced as follows

- If the metric state is `Tensor`, the synced value will be a stacked `Tensor` across the process dimension if the metric state was a `Tensor`. The original `Tensor` metric state retains dimension and hence the synchronized output will be of shape `(num_process, ...)`.
- If the metric state is a `list`, the synced value will be a `list` containing the combined elements from all processes.

---

**Note:** When passing a custom function to `dist_reduce_fx`, expect the synchronized metric state to

follow the format discussed in the above note.

#### Raises

- **ValueError** – If default is not a tensor or an empty list.
- **ValueError** – If `dist_reduce_fx` is not callable or one of "mean", "sum", "cat", "min", "max" or None.

#### `clone()`

Make a copy of the metric.

#### Return type

*Metric*

#### `abstract compute()`

Override this method to compute the final metric value.

This method will automatically synchronize state variables when running in distributed backend.

#### Return type

*Any*

#### `double()`

Override default and prevent dtype casting.

Please use `Metric.set_dtype()` instead.

#### Return type

*Metric*

#### `float()`

Override default and prevent dtype casting.

Please use `Metric.set_dtype()` instead.

#### Return type

*Metric*

#### `forward(*args, **kwargs)`

Aggregate and evaluate batch input directly.

Serves the dual purpose of both computing the metric on the current batch of inputs but also add the batch statistics to the overall accumulating metric state. Input arguments are the exact same as corresponding update method. The returned output is the exact same as the output of `compute`.

#### Parameters

- **args** (*Any*) – Any arguments as required by the metric update method.
- **kwargs** (*Any*) – Any keyword arguments as required by the metric update method.

#### Return type

*Any*

#### Returns

The output of the `compute` method evaluated on the current batch.

#### Raises

**TorchMetricsUserError** – If the metric is already synced and `forward` is called again.

**half()**

Override default and prevent dtype casting.

Please use `Metric.set_dtype()` instead.

**Return type**

`Metric`

**persistent(mode=False)**

Change post-init if metric states should be saved to its state\_dict.

**Return type**

`None`

**plot(\*\_, \*\*\_\_)**

Override this method plot the metric value.

**Return type**

`Any`

**reset()**

Reset metric state variables to their default value.

**Return type**

`None`

**set\_dtype(dst\_type)**

Transfer all metric state to specific dtype. Special version of standard `type` method.

**Parameters**

**dst\_type** (`type` or `string`) – the desired type.

**Return type**

`Metric`

**state\_dict(destination=None, prefix="", keep\_vars=False)**

Get the current state of metric as an dictionary.

**Parameters**

- **destination** (`Optional[Dict[str, Any]]`) – Optional dictionary, that if provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned.
- **prefix** (`str`) – optional string, a prefix added to parameter and buffer names to compose the keys in state\_dict.
- **keep\_vars** (`bool`) – by default the If set to `True`, detaching will not be performed.

**Return type**

`Dict[str, Any]`

**sync(dist\_sync\_fn=None, process\_group=None, should\_sync=True, distributed\_available=None)**

Sync function for manually controlling when metrics states should be synced across processes.

**Parameters**

- **dist\_sync\_fn** (`Optional[Callable]`) – Function to be used to perform states synchronization
- **process\_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)

- **should\_sync** (`bool`) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.
- **distributed\_available** (`Optional[Callable]`) – Function to determine if we are running inside a distributed setting

**Raises**

**TorchMetricsUserError** – If the metric is already synced and `sync` is called again.

**Return type**

`None`

**sync\_context** (`dist_sync_fn=None, process_group=None, should_sync=True, should_unsync=True, distributed_available=None`)

Context manager to synchronize states.

This context manager is used in distributed setting and makes sure that the local cache states are restored after yielding the synchronized state.

**Parameters**

- **dist\_sync\_fn** (`Optional[Callable]`) – Function to be used to perform states synchronization
- **process\_group** (`Optional[Any]`) – Specify the process group on which synchronization is called. default: `None` (which selects the entire world)
- **should\_sync** (`bool`) – Whether to apply to state synchronization. This will have an impact only when running in a distributed setting.
- **should\_unsync** (`bool`) – Whether to restore the cache state so that the metrics can continue to be accumulated.
- **distributed\_available** (`Optional[Callable]`) – Function to determine if we are running inside a distributed setting

**Return type**

`Generator`

**type** (`dst_type`)

Override default and prevent dtype casting.

Please use `Metric.set_dtype()` instead.

**Return type**

`Metric`

**unsync** (`should_unsync=True`)

Unsync function for manually controlling when metrics states should be reverted back to their local states.

**Parameters**

- **should\_unsync** (`bool`) – Whether to perform unsync

**Return type**

`None`

**abstract update** (`*, ** __`)

Override this method to update the state variables of your metric class.

**Return type**

`None`

**property device:** `device`

Return the device of the metric.

**property update\_called:** `bool`

Returns *True* if *update* or *forward* has been called initialization or last *reset*.

**property update\_count:** `int`

Get the number of times *update* and/or *forward* has been called since initialization or last *reset*.

## 1.135 torchmetrics.utilities.data

### 1.135.1 select\_topk

`torchmetrics.utilities.data.select_topk(prob_tensor, topk=1, dim=1)`

Convert a probability tensor to binary by selecting top-k the highest entries.

#### Parameters

- **prob\_tensor** (`Tensor`) – dense tensor of shape  $[\dots, C, \dots]$ , where  $C$  is in the position defined by the `dim` argument
- **topk** (`int`) – number of the highest entries to turn into 1s
- **dim** (`int`) – dimension on which to compare entries

#### Return type

`Tensor`

#### Returns

A binary tensor of the same shape as the input tensor of type `torch.int32`

#### Example

```
>>> x = torch.tensor([[1.1, 2.0, 3.0], [2.0, 1.0, 0.5]])
>>> select_topk(x, topk=2)
tensor([[0, 1, 1],
        [1, 1, 0]], dtype=torch.int32)
```

### 1.135.2 to\_categorical

`torchmetrics.utilities.data.to_categorical(x, argmax_dim=1)`

Convert a tensor of probabilities to a dense label tensor.

#### Parameters

- **x** (`Tensor`) – probabilities to get the categorical label  $[N, d1, d2, \dots]$
- **argmax\_dim** (`int`) – dimension to apply

#### Return type

`Tensor`

#### Returns

A tensor with categorical labels  $[N, d2, \dots]$

### Example

```
>>> x = torch.tensor([[0.2, 0.5], [0.9, 0.1]])
>>> to_categorical(x)
tensor([1, 0])
```

## 1.135.3 to\_onehot

`torchmetrics.utilities.data.to_onehot(label_tensor, num_classes=None)`

Convert a dense label tensor to one-hot format.

#### Parameters

- **label\_tensor** (`Tensor`) – dense label tensor, with shape  $[N, d1, d2, \dots]$
- **num\_classes** (`Optional[int]`) – number of classes  $C$

#### Return type

`Tensor`

#### Returns

A sparse label tensor with shape  $[N, C, d1, d2, \dots]$

### Example

```
>>> x = torch.tensor([1, 2, 3])
>>> to_onehot(x)
tensor([[0, 1, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]])
```

## 1.136 TorchMetrics Governance

This document describes governance processes we follow in developing TorchMetrics.

### 1.136.1 Persons of Interest

#### Leads

- Nicki Skafte ([skaftenicki](#))
- Jirka Borovec ([Borda](#))
- Justus Schock ([justusschock](#))

## Core Maintainers

- Daniel Stancil ([stancld](#))
- Luca Di Liello ([lucadiliello](#))
- Changsheng Quan ([quancs](#))

## Alumni

- Ananya Harsh Jha ([ananyahjha93](#))
- Teddy Koker ([teddykoker](#))
- Maxim Grechkin ([maximsch2](#))

## 1.136.2 Releases

We release a new minor version (e.g., 0.5.0) every few months and bugfix releases if needed. The minor versions contain new features, API changes, deprecations, removals, potential backward-incompatible changes and also all previous bugfixes included in any bugfix release. With every release, we publish a changelog where we list additions, removals, changed functionality and fixes.

## 1.136.3 Project Management and Decision Making

The decision what goes into a release is governed by the *staff contributors and leaders* of TorchMetrics development. Whenever possible, discussion happens publicly on GitHub and includes the whole community. When a consensus is reached, staff and core contributors assign milestones and labels to the issue and/or pull request and start tracking the development. It is possible that priorities change over time.

Commits to the project are exclusively to be added by pull requests on GitHub and anyone in the community is welcome to review them. However, reviews submitted by *code owners* have higher weight and it is necessary to get the approval of code owners before a pull request can be merged. Additional requirements may apply case by case.

## 1.136.4 API Evolution

TorchMetrics development is driven by research and best practices in a rapidly developing field of AI and machine learning. Change is inevitable and when it happens, the Torchmetric team is committed to minimizing user friction and maximizing ease of transition from one version to the next. We take backward compatibility and reproducibility very seriously.

For API removal, renaming or other forms of backward-incompatible changes, the procedure is:

1. A deprecation process is initiated at version X, producing warning messages at runtime and in the documentation.
2. Calls to the deprecated API remain unchanged in their function during the deprecation phase.
3. One minor versions in the future at version X+1 the breaking change takes effect.

The “X+1” rule is a recommendation and not a strict requirement. Longer deprecation cycles may apply for some cases.



## 1.137 Contributor Covenant Code of Conduct

### 1.137.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

### 1.137.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

### 1.137.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

### 1.137.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

### 1.137.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at [waf2107@columbia.edu](mailto:waf2107@columbia.edu). All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

### 1.137.6 Attribution

This Code of Conduct is adapted from the [Contributor Covenant](https://www.contributor-covenant.org/version/1/4/code-of-conduct.html), version 1.4, available at <https://www.contributor-covenant.org/version/1/4/code-of-conduct.html>

For answers to common questions about this code of conduct, see <https://www.contributor-covenant.org/faq>

## 1.138 Contributing

Welcome to the Torchmetrics community! We're building largest collection of native pytorch metrics, with the goal of reducing boilerplate and increasing reproducibility.

### 1.138.1 Contribution Types

We are always looking for help implementing new features or fixing bugs.

#### Bug Fixes:

1. If you find a bug please submit a github issue.
  - Make sure the title explains the issue.
  - Describe your setup, what you are trying to do, expected vs. actual behaviour. Please add configs and code samples.
  - Add details on how to reproduce the issue - a minimal test case is always best, colab is also great. Note, that the sample code shall be minimal and if needed with publicly available data.
2. Try to fix it or recommend a solution. We highly recommend to use test-driven approach:
  - Convert your minimal code example to a unit/integration test with assert on expected results.
  - Start by debugging the issue... You can run just this particular test in your IDE and draft a fix.
  - Verify that your test case fails on the master branch and only passes with the fix applied.
3. Submit a PR!

***Note**, even if you do not find the solution, sending a PR with a test covering the issue is a valid contribution and we can help you or finish it with you :]*

### New Features:

1. Submit a github issue - describe what is the motivation of such feature (adding the use case or an example is helpful).
2. Let's discuss to determine the feature scope.
3. Submit a PR! We recommend test driven approach to adding new features as well:
  - Write a test for the functionality you want to add.
  - Write the functional code until the test passes.
4. Add/update the relevant tests!
  - [This PR](#) is a good example for adding a new metric

### Test cases:

Want to keep Torchmetrics healthy? Love seeing those green tests? So do we! How to we keep it that way? We write tests! We value tests contribution even more than new features. One of the core values of torchmetrics is that our users can trust our metric implementation. We can only guarantee this if our metrics are well tested.

---

## 1.138.2 Guidelines

### Developments scripts

To build the documentation locally, simply execute the following commands from project root (only for Unix):

- `make clean` cleans repo from temp/generated files
- `make docs` builds documentation under `docs/build/html`
- `make test` runs all project's tests with coverage

### Original code

All added or edited code shall be the own original work of the particular contributor. If you use some third-party implementation, all such blocks/functions/modules shall be properly referred and if possible also agreed by code's author. For example - This code is inspired from <http://...>. In case you adding new dependencies, make sure that they are compatible with the actual Torchmetrics license (ie. dependencies should be *at least* as permissive as the Torchmetrics license).

### Coding Style

1. Use f-strings for output formation (except logging when we stay with lazy `logging.info("Hello %s!", name)`).
2. You can use `pre-commit` to make sure your code style is correct.

## Documentation

We are using Sphinx with Napoleon extension. Moreover, we set Google style to follow with type convention.

- Napoleon formatting with Google style
- ReStructured Text (reST)
- Paragraph-level markup

See following short example of a sample function taking one position string and optional

```
from typing import Optional

def my_func(param_a: int, param_b: Optional[float] = None) -> str:
    """Sample function.

    Args:
        param_a: first parameter
        param_b: second parameter

    Return:
        sum of both numbers

    Example:
        Sample doctest example...
        >>> my_func(1, 2)
        3

    .. note:: If you want to add something.
    """
    p = param_b if param_b else 0
    return str(param_a + p)
```

When updating the docs make sure to build them first locally and visually inspect the html files (in the browser) for formatting errors. In certain cases, a missing blank line or a wrong indent can lead to a broken layout. Run these commands

```
make docs
```

and open docs/build/html/index.html in your browser.

Notes:

- You need to have LaTeX installed for rendering math equations. You can for example install TeXLive by doing one of the following:
  - on Ubuntu (Linux) run `apt-get install texlive` or otherwise follow the instructions on the TeXLive website
  - use the [RTD docker image](#)
- with PL used class meta you need to use python 3.7 or higher

When you send a PR the continuous integration will run tests and build the docs.

## Testing

**Local:** Testing your work locally will help you speed up the process since it allows you to focus on particular (failing) test-cases. To setup a local development environment, install both local and test dependencies:

```
python -m pip install -r requirements/test.txt
python -m pip install pre-commit
```

You can run the full test-case in your terminal via this make script:

```
make test
# or natively
python -m pytest torchmetrics tests
```

Note: if your computer does not have multi-GPU nor TPU these tests are skipped.

**GitHub Actions:** For convenience, you can also use your own GHActions building which will be triggered with each commit. This is useful if you do not test against all required dependency versions.

## 1.139 Changelog

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

**Note: we move fast, but still we preserve 0.1 version (one feature release) back compatibility.**

### 1.139.1 [1.0.0] - 2022-07-04

#### [1.0.0] - Added

- Added `prefix` and `postfix` arguments to `ClasswiseWrapper` (#1866)
- Added speech-to-reverberation modulation energy ratio (SRMR) metric (#1792, #1872)
- Added new global arg `compute_with_cache` to control caching behaviour after `compute` method (#1754)
- Added `ComplexScaleInvariantSignalNoiseRatio` for audio package (#1785)
- Added `Running` wrapper for calculate running statistics (#1752)
- Added `RelativeAverageSpectralError` and `RootMeanSquaredErrorUsingSlidingWindow` to image package (#816)
- Added support for `SpecificityAtSensitivity` Metric (#1432)
- Added support for plotting of metrics through `.plot()` method ( #1328, #1481, #1480, #1490, #1581, #1585, #1593, #1600, #1605, #1610, #1609, #1621, #1624, #1623, #1638, #1631, #1650, #1639, #1660, #1682, #1786, )
- Added support for plotting of audio metrics through `.plot()` method (#1434)
- Added classes to output from MAP metric (#1419)
- Added Binary group fairness metrics to classification package (#1404)
- Added `MinkowskiDistance` to regression package (#1362)
- Added `pairwise_minkowski_distance` to pairwise package (#1362)

- Added new detection metric `PanopticQuality` ([#929](#), [#1527](#), )
- Added PSNRB metric ([#1421](#))
- Added `ClassificationTask` Enum and use in metrics ([#1479](#))
- Added `ignore_index` option to `exact_match` metric ([#1540](#))
- Add parameter `top_k` to `RetrievalMAP` ([#1501](#))
- Added support for deterministic evaluation on GPU for metrics that uses `torch.cumsum` operator ([#1499](#))
- Added support for plotting of aggregation metrics through `.plot()` method ([#1485](#))
- Added support for python 3.11 ([#1612](#))
- Added support for auto clamping of input for metrics that uses the `data_range` ([#1606](#))(argument <https://github.com/Lightning-AI/metrics/pull/1606>)
- Added `ModifiedPanopticQuality` metric to detection package ([#1627](#))
- Added `PrecisionAtFixedRecall` metric to classification package ([#1683](#))
- Added multiple metrics to detection package ([#1284](#))
  - `IntersectionOverUnion`
  - `GeneralizedIntersectionOverUnion`
  - `CompleteIntersectionOverUnion`
  - `DistanceIntersectionOverUnion`
- Added `MultitaskWrapper` to wrapper package ([#1762](#))
- Added `RelativeSquaredError` metric to regression package ([#1765](#))
- Added `MemorizationInformedFrechetInceptionDistance` metric to image package ([#1580](#))

## [1.0.0] - Changed

- Changed `permutation_invariant_training` to allow using a 'permutation-wise' metric function ([#1794](#))
- Changed `update_count` and `update_called` from private to public methods ([#1370](#))
- Raise exception for invalid kwargs in `Metric` base class ([#1427](#))
- Extend `EnumStr` raising `ValueError` for invalid value ([#1479](#))
- Improve speed and memory consumption of binned `PrecisionRecallCurve` with large number of samples ([#1493](#))
- Changed `__iter__` method from raising `NotImplementedError` to `TypeError` by setting to `None` ([#1538](#))
- FID metric will now raise an error if too few samples are provided ([#1655](#))
- Allowed FID with `torch.float64` ([#1628](#))
- Changed LPIPS implementation to no more rely on third-party package ([#1575](#))
- Changed FID matrix square root calculation from `scipy` to `torch` ([#1708](#))
- Changed calculation in `PearsonCorrCoeff` to be more robust in certain cases ([#1729](#))
- Changed `MeanAveragePrecision` to `pycocotools` backend ([#1832](#))

### [1.0.0] - Deprecated

- Deprecated domain metrics import from package root ( [#1685](#), [#1694](#), [#1696](#), [#1699](#), [#1703](#), )

### [1.0.0] - Removed

- Support for python 3.7 ([#1640](#))

### [1.0.0] - Fixed

- Fixed support in `MetricTracker` for `MultioutputWrapper` and nested structures ([#1608](#))
- Fixed restrictive check in `PearsonCorrCoef` ([#1649](#))
- Fixed integration with `jsonargparse` and `LightningCLI` ([#1651](#))
- Fixed corner case in calibration error for zero confidence input ([#1648](#))
- Fix precision-recall curve based computations for float target ([#1642](#))
- Fixed missing kwarg squeeze in `MultiOutputWrapper` ([#1675](#))
- Fixed padding removal for 3d input in `MSSSIM` ([#1674](#))
- Fixed `max_det_threshold` in MAP detection ([#1712](#))
- Fixed states being saved in metrics that use `register_buffer` ([#1728](#))
- Fixed states not being correctly synced and device transfered in `MeanAveragePrecision` for `iou_type="segm"` ([#1763](#))
- Fixed use of `prefix` and `postfix` in nested `MetricCollection` ([#1773](#))
- Fixed `ax` plotting logging in `MetricCollection` ([#1783](#))
- Fixed lookup for punkt sources being downloaded in `RougeScore` ([#1789](#))
- Fixed integration with lightning for `CompositionalMetric` ([#1761](#))
- Fixed several bugs in `SpectralDistortionIndex` metric ([#1808](#))
- Fixed bug for corner cases in `MatthewsCorrCoef` ( [#1812](#), [#1863](#) )
- Fixed support for half precision in `PearsonCorrCoef` ([#1819](#))
- Fixed number of bugs related to `average="macro"` in classification metrics ([#1821](#))
- Fixed off-by-one issue when `ignore_index = num_classes + 1` in Multiclass-jaccard ([#1860](#))

## 1.139.2 [0.11.4] - 2023-03-10

### [0.11.4] - Fixed

- Fixed evaluation of `R2Score` with near constant target ([#1576](#))
- Fixed dtype conversion when metric is submodule ([#1583](#))
- Fixed bug related to `top_k>1` and `ignore_index!=None` in `StatScores` based metrics ([#1589](#))
- Fixed corner case for `PearsonCorrCoef` when running in `ddp` mode but only on single device ([#1587](#))
- Fixed overflow error for specific cases in `MAP` when big areas are calculated ([#1607](#))

### 1.139.3 [0.11.3] - 2023-02-28

#### [0.11.3] - Fixed

- Fixed classification metrics for byte input (#1521)
- Fixed the use of `ignore_index` in `MulticlassJaccardIndex` (#1386)

### 1.139.4 [0.11.2] - 2023-02-21

#### [0.11.2] - Fixed

- Fixed compatibility between XLA in `_bincount` function (#1471)
- Fixed type hints in methods belonging to `MetricTracker` wrapper (#1472)
- Fixed `multilabel` in `ExactMatch` (#1474)

### 1.139.5 [0.11.1] - 2023-01-30

#### [0.11.1] - Fixed

- Fixed type checking on the `maximize` parameter at the initialization of `MetricTracker` (#1428)
- Fixed mixed precision autocast for SSIM metric (#1454)
- Fixed checking for `nlTK.punkt` in `RougeScore` if a machine is not online (#1456)
- Fixed wrongly reset method in `MultioutputWrapper` (#1460)
- Fixed dtype checking in `PrecisionRecallCurve` for `target` tensor (#1457)

### 1.139.6 [0.11.0] - 2022-11-30

#### [0.11.0] - Added

- Added `MulticlassExactMatch` to classification metrics (#1343)
- Added `TotalVariation` to image package (#978)
- Added `CLIPScore` to new multimodal package (#1314)
- Added regression metrics:
  - `KendallRankCorrCoef` (#1271)
  - `LogCoshError` (#1316)
- Added new nominal metrics:
  - `CramersV` (#1298)
  - `PearsonsContingencyCoefficient` (#1334)
  - `TschuprowsT` (#1334)
  - `TheilsU` (#1337)
- Added option to pass `distributed_available_fn` to metrics to allow checks for custom communication backend for making `dist_sync_fn` actually useful (#1301)



- Added `normalize` argument to Inception, FID, KID metrics (#1246)

#### [0.11.0] - Changed

- Changed minimum Pytorch version to be 1.8 (#1263)
- Changed interface for all functional and modular classification metrics after refactor (#1252)

#### [0.11.0] - Removed

- Removed deprecated `BinnedAveragePrecision`, `BinnedPrecisionRecallCurve`, `RecallAtFixedPrecision` (#1251)
- Removed deprecated `LabelRankingAveragePrecision`, `LabelRankingLoss` and `CoverageError` (#1251)
- Removed deprecated `KLDivergence` and `AUC` (#1251)

#### [0.11.0] - Fixed

- Fixed precision bug in `pairwise_euclidean_distance` (#1352)

### 1.139.7 [0.10.3] - 2022-11-16

#### [0.10.3] - Fixed

- Fixed bug in `MetricTracker.best_metric` when `return_step=False` (#1306)
- Fixed bug to prevent users from going into an infinite loop if trying to iterate of a single metric (#1320)

### 1.139.8 [0.10.2] - 2022-10-31

#### [0.10.2] - Changed

- Changed in-place operation to out-of-place operation in `pairwise_cosine_similarity` (#1288)

#### [0.10.2] - Fixed

- Fixed high memory usage for certain classification metrics when `average='micro'` (#1286)
- Fixed precision problems when `structural_similarity_index_measure` was used with `autocast` (#1291)
- Fixed slow performance for confusion matrix based metrics (#1302)
- Fixed restrictive dtype checking in `spearman_corrcoef` when used with `autocast` (#1303)

## 1.139.9 [0.10.1] - 2022-10-21

### [0.10.1] - Fixed

- Fixed broken clone method for classification metrics (#1250)
- Fixed unintentional downloading of `nlTK.punkt` when `lsum` not in `rouge_keys` (#1258)
- Fixed type casting in MAP metric between `bool` and `float32` (#1150)

## 1.139.10 [0.10.0] - 2022-10-04

### [0.10.0] - Added

- Added a new NLP metric `InfoLM` (#915)
- Added `Perplexity` metric (#922)
- Added `ConcordanceCorrCoef` metric to regression package (#1201)
- Added argument `normalize` to `LPIPS` metric (#1216)
- Added support for multiprocessing of batches in `PESQ` metric (#1227)
- Added support for multioutput in `PearsonCorrCoef` and `SpearmanCorrCoef` (#1200)

### [0.10.0] - Changed

- Classification refactor ( #1054, #1143, #1145, #1151, #1159, #1163, #1167, #1175, #1189, #1197, #1215, #1195 )
- Changed update in FID metric to be done in online fashion to save memory (#1199)
- Improved performance of retrieval metrics (#1242)
- Changed SSIM and MSSSIM update to be online to reduce memory usage (#1231)

### [0.10.0] - Deprecated

- Deprecated `BinnedAveragePrecision`, `BinnedPrecisionRecallCurve`, `BinnedRecallAtFixedPrecision` (#1163)
  - `BinnedAveragePrecision` -> use `AveragePrecision` with `thresholds` arg
  - `BinnedPrecisionRecallCurve` -> use `AveragePrecisionRecallCurve` with `thresholds` arg
  - `BinnedRecallAtFixedPrecision` -> use `RecallAtFixedPrecision` with `thresholds` arg
- Renamed and refactored `LabelRankingAveragePrecision`, `LabelRankingLoss` and `CoverageError` (#1167)
  - `LabelRankingAveragePrecision` -> `MultilabelRankingAveragePrecision`
  - `LabelRankingLoss` -> `MultilabelRankingLoss`
  - `CoverageError` -> `MultilabelCoverageError`
- Deprecated `KLDivergence` and `AUC` from classification package (#1189)
  - `KLDivergence` moved to regression package

- Instead of AUC use `torchmetrics.utils.compute_auc`

#### [0.10.0] - Fixed

- Fixed a bug in `ssim` when `return_full_image=True` where the score was still reduced (#1204)
- Fixed MPS support for:
  - MAE metric (#1210)
  - Jaccard index (#1205)
- Fixed bug in `ClasswiseWrapper` such that `compute` gave wrong result (#1225)
- Fixed synchronization of empty list states (#1219)

### 1.139.11 [0.9.3] - 2022-08-22

#### [0.9.3] - Added

- Added global option `sync_on_compute` to disable automatic synchronization when `compute` is called (#1107)

#### [0.9.3] - Fixed

- Fixed missing reset in `ClasswiseWrapper` (#1129)
- Fixed `JaccardIndex` multi-label compute (#1125)
- Fix SSIM propagate device if `gaussian_kernel` is `False`, add test (#1149)

### 1.139.12 [0.9.2] - 2022-06-29

#### [0.9.2] - Fixed

- Fixed mAP calculation for areas with 0 predictions (#1080)
- Fixed bug where avg precision state and auroc state was not merge when using `MetricCollections` (#1086)
- Skip box conversion if no boxes are present in `MeanAveragePrecision` (#1097)
- Fixed inconsistency in docs and code when setting `average="none"` in `AveragePrecision` metric (#1116)

### 1.139.13 [0.9.1] - 2022-06-08

#### [0.9.1] - Added

- Added specific `RuntimeError` when metric object is on the wrong device (#1056)
- Added an option to specify own n-gram weights for `BLEUScore` and `SacreBLEUScore` instead of using uniform weights only. (#1075)

### [0.9.1] - Fixed

- Fixed aggregation metrics when input only contains zero (#1070)
- Fixed `TypeError` when providing superclass arguments as `kwargs` (#1069)
- Fixed bug related to state reference in metric collection when using compute groups (#1076)

## 1.139.14 [0.9.0] - 2022-05-30

### [0.9.0] - Added

- Added `RetrievalPrecisionRecallCurve` and `RetrievalRecallAtFixedPrecision` to retrieval package (#951)
- Added class property `full_state_update` that determines `forward` should call `update` once or twice ( #984, #1033)
- Added support for nested metric collections (#1003)
- Added `Dice` to classification package (#1021)
- Added support to segmentation type `segm` as IOU for mean average precision (#822)

### [0.9.0] - Changed

- Renamed `reduction` argument to `average` in Jaccard score and added additional options (#874)

### [0.9.0] - Removed

- Removed deprecated `compute_on_step` argument ( #962, #967, #979, #990, #991, #993, #1005, #1004, #1007 )

### [0.9.0] - Fixed

- Fixed non-empty state dict for a few metrics (#1012)
- Fixed bug when comparing states while finding compute groups (#1022)
- Fixed `torch.double` support in stat score metrics (#1023)
- Fixed FID calculation for non-equal size real and fake input (#1028)
- Fixed case where `KLDivergence` could output `Nan` (#1030)
- Fixed deterministic for `PyTorch<1.8` (#1035)
- Fixed default value for `mdmc_average` in `Accuracy` (#1036)
- Fixed missing copy of property when using compute groups in `MetricCollection` (#1052)

### 1.139.15 [0.8.2] - 2022-05-06

#### [0.8.2] - Fixed

- Fixed multi device aggregation in `PearsonCorrCoef` (#998)
- Fixed MAP metric when using custom list of thresholds (#995)
- Fixed compatibility between compute groups in `MetricCollection` and prefix/postfix arg (#1007)
- Fixed compatibility with future Pytorch 1.12 in `safe_matmul` (#1011, #1014)

### 1.139.16 [0.8.1] - 2022-04-27

#### [0.8.1] - Changed

- Reimplemented the `signal_distortion_ratio` metric, which removed the absolute requirement of `fast-bss-eval` (#964)

#### [0.8.1] - Fixed

- Fixed “Sort currently does not support bool dtype on CUDA” error in MAP for empty preds (#983)
- Fixed `BinnedPrecisionRecallCurve` when `thresholds` argument is not provided (#968)
- Fixed `CalibrationError` to work on logit input (#985)

### 1.139.17 [0.8.0] - 2022-04-14

#### [0.8.0] - Added

- Added `WeightedMeanAbsolutePercentageError` to regression package (#948)
- Added new classification metrics:
  - `CoverageError` (#787)
  - `LabelRankingAveragePrecision` and `LabelRankingLoss` (#787)
- Added new image metric:
  - `SpectralAngleMapper` (#885)
  - `ErrorRelativeGlobalDimensionlessSynthesis` (#894)
  - `UniversalImageQualityIndex` (#824)
  - `SpectralDistortionIndex` (#873)
- Added support for `MetricCollection` in `MetricTracker` (#718)
- Added support for 3D image and uniform kernel in `StructuralSimilarityIndexMeasure` (#818)
- Added smart update of `MetricCollection` (#709)
- Added `ClasswiseWrapper` for better logging of classification metrics with multiple output values (#832)
- Added `**kwargs` argument for passing additional arguments to base class (#833)
- Added negative `ignore_index` for the Accuracy metric (#362)

- Added `adaptive_k` for the `RetrievalPrecision` metric (#910)
- Added `reset_real_features` argument image quality assessment metrics (#722)
- Added new keyword argument `compute_on_cpu` to all metrics (#867)

#### [0.8.0] - Changed

- Made `num_classes` in `jaccard_index` a required argument (#853, #914)
- Added `normalizer`, `tokenizer` to `ROUGE` metric (#838)
- Improved shape checking of `permutation_invariant_training` (#864)
- Allowed reduction `None` (#891)
- `MetricTracker.best_metric` will now give a warning when computing on metric that do not have a best (#913)

#### [0.8.0] - Deprecated

- Deprecated argument `compute_on_step` (#792)
- Deprecated passing in `dist_sync_on_step`, `process_group`, `dist_sync_fn` direct argument (#833)

#### [0.8.0] - Removed

- Removed support for versions of `Pytorch-Lightning` lower than v1.5 (#788)
- Removed deprecated functions, and warnings in `Text` (#773)
  - `WER` and `functional.wer`
- Removed deprecated functions and warnings in `Image` (#796)
  - `SSIM` and `functional.ssim`
  - `PSNR` and `functional.psnr`
- Removed deprecated functions, and warnings in classification and regression (#806)
  - `FBeta` and `functional.fbeta`
  - `F1` and `functional.f1`
  - `Hinge` and `functional.hinge`
  - `IoU` and `functional.iou`
  - `MatthewsCorrcoef`
  - `PearsonCorrcoef`
  - `SpearmanCorrcoef`
- Removed deprecated functions, and warnings in detection and pairwise (#804)
  - `MAP` and `functional.pairwise.manhattan`
- Removed deprecated functions, and warnings in `Audio` (#805)
  - `PESQ` and `functional.audio.pesq`
  - `PIT` and `functional.audio.pit`

- SDR and `functional.audio.sdr` and `functional.audio.si_sdr`
- SNR and `functional.audio.snr` and `functional.audio.si_snr`
- STOI and `functional.audio.stoi`
- Removed unused `get_num_classes` from `torchmetrics.utilities.data` (#914)

#### [0.8.0] - Fixed

- Fixed device mismatch for MAP metric in specific cases (#950)
- Improved testing speed (#820)
- Fixed compatibility of `ClasswiseWrapper` with the `prefix` argument of `MetricCollection` (#843)
- Fixed `BestScore` on GPU (#912)
- Fixed `Lsum` computation for `ROUGEScore` (#944)

### 1.139.18 [0.7.3] - 2022-03-23

#### [0.7.3] - Fixed

- Fixed unsafe log operation in `TweedieDeviance` for `power=1` (#847)
- Fixed bug in MAP metric related to either no ground truth or no predictions (#884)
- Fixed `ConfusionMatrix`, `AUROC` and `AveragePrecision` on GPU when running in deterministic mode (#900)
- Fixed NaN or Inf results returned by `signal_distortion_ratio` (#899)
- Fixed memory leak when using `update` method with tensor where `requires_grad=True` (#902)

### 1.139.19 [0.7.2] - 2022-02-10

#### [0.7.2] - Fixed

- Minor patches in JOSS paper.

### 1.139.20 [0.7.1] - 2022-02-03

#### [0.7.1] - Changed

- Used `torch.bucketize` in calibration error when `torch>1.8` for faster computations (#769)
- Improve mAP performance (#742)

### [0.7.1] - Fixed

- Fixed check for available modules (#772)
- Fixed Matthews correlation coefficient when the denominator is 0 (#781)

## 1.139.21 [0.7.0] - 2022-01-17

### [0.7.0] - Added

- Added NLP metrics:
  - MatchErrorRate (#619)
  - WordInfoLost and WordInfoPreserved (#630)
  - SQuAD (#623)
  - CHRFScore (#641)
  - TranslationEditRate (#646)
  - ExtendedEditDistance (#668)
- Added MultiScaleSSIM into image metrics (#679)
- Added Signal to Distortion Ratio (SDR) to audio package (#565)
- Added MinMaxMetric to wrappers (#556)
- Added ignore\_index to retrieval metrics (#676)
- Added support for multi references in ROUGEScore (#680)
- Added a default VSCode devcontainer configuration (#621)

### [0.7.0] - Changed

- Scalar metrics will now consistently have additional dimensions squeezed (#622)
- Metrics having third party dependencies removed from global import (#463)
- Untokenized for BLEUScore input stay consistent with all the other text metrics (#640)
- Arguments reordered for TER, BLEUScore, SacreBLEUScore, CHRFScore now expect input order as predictions first and target second (#696)
- Changed dtype of metric state from torch.float to torch.long in ConfusionMatrix to accommodate larger values (#715)
- Unify preds, target input argument's naming across all text metrics (#723, #727)
  - bert, bleu, chrf, sacre\_bleu, wip, wil, cer, ter, wer, mer, rouge, squad



**[0.7.0] - Deprecated**

- Renamed IoU -> Jaccard Index (#662)
- Renamed text WER metric (#714)
  - `functional.wer` -> `functional.word_error_rate`
  - `WER` -> `WordErrorRate`
- Renamed correlation coefficient classes: (#710)
  - `MatthewsCorrcoef` -> `MatthewsCorrCoef`
  - `PearsonCorrcoef` -> `PearsonCorrCoef`
  - `SpearmanCorrcoef` -> `SpearmanCorrCoef`
- Renamed audio STOI metric: (#753, #758)
  - `audio.STOI` to `audio.ShortTimeObjectiveIntelligibility`
  - `functional.audio.stoi` to `functional.audio.short_time_objective_intelligibility`
- Renamed audio PESQ metrics: (#751)
  - `functional.audio.pesq` -> `functional.audio.perceptual_evaluation_speech_quality`
  - `audio.PESQ` -> `audio.PerceptualEvaluationSpeechQuality`
- Renamed audio SDR metrics: (#711)
  - `functional.sdr` -> `functional.signal_distortion_ratio`
  - `functional.si_sdr` -> `functional.scale_invariant_signal_distortion_ratio`
  - `SDR` -> `SignalDistortionRatio`
  - `SI_SDR` -> `ScaleInvariantSignalDistortionRatio`
- Renamed audio SNR metrics: (#712)
  - `functional.snr` -> `functional.signal_distortion_ratio`
  - `functional.si_snr` -> `functional.scale_invariant_signal_noise_ratio`
  - `SNR` -> `SignalNoiseRatio`
  - `SI_SNR` -> `ScaleInvariantSignalNoiseRatio`
- Renamed F-score metrics: (#731, #740)
  - `functional.f1` -> `functional.f1_score`
  - `F1` -> `F1Score`
  - `functional.fbeta` -> `functional.fbeta_score`
  - `FBeta` -> `FBetaScore`
- Renamed Hinge metric: (#734)
  - `functional.hinge` -> `functional.hinge_loss`
  - `Hinge` -> `HingeLoss`
- Renamed image PSNR metrics (#732)
  - `functional.psnr` -> `functional.peak_signal_noise_ratio`
  - `PSNR` -> `PeakSignalNoiseRatio`

- Renamed image PIT metric: (#737)
  - `functional.pit` -> `functional.permutation_invariant_training`
  - PIT -> `PermutationInvariantTraining`
- Renamed image SSIM metric: (#747)
  - `functional.ssim` -> `functional.scale_invariant_signal_noise_ratio`
  - SSIM -> `StructuralSimilarityIndexMeasure`
- Renamed detection MAP to MeanAveragePrecision metric (#754)
- Renamed Fidelity & LPIPS image metric: (#752)
  - `image.FID` -> `image.FrechetInceptionDistance`
  - `image.KID` -> `image.KernelInceptionDistance`
  - `image.LPIPS` -> `image.LearnedPerceptualImagePatchSimilarity`

#### [0.7.0] - Removed

- Removed `embedding_similarity` metric (#638)
- Removed argument `concatenate_texts` from `wer` metric (#638)
- Removed arguments `newline_sep` and `decimal_places` from `rouge` metric (#638)

#### [0.7.0] - Fixed

- Fixed `MetricCollection` kwargs filtering when no kwargs are present in update signature (#707)

### 1.139.22 [0.6.2] - 2021-12-15

#### [0.6.2] - Fixed

- Fixed `torch.sort` currently does not support `bool` dtype on CUDA (#665)
- Fixed `mAP` properly checks if ground truths are empty (#684)
- Fixed initialization of tensors to be on correct device for `MAP` metric (#673)

### 1.139.23 [0.6.1] - 2021-12-06

#### [0.6.1] - Changed

- Migrate `MAP` metrics from `pycocotools` to `PyTorch` (#632)
- Use `torch.topk` instead of `torch.argsort` in retrieval precision for speedup (#627)

**[0.6.1] - Fixed**

- Fix empty predictions in MAP metric (#594, #610, #624)
- Fix edge case of AUROC with average=weighted on GPU (#606)
- Fixed forward in compositional metrics (#645)

**1.139.24 [0.6.0] - 2021-10-28****[0.6.0] - Added**

- Added audio metrics:
  - Perceptual Evaluation of Speech Quality (PESQ) (#353)
  - Short-Time Objective Intelligibility (STOI) (#353)
- Added Information retrieval metrics:
  - RetrievalRPrecision (#577)
  - RetrievalHitRate (#576)
- Added NLP metrics:
  - SacreBLEUScore (#546)
  - CharErrorRate (#575)
- Added other metrics:
  - Tweedie Deviance Score (#499)
  - Learned Perceptual Image Patch Similarity (LPIPS) (#431)
- Added MAP (mean average precision) metric to new detection package (#467)
- Added support for float targets in nDCG metric (#437)
- Added average argument to AveragePrecision metric for reducing multi-label and multi-class problems (#477)
- Added MultioutputWrapper (#510)
- Added metric sweeping:
  - higher\_is\_better as constant attribute (#544)
  - higher\_is\_better to rest of codebase (#584)
- Added simple aggregation metrics: SumMetric, MeanMetric, CatMetric, MinMetric, MaxMetric (#506)
- Added pairwise submodule with metrics (#553)
  - pairwise\_cosine\_similarity
  - pairwise\_euclidean\_distance
  - pairwise\_linear\_similarity
  - pairwise\_manhattan\_distance

### [0.6.0] - Changed

- `AveragePrecision` will now as default output the macro average for multilabel and multiclass problems (#477)
- `half`, `double`, `float` will no longer change the dtype of the metric states. Use `metric.set_dtype` instead (#493)
- Renamed `AverageMeter` to `MeanMetric` (#506)
- Changed `is_differentiable` from property to a constant attribute (#551)
- ROC and AUROC will no longer throw an error when either the positive or negative class is missing. Instead return 0 score and give a warning

### [0.6.0] - Deprecated

- Deprecated `functional.self_supervised.embedding_similarity` in favour of new pairwise submodule

### [0.6.0] - Removed

- Removed `dtype` property (#493)

### [0.6.0] - Fixed

- Fixed bug in F1 with `average='macro'` and `ignore_index!=None` (#495)
- Fixed bug in `pit` by using the returned first result to initialize device and type (#533)
- Fixed SSIM metric using too much memory (#539)
- Fixed bug where device property was not properly update when metric was a child of a module (#542)

## 1.139.25 [0.5.1] - 2021-08-30

### [0.5.1] - Added

- Added device and dtype properties (#462)
- Added `TextTester` class for robustly testing text metrics (#450)

### [0.5.1] - Changed

- Added support for float targets in `nDCG` metric (#437)

### [0.5.1] - Removed

- Removed rouge-score as dependency for text package (#443)
- Removed jiwer as dependency for text package (#446)
- Removed bert-score as dependency for text package (#473)

### [0.5.1] - Fixed

- Fixed ranking of samples in SpearmanCorrCoef metric (#448)
- Fixed bug where compositional metrics where unable to sync because of type mismatch (#454)
- Fixed metric hashing (#478)
- Fixed BootStrapper metrics not working on GPU (#462)
- Fixed the semantic ordering of kernel height and width in SSIM metric (#474)

## 1.139.26 [0.5.0] - 2021-08-09

### [0.5.0] - Added

- Added **Text-related (NLP) metrics**:
  - Word Error Rate (WER) (#383)
  - ROUGE (#399)
  - BERT score (#424)
  - BLUE score (#360)
- Added **MetricTracker** wrapper metric for keeping track of the same metric over multiple epochs (#238)
- Added other metrics:
  - Symmetric Mean Absolute Percentage error (SMAPE) (#375)
  - Calibration error (#394)
  - Permutation Invariant Training (PIT) (#384)
- Added support in nDCG metric for target with values larger than 1 (#349)
- Added support for negative targets in nDCG metric (#378)
- Added None as reduction option in CosineSimilarity metric (#400)
- Allowed passing labels in (n\_samples, n\_classes) to AveragePrecision (#386)

### [0.5.0] - Changed

- Moved `psnr` and `ssim` from `functional.regression.*` to `functional.image.*` (#382)
- Moved `image_gradient` from `functional.image_gradients` to `functional.image.gradients` (#381)
- Moved `R2Score` from `regression.r2score` to `regression.r2` (#371)
- Pearson metric now only store 6 statistics instead of all predictions and targets (#380)
- Use `torch.argmax` instead of `torch.topk` when `k=1` for better performance (#419)
- Moved check for number of samples in `R2` score to support single sample updating (#426)

### [0.5.0] - Deprecated

- Rename `r2score >> r2_score` and `kldivergence >> kl_divergence` in `functional` (#371)
- Moved `bleu_score` from `functional.nlp` to `functional.text.bleu` (#360)

### [0.5.0] - Removed

- Removed restriction that `threshold` has to be in (0,1) range to support logit input ( #351 #401)
- Removed restriction that `preds` could not be bigger than `num_classes` to support logit input (#357)
- Removed module `regression.psnr` and `regression.ssim` (#382):
- Removed (#379):
  - function `functional.mean_relative_error`
  - `num_thresholds` argument in `BinnedPrecisionRecallCurve`

### [0.5.0] - Fixed

- Fixed bug where classification metrics with `average='macro'` would lead to wrong result if a class was missing (#303)
- Fixed `weighted, multi-class` AUROC computation to allow for 0 observations of some class, as contribution to final AUROC is 0 (#376)
- Fixed that `_forward_cache` and `_computed` attributes are also moved to the correct device if metric is moved (#413)
- Fixed calculation in IoU metric when using `ignore_index` argument (#328)

## 1.139.27 [0.4.1] - 2021-07-05

### [0.4.1] - Changed

- Extend typing (#330, #332, #333, #335, #314)

**[0.4.1] - Fixed**

- Fixed DDP by `is_sync` logic to `Metric` (#339)

**1.139.28 [0.4.0] - 2021-06-29****[0.4.0] - Added**

- Added **Image-related metrics**:
  - Fréchet inception distance (FID) (#213)
  - Kernel Inception Distance (KID) (#301)
  - Inception Score (#299)
  - KL divergence (#247)
- Added **Audio metrics**: SNR, SI\_SDR, SI\_SNR (#292)
- Added other metrics:
  - Cosine Similarity (#305)
  - Specificity (#210)
  - Mean Absolute Percentage error (MAPE) (#248)
- Added `add_metrics` method to `MetricCollection` for adding additional metrics after initialization (#221)
- Added pre-gather reduction in the case of `dist_reduce_fx="cat"` to reduce communication cost (#217)
- Added better error message for AUROC when `num_classes` is not provided for multiclass input (#244)
- Added support for unnormalized scores (e.g. logits) in `Accuracy`, `Precision`, `Recall`, `FBeta`, `F1`, `StatScore`, `Hamming`, `ConfusionMatrix` metrics (#200)
- Added `squared` argument to `MeanSquaredError` for computing RMSE (#249)
- Added `is_differentiable` property to `ConfusionMatrix`, `F1`, `FBeta`, `Hamming`, `Hinge`, `IOU`, `MatthewsCorrcoef`, `Precision`, `Recall`, `PrecisionRecallCurve`, `ROC`, `StatScores` (#253)
- Added `sync` and `sync_context` methods for manually controlling when metric states are synced (#302)

**[0.4.0] - Changed**

- Forward cache is reset when `reset` method is called (#260)
- Improved per-class metric handling for imbalanced datasets for `precision`, `recall`, `precision_recall`, `fbeta`, `f1`, `accuracy`, and `specificity` (#204)
- Decorated `torch.jit.unused` to `MetricCollection` forward (#307)
- Renamed `thresholds` argument to binned metrics for manually controlling the thresholds (#322)
- Extend typing (#324, #326, #327)

#### [0.4.0] - Deprecated

- Deprecated `functional.mean_relative_error`, use `functional.mean_absolute_percentage_error` (#248)
- Deprecated `num_thresholds` argument in `BinnedPrecisionRecallCurve` (#322)

#### [0.4.0] - Removed

- Removed argument `is_multiclass` (#319)

#### [0.4.0] - Fixed

- AUC can also support more dimensional inputs when all but one dimension are of size 1 (#242)
- Fixed dtype of modular metrics after reset has been called (#243)
- Fixed calculation in `matthews_corrcoef` to correctly match formula (#321)

### 1.139.29 [0.3.2] - 2021-05-10

#### [0.3.2] - Added

- Added `is_differentiable` property:
  - To AUC, AUROC, CohenKappa and AveragePrecision (#178)
  - To PearsonCorrCoef, SpearmanCorrcoef, R2Score and ExplainedVariance (#225)

#### [0.3.2] - Changed

- `MetricCollection` should return metrics with prefix on `items()`, `keys()` (#209)
- Calling `compute` before `update` will now give warning (#164)

#### [0.3.2] - Removed

- Removed `numpy` as direct dependency (#212)

#### [0.3.2] - Fixed

- Fixed auc calculation and add tests (#197)
- Fixed loading persisted metric states using `load_state_dict()` (#202)
- Fixed PSNR not working with DDP (#214)
- Fixed metric calculation with unequal batch sizes (#220)
- Fixed metric concatenation for list states for zero-dim input (#229)
- Fixed numerical instability in AUROC metric for large input (#230)



### 1.139.30 [0.3.1] - 2021-04-21

- Cleaning remaining inconsistency and fix PL develop integration ( [#191](#), [#192](#), [#193](#), [#194](#) )

### 1.139.31 [0.3.0] - 2021-04-20

#### [0.3.0] - Added

- Added BootStrapper to easily calculate confidence intervals for metrics ([#101](#))
- Added Binned metrics ([#128](#))
- Added metrics for Information Retrieval (([PL^5032](#))):
  - RetrievalMAP ([PL^5032](#))
  - RetrievalMRR ([#119](#))
  - RetrievalPrecision ([#139](#))
  - RetrievalRecall ([#146](#))
  - RetrievalNormalizedDCG ([#160](#))
  - RetrievalFallOut ([#161](#))
- Added other metrics:
  - CohenKappa ([#69](#))
  - MatthewsCorrcoef ([#98](#))
  - PearsonCorrcoef ([#157](#))
  - SpearmanCorrcoef ([#158](#))
  - Hinge ([#120](#))
- Added average='micro' as an option in AUROC for multilabel problems ([#110](#))
- Added multilabel support to ROC metric ([#114](#))
- Added testing for half precision ([#77](#), [#135](#) )
- Added AverageMeter for ad-hoc averages of values ([#138](#))
- Added prefix argument to MetricCollection ([#70](#))
- Added \_\_getitem\_\_ as metric arithmetic operation ([#142](#))
- Added property is\_differentiable to metrics and test for differentiability ([#154](#))
- Added support for average, ignore\_index and mdmc\_average in Accuracy metric ([#166](#))
- Added postfix arg to MetricCollection ([#188](#))

### [0.3.0] - Changed

- Changed ExplainedVariance from storing all preds/targets to tracking 5 statistics (#68)
- Changed behaviour of confusionmatrix for multilabel data to better match multilabel\_confusion\_matrix from sklearn (#134)
- Updated FBeta arguments (#111)
- Changed reset method to use detach.clone() instead of deepcopy when resetting to default (#163)
- Metrics passed as dict to MetricCollection will now always be in deterministic order (#173)
- Allowed MetricCollection pass metrics as arguments (#176)

### [0.3.0] - Deprecated

- Rename argument is\_multiclass -> multiclass (#162)

### [0.3.0] - Removed

- Prune remaining deprecated (#92)

### [0.3.0] - Fixed

- Fixed when \_stable\_1d\_sort to work when  $n \geq N$  (PL#6177)
- Fixed \_computed attribute not being correctly reset (#147)
- Fixed to Blau score (#165)
- Fixed backwards compatibility for logging with older version of pytorch-lightning (#182)

## 1.139.32 [0.2.0] - 2021-03-12

### [0.2.0] - Changed

- Decoupled PL dependency (#13)
- Refactored functional - mimic the module-like structure: classification, regression, etc. (#16)
- Refactored utilities - split to topics/submodules (#14)
- Refactored MetricCollection (#19)

### [0.2.0] - Removed

- Removed deprecated metrics from PL base (#12, #15)

**1.139.33 [0.1.0] - 2021-02-22**

- Added Accuracy metric now generalizes to Top-k accuracy for (multi-dimensional) multi-class inputs using the `top_k` parameter (PL^4838)
- Added Accuracy metric now enables the computation of subset accuracy for multi-label or multi-dimensional multi-class inputs with the `subset_accuracy` parameter (PL^4838)
- Added HammingDistance metric to compute the hamming distance (loss) (PL^4838)
- Added StatScores metric to compute the number of true positives, false positives, true negatives and false negatives (PL^4839)
- Added R2Score metric (PL^5241)
- Added MetricCollection (PL^4318)
- Added `.clone()` method to metrics (PL^4318)
- Added IoU class interface (PL^4704)
- The Recall and Precision metrics (and their functional counterparts `recall` and `precision`) can now be generalized to Recall@K and Precision@K with the use of `top_k` parameter (PL^4842)
- Added compositional metrics (PL^5464)
- Added AUC/AUROC class interface (PL^5479)
- Added QuantizationAwareTraining callback (PL^5706)
- Added ConfusionMatrix class interface (PL^4348)
- Added multiclass AUROC metric (PL^4236)
- Added PrecisionRecallCurve, ROC, AveragePrecision class metric (PL^4549)
- Classification metrics overhaul (PL^4837)
- Added F1 class metric (PL^4656)
- Added metrics aggregation in Horovod and fixed early stopping (PL^3775)
- Added `persistent(mode)` method to metrics, to enable and disable metric states being added to `state_dict` (PL^4482)
- Added unification of regression metrics (PL^4166)
- Added persistent flag to `Metric.add_state` (PL^4195)
- Added classification metrics (PL^4043)
- Added new Metrics API. (PL^3868, PL^3921)
- Added EMB similarity (PL^3349)
- Added SSIM metrics (PL^2671)
- Added BLEU metrics (PL^2535)



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## A

`add_state()` (*torchmetrics.Metric method*), 33

## B

`bert_score()` (in module *torchmetrics.functional.text.bert*), 501

## C

`clone()` (*torchmetrics.Metric method*), 34

`compute()` (*torchmetrics.Metric method*), 34

## D

`device` (*torchmetrics.Metric property*), 37

`double()` (*torchmetrics.Metric method*), 34

## F

`float()` (*torchmetrics.Metric method*), 35

`forward()` (*torchmetrics.Metric method*), 35

## H

`half()` (*torchmetrics.Metric method*), 35

## I

`infoml()` (in module *torchmetrics.functional.text.infoml*), 507

## M

`Metric` (class in *torchmetrics*), 33

## P

`persistent()` (*torchmetrics.Metric method*), 35

`plot()` (*torchmetrics.Metric method*), 35

## R

`reset()` (*torchmetrics.Metric method*), 35

## S

`set_dtype()` (*torchmetrics.Metric method*), 35

`spectral_distortion_index()` (in module *torchmetrics.functional*), 414

`state_dict()` (*torchmetrics.Metric method*), 36

`sync()` (*torchmetrics.Metric method*), 36

`sync_context()` (*torchmetrics.Metric method*), 36

## T

`type()` (*torchmetrics.Metric method*), 37

## U

`unsync()` (*torchmetrics.Metric method*), 37

`update()` (*torchmetrics.Metric method*), 37

`update_called` (*torchmetrics.Metric property*), 37

`update_count` (*torchmetrics.Metric property*), 37