MMEval

Release 0.2.1

MMEval Contributors

Apr 03, 2023

GET STARTED

CHAPTER

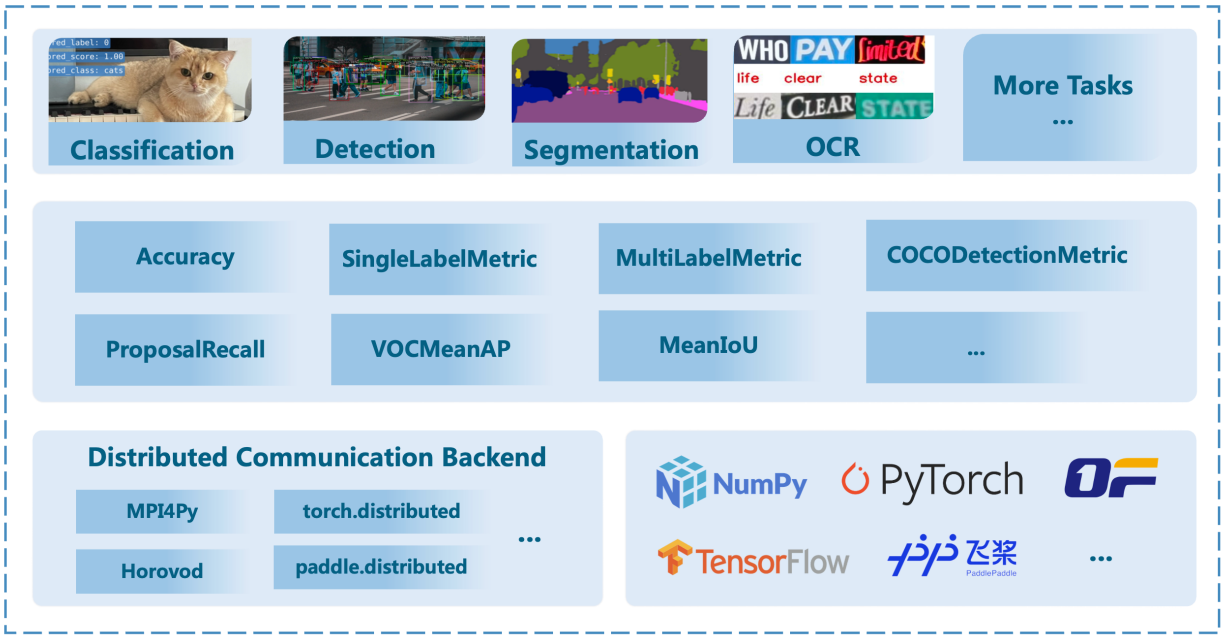
ONE

INTRODUCTION

MMEval is a machine learning evaluation library that supports efficient and accurate distributed evaluation on a variety of machine learning frameworks.

Major features:

- Comprehensive metrics for various computer vision tasks (NLP will be covered soon!)
- Efficient and accurate distributed evaluation, backed by multiple distributed communication backends
- Support multiple machine learning frameworks via dynamic input dispatching mechanism



CHAPTER

TWO

INSTALLATION AND USAGE

2.1 Installation

`MMEval` requires Python 3.6+ and can be installed via pip.

```
pip install mmeval
```

To install the dependencies required for all the metrics provided in `MMEval`, you can install them with the following command.

```
pip install 'mmeval[all]'
```

2.2 How to use

There are two ways to use `MMEval`'s metrics, using *Accuracy* as an example:

```
from mmeval import Accuracy
import numpy as np

accuracy = Accuracy()
```

The first way is to directly call the instantiated `Accuracy` object to calculate the metric.

```
labels = np.asarray([0, 1, 2, 3])
preds = np.asarray([0, 2, 1, 3])
accuracy(preds, labels)
# {'top1': 0.5}
```

The second way is to calculate the metric after accumulating data from multiple batches.

```
for i in range(10):
    labels = np.random.randint(0, 4, size=(100, ))
    predicts = np.random.randint(0, 4, size=(100, ))
    accuracy.add(predicts, labels)

accuracy.compute()
# {'top1': ...}
```

CHAPTER

THREE

SUPPORT MATRIX

3.1 Supported distributed communication backends

3.2 Supported metrics and ML frameworks

Note: The following table lists the metrics implemented by MMEval and the corresponding machine learning framework support. A check mark indicates that the data type of the corresponding framework (e.g. Tensor) can be directly passed for computation.

Note: MMEval tested with PyTorch 1.6+, TensorFlow 2.4+, Paddle 2.2+ and OneFlow 0.8+.

CHAPTER

FOUR

IMPLEMENTING A METRIC

To implement a metric in `MMEval`, you should implement a subclass of *BaseMetric* that overrides the `add` and `compute_metric` methods.

In the evaluation process, each metric will update `self._results` to store intermediate results after each call of `add`. When computing the final metric result, the `self._results` will be synchronized between processes.

An example that implementing simple `Accuracy` metric:

```python
import numpy as np
from mmeval.core import BaseMetric

class Accuracy(BaseMetric):

    def add(self, predictions, labels):
        self._results.append((predictions, labels))

    def compute_metric(self, results):
        predictions = np.concatenate(
            [res[0] for res in results])
        labels = np.concatenate(
            [res[1] for res in results])
        correct = (predictions == labels)
        accuracy = sum(correct) / len(predictions)
        return {'accuracy': accuracy}
```

Use `Accuracy`

```python
# stateless call
accuracy = Accuracy()
metric_results = accuracy(predictions=[1, 2, 3, 4], labels=[1, 2, 3, 1])
print(metric_results)
# {'accuracy': 0.75}

# Accumulate batch
for i in range(10):
    predicts = np.random.randint(0, 4, size=(10,))
    labels = predicts = np.random.randint(0, 4, size=(10,))
    accuracy.add(predicts, labels)

metric_results = accuracy.compute()
accuracy.reset()  # clear the intermediate results
```

CHAPTER

FIVE

USING DISTRIBUTED EVALUATION

Distributed evaluation generally uses a strategy of data parallelism, where each process executes the same program to process different data.

The supported distributed communication backends in MMEval can be viewed via *list_all_backends*.

```
import mmeval

print(mmeval.core.dist.list_all_backends())
# ['non_dist', 'mpi4py', 'tf_horovod', 'torch_cpu', 'torch_cuda', ...]
```

This section shows how to use MMEval in the combination of `torch.distributed` and `MPI4Py` for distributed evaluation, using the CIFAR-10 dataset as an example. The related code can be found at mmeval/examples/cifar10_dist_eval.

5.1 Prepare the evaluation dataset and model

First of all, we need to load the CIFAR-10 test data, we can use the dataset classes provided by `Torchvison`.

In addition, to be able to slice the dataset according to the number of processes in a distributed evaluation, we need to introduce the `DistributedSampler`.

```
import torchvision as tv
from torch.utils.data import DataLoader, DistributedSampler

def get_eval_dataloader(rank=0, num_replicas=1):
    dataset = tv.datasets.CIFAR10(
        root='./', train=False, download=True,
        transform=tv.transforms.ToTensor())
    dist_sampler = DistributedSampler(
        dataset, num_replicas=num_replicas, rank=rank)
    data_loader = DataLoader(dataset, batch_size=1, sampler=dist_sampler)
    return data_loader, len(dataset)
```

Secondly, we need to prepare the model to be evaluated, here we use `resnet18` from `Torchvision`.

```
import torch
import torchvision as tv

def get_model(pretrained_model_fpath=None):
    model = tv.models.resnet18(num_classes=10)
    if pretrained_model_fpath is not None:
```

(continues on next page)

(continued from previous page)

```
        model.load_state_dict(torch.load(pretrained_model_fpath))
    return model.eval()
```

5.2 Single process evaluation

After preparing the test data and the model, the model predictions can be evaluated using the *mmeval.Accuracy* metric. The following is an example of a single process evaluation.

```
import tqdm
import torch
from mmeval import Accuracy

eval_dataloader, total_num_samples = get_eval_dataloader()
model = get_model()
# Instantiate `Accuracy` and calculate the top1 and top3 accuracy
accuracy = Accuracy(topk=(1, 3))

with torch.no_grad():
    for images, labels in tqdm.tqdm(eval_dataloader):
        predicted_score = model(images)
        # Accumulate batch data, intermediate results will be saved in
        # `accuracy._results`.
        accuracy.add(predictions=predicted_score, labels=labels)

# Invoke `accuracy.compute` for metric calculation
print(accuracy.compute())
# Invoke `accuracy.reset` to clear the intermediate results saved in
# `accuracy._results`
accuracy.reset()
```

5.3 Distributed evaluation with torch.distributed

There are two distributed communication backends implemented in `MMEval` for `torch.distributed`, *TorchCPUDist* and *TorchCUDADist*.

There are 2 ways to set up a distributed communication backend for `MMEval`:

```
from mmeval.core import set_default_dist_backend
from mmeval import Accuracy

# 1. Set the global default distributed communication backend.
set_default_dist_backend('torch_cpu')

# 2. Initialize the evaluation metrics by passing `dist_backend`.
accuracy = Accuracy(dist_backend='torch_cpu')
```

Together with the above code for single process evaluation, the distributed evaluation can be implemented by adding the distributed environment startup and initialization.

```
import tqdm
import torch
from mmeval import Accuracy


def eval_fn(rank, process_num):
    # Distributed environment initialization
    torch.distributed.init_process_group(
        backend='gloo',
        init_method=f'tcp://127.0.0.1:2345',
        world_size=process_num,
        rank=rank)

    eval_dataloader, total_num_samples = get_eval_dataloader(rank, process_num)
    model = get_model()
    # Instantiate `Accuracy` and set up a distributed communication backend
    accuracy = Accuracy(topk=(1, 3), dist_backend='torch_cpu')

    with torch.no_grad():
        for images, labels in tqdm.tqdm(eval_dataloader, disable=(rank!=0)):
            predicted_score = model(images)
            accuracy.add(predictions=predicted_score, labels=labels)

    # Specify the number of dataset samples by size in order to remove
    # duplicate samples padded by the `DistributedSampler`.
    print(accuracy.compute(size=total_num_samples))
    accuracy.reset()


if __name__ == "__main__":
    # Number of distributed processes
    process_num = 3
    # Launching distributed with spawn
    torch.multiprocessing.spawn(
        eval_fn, nprocs=process_num, args=(process_num, ))
```

5.4 Distributed evaluation with MPI4Py

`MMEval` has decoupled the distributed communication capability. While the above example uses the `PyTorch` model and data loading, we can still use distributed communication backends other than `torch.distributed` to implement distributed evaluation.

The following will show how to use `MPI4Py` as a distributed communication backend for distributed evaluation.

First, you need to install `MPI4Py` and `openmpi`, it is recommended to use `conda` to install.

```
conda install openmpi
conda install mpi4py
```

Then modify the above code to use `MPI4Py` as the distributed communication backend:

```python
# cifar10_eval_mpi4py.py

import tqdm
from mpi4py import MPI
import torch
from mmeval import Accuracy


def eval_fn(rank, process_num):
    eval_dataloader, total_num_samples = get_eval_dataloader(rank, process_num)
    model = get_model()
    accuracy = Accuracy(topk=(1, 3), dist_backend='mpi4py')

    with torch.no_grad():
        for images, labels in tqdm.tqdm(eval_dataloader, disable=(rank!=0)):
            predicted_score = model(images)
            accuracy.add(predictions=predicted_score, labels=labels)

    print(accuracy.compute(size=total_num_samples))
    accuracy.reset()


if __name__ == "__main__":
    comm = MPI.COMM_WORLD
    eval_fn(comm.Get_rank(), comm.Get_size())
```

Using `mpirun` as the distributed launch method.

```
# Launch 3 processes with mpirun
mpirun -np 3 python3 cifar10_eval_mpi4py.py
```

CHAPTER

SIX

MMCLS

BaseMetric in `MMEval` follows the design of the mmengine.evaluator module and introduces distributed communication backend to meet the needs of a diverse distributed communication library.

Therefore, `MMEval` naturally supports the evaluation based on OpenMMLab 2.0 algorithm library, and the evaluation metrics using MMEval in OpenMMLab 2.0 algorithm library need not be modified.

For example, use *mmeval.Accuracy* in MMCls, just configure the Metric to be Accuracy in the config:

```
val_evaluator = dict(type='Accuracy', topk=(1, ))

test_evaluator = val_evaluator
```

MMEval's support for OpenMMLab 2.0 algorithm library is being gradually improved, and the supported metric can be viewed in the *support matrix*.

CHAPTER

SEVEN

TENSORPACK

TensorPack is a neural net training interface on TensorFlow, with focus on speed + flexibility

There are many examples of classic models and tasks provided in the TensorPack repository. This section shows how to use *mmeval.COCODetection* for evaluation in TensorPack-FasterRCNN, and the related code can be found at mmeval/examples/tensorpack.

First you need to install `TensorFlow` and `TensorPack`, then follow the preparation steps in the TensorPack-FasterRCNN example to install the dependencies and prepare the COCO dataset, as well as download the pre-trained model weights to be evaluated.

Scripts for model evaluation are provided in predict.py, and the model can be evaluated with the following commands:

```
./predict.py --evaluate output.json --load /path/to Trained-Model-Checkpoint --config␣
↪SAME-AS-TRAINING
```

`MMEval` provides a evaluation tools for `TensorPack-FasterRCNN` that use *mmeval.COCODetection*. This evaluation script needs to be placed in the `TensorPack-FasterRCNN` example directory, and then the evaluation can be executed with the following command.

```
# run evaluation
python tensorpack_mmeval.py --load <model_path>

# launch multi-gpus evaluation by mpirun
mpirun -np 8 python tensorpack_mmeval.py --load <model_path>
```

We tested this evaluation script on COCO-MaskRCNN-R50C41x and got the same evaluation results as the TensorPack report.

CHAPTER

EIGHT

PADDLESEG

PaddleSeg is a semantic segmentation algorithm library based on Paddle that supports many downstream tasks related to semantic segmentation.

This section shows how to use *mmeval.MeanIoU* for evaluation in PaddleSeg, and the related code can be found at mmeval/examples/paddleseg.

First you need to install `Paddle` and `PaddleSeg`, you can refer to the installation documentation in `PaddleSeg`. In addition, you need to download the pre-trained model to be evaluated, and prepare the evaluation data according to the configuration.

Scripts for model evaluation are provided in the `PaddleSeg` repo, and the model can be evaluated with the following commands:

```
python val.py --config <config_path> --model_path <model_path>
```

Note that the `val.py` script in the `PaddleSeg` only supports single-GPU evaluation, not multi-GPU evaluation yet.

`MMEval` provides a evaluation tools for `PaddleSeg` that use *mmeval.MeanIoU*, which can be executed with the following command:

```
# run evaluation
python ppseg_mmeval.py --config <config_path> --model_path <model_path>

# run evaluation with multi-gpus
python ppseg_mmeval.py --config <config_path> --model_path <model_path> --launcher␣
↪paddle --num_process <num_gpus>
```

We tested this evaluation script on fastfcn_resnet50_os8_ade20k_480x480_120k and got the same evaluation results as the val.py in PaddleSeg.

CHAPTER

NINE

BASEMETRIC DESIGN

During the evaluation process, the results of partial datasets are usually inferred on each GPU in data parallel to speed up the evaluation.

Most of the time, we can't just reduce the metric results from each subset of the dataset as the metric result of the dataset.

Therefore, the usual practice is to save the inference results obtained by each process or the intermediate results of the metric calculation. Then perform an all-gather operation across all processes, and finally calculate the metric results of the entire evaluation dataset.

The above operations are completed by *BaseMetric* in `MMEval`, and its interface design is shown in the following:

The `add` and `compute_metric` methods are interfaces that need to be implemented by users. For more details, please refer to *Custom Evaluation Metrics*.

It can be seen from the `[BaseMetric](mmeval.core.BaseMetric) interface that the main function of` *BaseMetric* is to provide distributed evaluation. The basic process is as follows:

1. The user calls the `add` method to save the inference result or the intermediate result of the metric calculation in the `BaseMetric._results` list.
2. The user calls the `compute` method, and `BaseMetric` synchronizes the data in the `_results` list across processes and calls the user-defined `compute_metric` method to calculate the metrics.

In addition, *BaseMetric* also considers that in distributed evaluation, some processes may pad repeated data samples, in order to ensure the same number of data samples in all processes. Such behavior will affect the indicators correctness of the calculation. E.g. `DistributedSampler` in PyTorch.

To deal with this problem, *BaseMetric.compute* can receive a `size` parameter, which represents the actual number of samples in the evaluation dataset. After `_results` completes process synchronization, the padded samples will be removed according to `dist_collect_mode` to achieve correct metric calculation.

Note: Be aware that the intermediate results stored in `_results` should correspond one-to-one with the samples, in that we need to remove the padded samples for the most accurate result.

CHAPTER

TEN

DISTRIBUTED COMMUNICATION BACKEND

The distributed communication requirements required by `MMEval` in the distributed evaluation mainly include the following:

- All-gather the intermediate results of the metric saved in each process.
- Broadcast the metric result calculated by the rank 0 process to all processes

In order to flexibly support multiple distributed communication libraries, MMEval abstracts the above distributed communication requirements and defines a distributed communication interface *BaseDistBackend*:

To implement a distributed communication backend, you need to inherit *BaseDistBackend* and implement the above interfaces, where:

- is_initialized: identifies whether the initialization of the distributed communication environment has been completed.
- rank: the rank index of the current process group.
- world_size: the world size of the current process group.
- all_gather_object: perform the all_tather operation on any Python object that can be serialized by `Pickle`.
- broadcast_object: broadcasts any Python object that can be serialized by `Pickle`.

Take the implementation of *MPI4PyDist* as an example:

```python
from mpi4py import MPI


class MPI4PyDist(BaseDistBackend):
    """A distributed communication backend for mpi4py."""

    @property
    def is_initialized(self) -> bool:
        """Returns True if the distributed environment has been initialized."""
        return 'OMPI_COMM_WORLD_SIZE' in os.environ

    @property
    def rank(self) -> int:
        """Returns the rank index of the current process group."""
        comm = MPI.COMM_WORLD
        return comm.Get_rank()

    @property
    def world_size(self) -> int:
```

(continues on next page)

(continued from previous page)

```
        """Returns the world size of the current process group."""
        comm = MPI.COMM_WORLD
        return comm.Get_size()

    def all_gather_object(self, obj: Any) -> List[Any]:
        """All gather the given object from the current process group and
        returns a list consisting gathered object of each process."""
        comm = MPI.COMM_WORLD
        return comm.allgather(obj)

    def broadcast_object(self, obj: Any, src: int = 0) -> Any:
        """Broadcast the given object from source process to the current
        process group."""
        comm = MPI.COMM_WORLD
        return comm.bcast(obj, root=src)
```

Some distributed communication backends have been implemented in `MMEval`, which can be viewed in the *support matrix*.

CHAPTER

ELEVEN

MULTIPLE DISPATCH

MMEval wants to support multiple machine learning frameworks. One of the simplest solutions is to have NumPy support for the computation of all metrics.

Since all machine learning frameworks have Tensor data types that can be converted to numpy.ndarray, this can satisfy most of the evaluation requirements.

However, there may be some problems in some cases:

- NumPy has some common operators that have not been implemented yet, such as topk, which can affect the computational speed of the evaluation metrics.
- It is time-consuming to move a large number of Tensors from CUDA devices to CPU memory.

Alternatively, if it is desired that the computation of the metrics for the rubric be differentiable, then the Tensor data type of the respective machine learning framework needs to be used for the computation.

To deal with the above, MMEval's evaluation metrics provide some implementations of metrics computed with specific machine learning frameworks, which can be found in [support_matrix](.. /get_started/support_matrix.md).

Meanwhile, in order to deal with the dispatch problem of different metrics calculation methods, MMEval adopts a dynamic multi-distribution mechanism based on type hints, which can dynamically select corresponding calculation methods according to the input data types.

A simple example of multiple dispatch based on type hints is as below:

```
from mmeval.core import dispatch

@dispatch
def compute(x: int, y: int):
    print('this is int')

@dispatch
def compute(x: str, y: str):
    print('this is str')

compute(1, 1)
# this is int

compute('1', '1')
# this is str
```

Currently, we use plum-dispatch to implement multiple dispatch mechanism in `MMEval`. Based on plum-dispatch, some speed optimizations have been made and extended to support `typing.ForwardRef`.

Warning: Due to the dynamically typed feature of Python, determining the exact type of a variable at runtime can be time-consuming, especially when you encounter large nested structures of data. Therefore, the dynamic multi-dispatch mechanism based on type hints may have some performance problems, for more information see atwesselb/plum/issues/53

CHAPTER

TWELVE

MMEVAL.CORE

mmeval.core

12.1 base_metric

BaseMetric	Base class for metric.

12.1.1 BaseMetric

class mmeval.core.**BaseMetric**(*dataset_meta: Optional[Dict] = None*, *dist_collect_mode: str = 'unzip'*, *dist_backend: Optional[str] = None*, *logger: Optional[logging.Logger] = None*)

Base class for metric.

To implement a metric, you should implement a subclass of `BaseMetric` that overrides the `add` and `compute_metric` methods. `BaseMetric` will automatically complete the distributed synchronization between processes.

In the evaluation process, each metric will update `self._results` to store intermediate results after each call of `add`. When computing the final metric result, the `self._results` will be synchronized between processes.

Parameters

- **dataset_meta** (*dict, optional*) – Meta information of the dataset, this is required for some metrics that require dataset information. Defaults to None.
- **dist_collect_mode** (*str, optional*) – The method of concatenating the collected synchronization results. This depends on how the distributed data is split. Currently only 'unzip' and 'cat' are supported. For PyTorch's `DistributedSampler`, 'unzip' should be used. Defaults to 'unzip'.
- **dist_backend** (*str, optional*) – The name of the distributed communication backend, you can get all the backend names through `mmeval.core.list_all_backends()`. If `None`, use the default backend. Defaults to None.

- **logger** (*Logger, optional*) – The logger used to log messages. If None, use the default logger of mmeval. Defaults to None.

Example to implement an accuracy metric:

```
>>> import numpy as np
>>> from mmeval.core import BaseMetric
>>>
>>> class Accuracy(BaseMetric):
...     def add(self, predictions, labels):
...         self._results.append((predictions, labels))
...     def compute_metric(self, results):
...         predictions = np.concatenate([res[0] for res in results])
...         labels = np.concatenate([res[1] for res in results])
...         correct = (predictions == labels)
...         accuracy = sum(correct) / len(predictions)
...         return {'accuracy': accuracy}
```

Stateless call of metric:

```
>>> accuracy = Accuracy()
>>> accuracy(predictions=[1, 2, 3, 4], labels=[1, 2, 3, 1])
{'accuracy': 0.75}
```

Accumulate batch:

```
>>> for i in range(10):
>>>     predicts = np.random.randint(0, 4, size=(10,))
>>>     labels = predicts = np.random.randint(0, 4, size=(10,))
>>>     accuracy.add(predicts, labels)
>>> accuracy.compute()
```

abstract add(**args*, ***kwargs*)

Override this method to add the intermediate results to `self._results`.

Note: For performance issues, what you add to the `self._results` should be as simple as possible. But be aware that the intermediate results stored in `self._results` should correspond one-to-one with the samples, in that we need to remove the padded samples for the most accurate result.

---

compute(*size: Optional[int] = None*) → Dict

Synchronize intermediate results and then call `self.compute_metric`.

Parameters **size** (*int, optional*) – The length of the entire dataset, it is only used when distributed evaluation. When batch size > 1, the dataloader may pad some data samples to make sure all ranks have the same length of dataset slice. The `compute` will drop the padded data based on this size. If None, do nothing. Defaults to None.

Returns The computed metric results.

Return type dict

abstract compute_metric(*results: List[Any]*) → Dict

Override this method to compute the metric result from collectd intermediate results.

The returned result of the metric compute should be a dictionary.

property dataset_meta: Optional[Dict]
Meta information of the dataset.

property name: str
The metric name, defaults to the name of the class.

reset() → None
Clear the metric stored results.

12.2 dist

list_all_backends	Returns a list of all distributed backend names.
set_default_dist_backend	Set the given distributed backend as the default distributed backend.
get_dist_backend	Returns distributed backend by the given distributed backend name.

12.2.1 mmeval.core.list_all_backends

mmeval.core.**list_all_backends**() → List[str]
Returns a list of all distributed backend names.

Returns A list of all distributed backend names.

Return type List[str]

12.2.2 mmeval.core.set_default_dist_backend

mmeval.core.**set_default_dist_backend**(*dist_backend: str*) → None
Set the given distributed backend as the default distributed backend.

Parameters **dist_backend** (*str*) – The distribute backend name to set.

12.2.3 mmeval.core.get_dist_backend

mmeval.core.**get_dist_backend**(*dist_backend: Optional[str] = None*) → *mmeval.core.dist_backends.base_backend.BaseDistBackend*
Returns distributed backend by the given distributed backend name.

Parameters **dist_backend** (*str, optional*) – The distributed backend name want to get. if None, return the default distributed backend.

Returns The distributed backend instance.

Return type BaseDistBackend

12.3 dispatch

dispatch	A Dispatcher inherited from `plum.Dispatcher` that resolve `typing.ForwardRef`.

12.3.1 mmeval.core.dispatch

mmeval.core.**dispatch = <mmeval.core.dispatcher._MMEvalDispatcher object>**

A Dispatcher inherited from `plum.Dispatcher` that resolve `typing.ForwardRef`.

This dispatcher tries to use `importlib.import_moudle` to import ForwardRerf type and convert unimportable type as a placeholder.

With the `_MMEvalDispatcher`, we can run the following code example without PyTorch installed, which is `plum.dispatch` can't do.

Example

```
>>> from mmeval.core import dispatch
```

```
>>> @dispatch
>>> def compute(x: 'torch.Tensor'):
...     print('The input is a `torch.Tensor`')
```

```
>>> @dispatch
>>> def compute(x: 'numpy.ndarray'):
...     print('The input is a `numpy.ndarray`')
```

CHAPTER

THIRTEEN

MMEVAL.CORE.DIST_BACKENDS

mmeval.core.dist_backends

- *dist_backends*

13.1 dist_backends

BaseDistBackend	The base backend of distributed communication used by mmeval Metric.
TensorBaseDistBackend	A base backend of Tensor base distributed communication like PyTorch.
NonDist	A dummy distributed communication for non-distributed environment.
MPI4PyDist	A distributed communication backend for mpi4py.
TorchCPUDist	A cpu distributed communication backend for torch.distributed.
TorchCUDADist	A cuda distributed communication backend for torch.distributed.
TFHorovodDist	A distributed communication backend for horovod.tensorflow.
PaddleDist	A distributed communication backend for paddle.distributed.
OneFlowDist	A distributed communication backend for oneflow.

13.1.1 BaseDistBackend

class mmeval.core.dist_backends.**BaseDistBackend**

The base backend of distributed communication used by mmeval Metric.

abstract all_gather_object(*obj: Any*) → List[Any]

All gather the given object from the current process group and returns a list consisting gathered object of each process..

Parameters **obj** (*any*) – Any pickle-able python object for all gather.

Returns A list of the all gathered object.

Return type list

abstract broadcast_object(*obj: Any*, *src: int*) → Any
Broadcast the given object from source process to the current process group.

Parameters

- **obj** (*any*) – Any pickle-able python object for broadcast.
- **src** (*int*) – The source rank index.

Returns The broadcast object.

Return type any

abstract property is_initialized: bool
Returns True if the distributed environment has been initialized.

Returns Returns True if the distributed environment has been initialized, otherwise returns False.

Return type bool

abstract property rank: int
Returns the rank index of the current process group.

Returns The rank index of the current process group.

Return type int

abstract property world_size: int
Returns the world size of the current process group.

The *world size* is the size of the communication process group.

Returns The size of the current process group.

Return type int

13.1.2 TensorBaseDistBackend

class mmeval.core.dist_backends.**TensorBaseDistBackend**
A base backend of Tensor base distributed communication like PyTorch.

all_gather_object(*obj: Any*) → List[Any]
All gather the given object from the current process group and returns a list consisting gathered object of each process..

There are 3 steps to all gather a python object using Tensor distributed communication:

1. Serialize picklable python object to tensor.
2. All gather the tensor size and padding the tensor with the same size.
3. All gather the padded tensor and deserialize tensor to picklable python object.

Parameters **obj** (*any*) – Any pickle-able python object for all gather.

Returns A list of the all gathered object.

Return type list

broadcast_object(*obj: Any*, *src: int = 0*) → Any
Broadcast the given object from source process to the current process group.

There are 3 steps to broadcast a python object use Tensor distributed communication:

1. Serialize picklable python object to tensor.

2. Broadcast the tensor size and padding the tensor with the same size.
3. Broadcast the padded tensor and deserialize tensor to picklable python object.

Parameters

- **obj** (*any*) – Any pickle-able python object for broadcast.
- **src** (*int*) – The source rank index.

Returns The broadcast object.

Return type any

13.1.3 NonDist

class mmeval.core.dist_backends.**NonDist**

A dummy distributed communication for non-distributed environment.

all_gather_object(*obj: Any*) → List[Any]

Returns the list with given obj in a non-distributed environment.

broadcast_object(*obj: Any*, *src: int = 0*) → Any

Returns the given obj directly in a non-distributed environment.

property is_initialized: bool

Returns False directly in a non-distributed environment.

property rank: int

Returns 0 as the rank index in a non-distributed environment.

property world_size: int

Returns 1 as the world_size in a non-distributed environment.

13.1.4 MPI4PyDist

class mmeval.core.dist_backends.**MPI4PyDist**

A distributed communication backend for mpi4py.

all_gather_object(*obj: Any*) → List[Any]

All gather the given object from the current process group and returns a list consisting gathered object of each process.

Parameters **obj** (*any*) – Any pickle-able python object for all gather.

Returns A list of the all gathered object.

Return type list

broadcast_object(*obj: Any*, *src: int = 0*) → Any

Broadcast the given object from source process to the current process group.

Parameters

- **obj** (*any*) – Any pickle-able python object for broadcast.
- **src** (*int*) – The source rank index.

Returns The broadcast object.

Return type any

property is_initialized: bool
Returns True if the distributed environment has been initialized.

Returns Returns True if the distributed environment has been initialized, otherwise returns False.

Return type bool

property rank: int
Returns the rank index of the current process group.

property world_size: int
Returns the world size of the current process group.

13.1.5 TorchCPUDist

class mmeval.core.dist_backends.TorchCPUDist
A cpu distributed communication backend for torch.distributed.

property is_initialized: bool
Returns True if the distributed environment has been initialized.

Returns Returns True if the distributed environment has been initialized, otherwise returns False.

Return type bool

property rank: int
Returns the rank index of the current process group.

property world_size: int
Returns the world size of the current process group.

13.1.6 TorchCUDADist

class mmeval.core.dist_backends.TorchCUDADist
A cuda distributed communication backend for torch.distributed.

13.1.7 TFHorovodDist

class mmeval.core.dist_backends.TFHorovodDist
A distributed communication backend for horovod.tensorflow.

all_gather_object(*obj: Any*) → List[Any]
All gather the given object from the current process group and returns a list consisting gathered object of each process..

Parameters **obj** (*any*) – Any pickle-able python object for all gather.

Returns A list of the all gathered object.

Return type list

broadcast_object(*obj: Any*, *src: int = 0*) → Any
Broadcast the given object from source process to the current process group.

Parameters

- **obj** (*any*) – Any pickle-able python object for broadcast.
- **src** (*int*) – The source rank index.

Returns The broadcast object.

Return type any

property is_initialized: bool
Returns True if the distributed environment has been initialized.

Returns Returns True if the distributed environment has been initialized, otherwise returns False.

Return type bool

property rank: int
Returns the rank index of the current process group.

property world_size: int
Returns the world size of the current process group.

13.1.8 PaddleDist

class mmeval.core.dist_backends.**PaddleDist**
A distributed communication backend for paddle.distributed.

property is_initialized: bool
Returns True if the distributed environment has been initialized.

Returns Returns True if the distributed environment has been initialized, otherwise returns False.

Return type bool

property rank: int
Returns the rank index of the current process group.

property world_size: int
Returns the world size of the current process group.

13.1.9 OneFlowDist

class mmeval.core.dist_backends.**OneFlowDist**
A distributed communication backend for oneflow.

property is_initialized: bool
Returns True if the distributed environment has been initialized.

Returns Returns True if the distributed environment has been initialized, otherwise returns False.

Return type bool

property rank: int
Returns the rank index of the current process group.

property world_size: int
Returns the world size of the current process group.

CHAPTER

FOURTEEN

MMEVAL.FILEIO

mmeval.fileio

14.1 File Backend

BaseStorageBackend	Abstract class of storage backends.
LocalBackend	Raw local storage backend.
HTTPBackend	HTTP and HTTPS storage bachend.
LmdbBackend	Lmdb storage backend.
MemcachedBackend	Memcached storage backend.
PetrelBackend	Petrel storage backend (for internal usage).

14.1.1 BaseStorageBackend

class mmeval.fileio.**BaseStorageBackend**

Abstract class of storage backends.

All backends need to implement two apis: *get()* and *get_text()*.

- *get()* reads the file as a byte stream.
- *get_text()* reads the file as texts.

14.1.2 LocalBackend

class mmeval.fileio.**LocalBackend**

Raw local storage backend.

exists(*filepath: Union[str, pathlib.Path]*) → bool

Check whether a file path exists.

Parameters **filepath** (*str or Path*) – Path to be checked whether exists.

Returns Return `True` if `filepath` exists, `False` otherwise.

Return type bool

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.exists(filepath)
True
```

get(*filepath: Union[str, pathlib.Path]*) → bytes

Read bytes from a given `filepath` with 'rb' mode.

Parameters **filepath** (*str or Path*) – Path to read data.

Returns Expected bytes object.

Return type bytes

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.get(filepath)
b'hello world'
```

get_local_path(*filepath: Union[str, pathlib.Path]*) → Generator[Union[str, pathlib.Path], None, None]

Only for unified API and do nothing.

Parameters

- **filepath** (*str or Path*) – Path to be read data.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Examples

```
>>> backend = LocalBackend()
>>> with backend.get_local_path('s3://bucket/abc.jpg') as path:
...     # do something here
```

get_text(*filepath: Union[str, pathlib.Path]*, *encoding: str = 'utf-8'*) → str
Read text from a given `filepath` with 'r' mode.

Parameters

- **filepath** (*str or Path*) – Path to read data.
- **encoding** (*str*) – The encoding format used to open the `filepath`. Defaults to 'utf-8'.

Returns Expected text reading from `filepath`.

Return type str

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.get_text(filepath)
'hello world'
```

isdir(*filepath: Union[str, pathlib.Path]*) → bool
Check whether a file path is a directory.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a directory.

Returns Return `True` if `filepath` points to a directory, `False` otherwise.

Return type bool

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/dir'
>>> backend.isdir(filepath)
True
```

isfile(*filepath: Union[str, pathlib.Path]*) → bool
Check whether a file path is a file.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a file.

Returns Return `True` if `filepath` points to a file, `False` otherwise.

Return type bool

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.isfile(filepath)
True
```

join_path(*filepath: Union[str, pathlib.Path]*, **filepaths: Union[str, pathlib.Path]*) → str
Concatenate all file paths.

Join one or more filepath components intelligently. The return value is the concatenation of filepath and any members of *filepaths.

Parameters **filepath** (*str or Path*) – Path to be concatenated.

Returns The result of concatenation.

Return type str

Examples

```
>>> backend = LocalBackend()
>>> filepath1 = '/path/of/dir1'
>>> filepath2 = 'dir2'
>>> filepath3 = 'path/of/file'
>>> backend.join_path(filepath1, filepath2, filepath3)
'/path/of/dir/dir2/path/of/file'
```

list_dir_or_file(*dir_path: Union[str, pathlib.Path]*, *list_dir: bool = True*, *list_file: bool = True*, *suffix: Optional[Union[str, Tuple[str]]] = None*, *recursive: bool = False*) → Iterator[str]
Scan a directory to find the interested directories or files in arbitrary order.

Note: *list_dir_or_file()* returns the path relative to `dir_path`.

Parameters

- **dir_path** (*str or Path*) – Path of the directory.
- **list_dir** (*bool*) – List the directories. Defaults to True.
- **list_file** (*bool*) – List the path of files. Defaults to True.
- **suffix** (*str or tuple[str], optional*) – File suffix that we are interested in. Defaults to None.
- **recursive** (*bool*) – If set to True, recursively scan the directory. Defaults to False.

Yields *Iterable[str]* – A relative path to `dir_path`.

Examples

```
>>> backend = LocalBackend()
>>> dir_path = '/path/of/dir'
>>> # list those files and directories in current directory
>>> for file_path in backend.list_dir_or_file(dir_path):
...     print(file_path)
>>> # only list files
>>> for file_path in backend.list_dir_or_file(dir_path, list_dir=False):
...     print(file_path)
>>> # only list directories
>>> for file_path in backend.list_dir_or_file(dir_path, list_file=False):
...     print(file_path)
>>> # only list files ending with specified suffixes
>>> for file_path in backend.list_dir_or_file(dir_path, suffix='.txt'):
...     print(file_path)
>>> # list all files and directory recursively
>>> for file_path in backend.list_dir_or_file(dir_path, recursive=True):
...     print(file_path)
```

14.1.3 HTTPBackend

class mmeval.fileio.**HTTPBackend**

HTTP and HTTPS storage bachend.

get(*filepath: str*) → bytes

Read bytes from a given `filepath`.

Parameters **filepath** (*str*) – Path to read data.

Returns Expected bytes object.

Return type bytes

Examples

```
>>> backend = HTTPBackend()
>>> backend.get('http://path/of/file')
b'hello world'
```

get_local_path(*filepath: str*) → Generator[Union[str, pathlib.Path], None, None]

Download a file from `filepath` to a local temporary directory, and return the temporary path.

`get_local_path` is decorated by `contxtlib.contextmanager()`. It can be called with `with` statement, and when exists from the `with` statement, the temporary path will be released.

Parameters **filepath** (*str*) – Download a file from `filepath`.

Yields *Iterable[str]* – Only yield one temporary path.

Examples

```
>>> backend = HTTPBackend()
>>> # After existing from the ``with`` clause,
>>> # the path will be removed
>>> with backend.get_local_path('http://path/of/file') as path:
...     # do something here
```

get_text(*filepath*, *encoding='utf-8'*) → str
Read text from a given `filepath`.

Parameters

- **filepath** (*str*) – Path to read data.
- **encoding** (*str*) – The encoding format used to open the `filepath`. Defaults to 'utf-8'.

Returns Expected text reading from `filepath`.

Return type str

Examples

```
>>> backend = HTTPBackend()
>>> backend.get_text('http://path/of/file')
'hello world'
```

14.1.4 LmdbBackend

class mmeval.fileio.**LmdbBackend**(*db_path*, *readonly=True*, *lock=False*, *readahead=False*, ***kwargs*)
Lmdb storage backend.

Parameters

- **db_path** (*str*) – Lmdb database path.
- **readonly** (*bool*) – Lmdb environment parameter. If True, disallow any write operations. Defaults to True.
- **lock** (*bool*) – Lmdb environment parameter. If False, when concurrent access occurs, do not lock the database. Defaults to False.
- **readahead** (*bool*) – Lmdb environment parameter. If False, disable the OS filesystem readahead mechanism, which may improve random read performance when a database is larger than RAM. Defaults to False.
- ****kwargs** – Keyword arguments passed to *lmdb.open*.

db_path
Lmdb database path.

Type str

get(*filepath: Union[str, pathlib.Path]*) → bytes
Get values according to the filepath.

Parameters **filepath** (*str or Path*) – Here, filepath is the lmdb key.

Returns Expected bytes object.

Return type bytes

Examples

```
>>> backend = LmdbBackend('path/to/lmdb')
>>> backend.get('key')
b'hello world'
```

14.1.5 MemcachedBackend

class mmeval.fileio.**MemcachedBackend**(*server_list_cfg*, *client_cfg*, *sys_path=None*)

Memcached storage backend.

server_list_cfg

Config file for memcached server list.

Type str

client_cfg

Config file for memcached client.

Type str

sys_path

Additional path to be appended to *sys.path*. Defaults to None.

Type str, optional

get(*filepath: Union[str, pathlib.Path]*)

Get values according to the filepath.

Parameters **filepath** (*str or Path*) – Path to read data.

Returns Expected bytes object.

Return type bytes

Examples

```
>>> server_list_cfg = '/path/of/server_list.conf'
>>> client_cfg = '/path/of/mc.conf'
>>> backend = MemcachedBackend(server_list_cfg, client_cfg)
>>> backend.get('/path/of/file')
b'hello world'
```

14.1.6 PetrelBackend

class mmeval.fileio.**PetrelBackend**(*path_mapping: Optional[dict] = None*, *enable_mc: bool = True*)

Petrel storage backend (for internal usage).

PetrelBackend supports reading and writing data to multiple clusters. If the file path contains the cluster name, PetrelBackend will read data from specified cluster or write data to it. Otherwise, PetrelBackend will access the default cluster.

Parameters

- **path_mapping** (*dict, optional*) – Path mapping dict from local path to Petrel path. When `path_mapping={'src': 'dst'}`, `src` in `filepath` will be replaced by `dst`. Defaults to None.
- **enable_mc** (*bool, optional*) – Whether to enable memcached support. Defaults to True.

Examples

```
>>> backend = PetrelBackend()
>>> filepath1 = 'petrel://path/of/file'
>>> filepath2 = 'cluster-name:petrel://path/of/file'
>>> backend.get(filepath1)  # get data from default cluster
>>> client.get(filepath2)  # get data from 'cluster-name' cluster
```

exists(*filepath: Union[str, pathlib.Path]*) → bool

Check whether a file path exists.

Parameters **filepath** (*str or Path*) – Path to be checked whether exists.

Returns Return `True` if `filepath` exists, `False` otherwise.

Return type bool

Examples

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.exists(filepath)
True
```

get(*filepath: Union[str, pathlib.Path]*) → bytes

Read bytes from a given `filepath` with 'rb' mode.

Parameters **filepath** (*str or Path*) – Path to read data.

Returns Return bytes read from filepath.

Return type bytes

Examples

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.get(filepath)
b'hello world'
```

get_local_path(*filepath: Union[str, pathlib.Path]*) → Generator[Union[str, pathlib.Path], None, None]
Download a file from `filepath` to a local temporary directory, and return the temporary path.

`get_local_path` is decorated by `contxtlib.contextmanager()`. It can be called with `with` statement, and when exists from the `with` statement, the temporary path will be released.

Parameters **filepath** (*str or Path*) – Download a file from `filepath`.

Yields *Iterable[str]* – Only yield one temporary path.

Examples

```
>>> backend = PetrelBackend()
>>> # After existing from the ``with`` clause,
>>> # the path will be removed
>>> filepath = 'petrel://path/of/file'
>>> with backend.get_local_path(filepath) as path:
...     # do something here
```

get_text(*filepath: Union[str, pathlib.Path]*, *encoding: str = 'utf-8'*) → str
Read text from a given `filepath` with 'r' mode.

Parameters

- **filepath** (*str or Path*) – Path to read data.
- **encoding** (*str*) – The encoding format used to open the `filepath`. Defaults to 'utf-8'.

Returns Expected text reading from `filepath`.

Return type str

Examples

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.get_text(filepath)
'hello world'
```

isdir(*filepath: Union[str, pathlib.Path]*) → bool
Check whether a file path is a directory.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a directory.

Returns Return `True` if `filepath` points to a directory, `False` otherwise.

Return type bool

Examples

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/dir'
>>> backend.isdir(filepath)
True
```

isfile(*filepath: Union[str, pathlib.Path]*) → bool
Check whether a file path is a file.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a file.

Returns Return `True` if `filepath` points to a file, `False` otherwise.

Return type bool

Examples

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.isfile(filepath)
True
```

join_path(*filepath: Union[str, pathlib.Path]*, **filepaths: Union[str, pathlib.Path]*) → str
Concatenate all file paths.

Join one or more filepath components intelligently. The return value is the concatenation of filepath and any members of *filepaths.

Parameters **filepath** (*str or Path*) – Path to be concatenated.

Returns The result after concatenation.

Return type str

Examples

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.join_path(filepath, 'another/path')
'petrel://path/of/file/another/path'
>>> backend.join_path(filepath, '/another/path')
'petrel://path/of/file/another/path'
```

list_dir_or_file(*dir_path: Union[str, pathlib.Path]*, *list_dir: bool = True*, *list_file: bool = True*, *suffix: Optional[Union[str, Tuple[str]]] = None*, *recursive: bool = False*) → Iterator[str]
Scan a directory to find the interested directories or files in arbitrary order.

Note: Petrel has no concept of directories but it simulates the directory hierarchy in the filesystem through public prefixes. In addition, if the returned path ends with '/', it means the path is a public prefix which is a logical directory.

Note: *list_dir_or_file()* returns the path relative to `dir_path`. In addition, the returned path of directory will not contains the suffix '/' which is consistent with other backends.

Parameters

- **dir_path** (*str | Path*) – Path of the directory.
- **list_dir** (*bool*) – List the directories. Defaults to True.
- **list_file** (*bool*) – List the path of files. Defaults to True.
- **suffix** (*str or tuple[str], optional*) – File suffix that we are interested in. Defaults to None.
- **recursive** (*bool*) – If set to True, recursively scan the directory. Defaults to False.

Yields *Iterable[str]* – A relative path to `dir_path`.

Examples

```
>>> backend = PetrelBackend()
>>> dir_path = 'petrel://path/of/dir'
>>> # list those files and directories in current directory
>>> for file_path in backend.list_dir_or_file(dir_path):
...     print(file_path)
>>> # only list files
>>> for file_path in backend.list_dir_or_file(dir_path, list_dir=False):
...     print(file_path)
>>> # only list directories
>>> for file_path in backend.list_dir_or_file(dir_path, list_file=False):
...     print(file_path)
>>> # only list files ending with specified suffixes
>>> for file_path in backend.list_dir_or_file(dir_path, suffix='.txt'):
...     print(file_path)
>>> # list all files and directory recursively
>>> for file_path in backend.list_dir_or_file(dir_path, recursive=True):
...     print(file_path)
```

register_backend	Register a backend.

14.1.7 mmeval.fileio.register_backend

mmeval.fileio.**register_backend**(*name: str*, *backend: Optional[Type[*mmeval.fileio.backends.base.BaseStorageBackend*]] = None*, *force: bool = False*, *prefixes: Optional[Union[str, list, tuple]] = None*)

Register a backend.

Parameters

- **name** (*str*) – The name of the registered backend.
- **backend** (*class, optional*) – The backend class to be registered, which must be a subclass of *BaseStorageBackend*. When this method is used as a decorator, backend is None.

Defaults to None.

- **force** (*bool*) – Whether to override the backend if the name has already been registered. Defaults to False.
- **prefixes** (*str or list[str] or tuple[str], optional*) – The prefix of the registered storage backend. Defaults to None.

This method can be used as a normal method or a decorator.

Examples

```
>>> class NewBackend(BaseStorageBackend):
...     def get(self, filepath):
...         return filepath
...
...     def get_text(self, filepath):
...         return filepath
>>> register_backend('new', NewBackend)
```

```
>>> @register_backend('new')
... class NewBackend(BaseStorageBackend):
...     def get(self, filepath):
...         return filepath
...
...     def get_text(self, filepath):
...         return filepath
```

14.2 File Handler

BaseFileHandler	A base class for file handler.
JsonHandler	A Json handler that parse json data from file object.
PickleHandler	A Pickle handler that parse pickle data from file object.
YamlHandler	A Yaml handler that parse yaml data from file object.

14.2.1 BaseFileHandler

class mmeval.fileio.**BaseFileHandler**

A base class for file handler.

14.2.2 JsonHandler

class mmeval.fileio.**JsonHandler**
A Json handler that parse json data from file object.

14.2.3 PickleHandler

class mmeval.fileio.**PickleHandler**
A Pickle handler that parse pickle data from file object.

14.2.4 YamlHandler

class mmeval.fileio.**YamlHandler**
A Yaml handler that parse yaml data from file object.

register_handler	A decorator that register a handler for some file extensions.

14.2.5 mmeval.fileio.register_handler

mmeval.fileio.**register_handler**(*file_formats*, ***kwargs*)
A decorator that register a handler for some file extensions.

14.3 File IO

load	Load data from json/yaml/pickle files.
exists	Check whether a file path exists.
get	Read bytes from a given `filepath` with 'rb' mode.
get_file_backend	Return a file backend based on the prefix of uri or backend_args.
get_local_path	Download data from `filepath` and write the data to local path.
get_text	Read text from a given `filepath` with 'r' mode.
isdir	Check whether a file path is a directory.
isfile	Check whether a file path is a file.
join_path	Concatenate all file paths.
list_dir_or_file	Scan a directory to find the interested directories or files in arbitrary order.

14.3.1 mmeval.fileio.load

mmeval.fileio.**load**(*file*, *file_format=None*, *backend_args=None*, ***kwargs*)

Load data from json/yaml/pickle files.

This method provides a unified api for loading data from serialized files.

`load` supports loading data from serialized files those can be storaged in different backends.

Parameters

- **file** (str or `Path` or file-like object) – Filename or a file-like object.
- **file_format** (*str, optional*) – If not specified, the file format will be inferred from the file extension, otherwise use the specified one. Currently supported formats include "json", "yaml/yml" and "pickle/pkl".
- **backend_args** (*dict, optional*) – Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.

Examples

```
>>> load('/path/of/your/file')  # file is storaged in disk
>>> load('https://path/of/your/file')  # file is storaged in Internet
>>> load('s3://path/of/your/file')  # file is storaged in petrel
```

Returns The content from the file.

14.3.2 mmeval.fileio.exists

mmeval.fileio.**exists**(*filepath: Union[str, pathlib.Path]*, *backend_args: Optional[dict] = None*) → bool

Check whether a file path exists.

Parameters

- **filepath** (*str or Path*) – Path to be checked whether exists.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Returns Return `True` if `filepath` exists, `False` otherwise.

Return type bool

Examples

```
>>> filepath = '/path/of/file'
>>> exists(filepath)
True
```

14.3.3 mmeval.fileio.get

mmeval.fileio.**get**(*filepath: Union[str, pathlib.Path]*, *backend_args: Optional[dict] = None*) → bytes

Read bytes from a given `filepath` with 'rb' mode.

Parameters

- **filepath** (*str or Path*) – Path to read data.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Returns Expected bytes object.

Return type bytes

Examples

```
>>> filepath = '/path/of/file'
>>> get(filepath)
b'hello world'
```

14.3.4 mmeval.fileio.get_file_backend

mmeval.fileio.**get_file_backend**(*uri: Optional[Union[str, pathlib.Path]] = None*, ***, *backend_args: Optional[dict] = None*, *enable_singleton: bool = False*)

Return a file backend based on the prefix of uri or backend_args.

Parameters

- **uri** (*str or Path*) – Uri to be parsed that contains the file prefix.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.
- **enable_singleton** (*bool*) – Whether to enable the singleton pattern. If it is True, the backend created will be reused if the signature is same with the previous one. Defaults to False.

Returns Instantiated Backend object.

Return type *BaseStorageBackend*

Examples

```
>>> # get file backend based on the prefix of uri
>>> uri = 's3://path/of/your/file'
>>> backend = get_file_backend(uri)
>>> # get file backend based on the backend_args
>>> backend = get_file_backend(backend_args={'backend': 'petrel'})
>>> # backend name has a higher priority if 'backend' in backend_args
>>> backend = get_file_backend(uri, backend_args={'backend': 'petrel'})
```

14.3.5 mmeval.fileio.get_local_path

mmeval.fileio.**get_local_path**(*filepath: Union[str, pathlib.Path]*, *backend_args: Optional[dict] = None*) → Generator[Union[str, pathlib.Path], None, None]

Download data from `filepath` and write the data to local path.

`get_local_path` is decorated by `contxtlib.contextmanager()`. It can be called with `with` statement, and when exists from the `with` statement, the temporary path will be released.

Note: If the `filepath` is a local path, just return itself and it will not be released (removed).

Parameters

- **filepath** (*str or Path*) – Path to be read data.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Yields *Iterable[str]* – Only yield one path.

Examples

```
>>> with get_local_path('s3://bucket/abc.jpg') as path:
...     # do something here
```

14.3.6 mmeval.fileio.get_text

mmeval.fileio.**get_text**(*filepath: Union[str, pathlib.Path]*, *encoding='utf-8'*, *backend_args: Optional[dict] = None*) → str

Read text from a given `filepath` with ‘r’ mode.

Parameters

- **filepath** (*str or Path*) – Path to read data.
- **encoding** (*str*) – The encoding format used to open the `filepath`. Defaults to ‘utf-8’.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Returns Expected text reading from `filepath`.

Return type str

Examples

```
>>> filepath = '/path/of/file'
>>> get_text(filepath)
'hello world'
```

14.3.7 mmeval.fileio.isdir

mmeval.fileio.**isdir**(*filepath: Union[str, pathlib.Path]*, *backend_args: Optional[dict] = None*) → bool

Check whether a file path is a directory.

Parameters

- **filepath** (*str or Path*) – Path to be checked whether it is a directory.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Returns Return `True` if `filepath` points to a directory, `False` otherwise.

Return type bool

Examples

```
>>> filepath = '/path/of/dir'
>>> isdir(filepath)
True
```

14.3.8 mmeval.fileio.isfile

mmeval.fileio.**isfile**(*filepath: Union[str, pathlib.Path]*, *backend_args: Optional[dict] = None*) → bool

Check whether a file path is a file.

Parameters

- **filepath** (*str or Path*) – Path to be checked whether it is a file.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Returns Return `True` if `filepath` points to a file, `False` otherwise.

Return type bool

Examples

```
>>> filepath = '/path/of/file'
>>> isfile(filepath)
True
```

14.3.9 mmeval.fileio.join_path

mmeval.fileio.**join_path**(*filepath: Union[str, pathlib.Path]*, **filepaths: Union[str, pathlib.Path]*, *backend_args: Optional[dict] = None*) → Union[str, pathlib.Path]

Concatenate all file paths.

Join one or more filepath components intelligently. The return value is the concatenation of filepath and any members of *filepaths.

Parameters

- **filepath** (*str or Path*) – Path to be concatenated.
- ***filepaths** (*str or Path*) – Other paths to be concatenated.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Returns The result of concatenation.

Return type str

Examples

```
>>> filepath1 = '/path/of/dir1'
>>> filepath2 = 'dir2'
>>> filepath3 = 'path/of/file'
>>> join_path(filepath1, filepath2, filepath3)
'/path/of/dir/dir2/path/of/file'
```

14.3.10 mmeval.fileio.list_dir_or_file

mmeval.fileio.**list_dir_or_file**(*dir_path: Union[str, pathlib.Path]*, *list_dir: bool = True*, *list_file: bool = True*, *suffix: Optional[Union[str, Tuple[str]]] = None*, *recursive: bool = False*, *backend_args: Optional[dict] = None*) → Iterator[str]

Scan a directory to find the interested directories or files in arbitrary order.

Note: *list_dir_or_file()* returns the path relative to dir_path.

Parameters

- **dir_path** (*str or Path*) – Path of the directory.
- **list_dir** (*bool*) – List the directories. Defaults to True.
- **list_file** (*bool*) – List the path of files. Defaults to True.
- **suffix** (*str or tuple[str], optional*) – File suffix that we are interested in. Defaults to None.
- **recursive** (*bool*) – If set to True, recursively scan the directory. Defaults to False.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Yields *Iterable[str]* – A relative path to dir_path.

Examples

```
>>> dir_path = '/path/of/dir'
>>> for file_path in list_dir_or_file(dir_path):
...     print(file_path)
>>> # list those files and directories in current directory
>>> for file_path in list_dir_or_file(dir_path):
...     print(file_path)
>>> # only list files
>>> for file_path in list_dir_or_file(dir_path, list_dir=False):
...     print(file_path)
>>> # only list directories
>>> for file_path in list_dir_or_file(dir_path, list_file=False):
...     print(file_path)
>>> # only list files ending with specified suffixes
>>> for file_path in list_dir_or_file(dir_path, suffix='.txt'):
...     print(file_path)
>>> # list all files and directory recursively
>>> for file_path in list_dir_or_file(dir_path, recursive=True):
...     print(file_path)
```

14.4 Parse File

dict_from_file	Load a text file and parse the content as a dict.
list_from_file	Load a text file and parse the content as a list of strings.

14.4.1 mmeval.fileio.dict_from_file

mmeval.fileio.**dict_from_file**(*filename*, *key_type=<class 'str'>*, *encoding='utf-8'*, *backend_args=None*)

Load a text file and parse the content as a dict.

Each line of the text file will be two or more columns split by whitespaces or tabs. The first column will be parsed as dict keys, and the following columns will be parsed as dict values.

`dict_from_file` supports loading a text file which can be storaged in different backends and parsing the content as a dict.

Parameters

- **filename** (*str*) – Filename.
- **key_type** (*type*) – Type of the dict keys. str is user by default and type conversion will be performed if specified.
- **encoding** (*str*) – Encoding used to open the file. Defaults to utf-8.
- **backend_args** (*dict, optional*) – Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.

Examples

```
>>> dict_from_file('/path/of/your/file')  # disk
{'key1': 'value1', 'key2': 'value2'}
>>> dict_from_file('s3://path/of/your/file')  # ceph or petrel
{'key1': 'value1', 'key2': 'value2'}
```

Returns The parsed contents.

Return type dict

14.4.2 mmeval.fileio.list_from_file

mmeval.fileio.**list_from_file**(*filename*, *prefix=''*, *offset=0*, *max_num=0*, *encoding='utf-8'*, *backend_args=None*)

Load a text file and parse the content as a list of strings.

`list_from_file` supports loading a text file which can be storaged in different backends and parsing the content as a list for strings.

Parameters

- **filename** (*str*) – Filename.
- **prefix** (*str*) – The prefix to be inserted to the beginning of each item.
- **offset** (*int*) – The offset of lines.
- **max_num** (*int*) – The maximum number of lines to be read, zeros and negatives mean no limitation.
- **encoding** (*str*) – Encoding used to open the file. Defaults to utf-8.
- **backend_args** (*dict, optional*) – Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.

Examples

```
>>> list_from_file('/path/of/your/file')  # disk
['hello', 'world']
>>> list_from_file('s3://path/of/your/file')  # ceph or petrel
['hello', 'world']
```

Returns A list of strings.

Return type list[str]

CHAPTER

FIFTEEN

MMEVAL.METRICS

mmeval.metrics

- *Metrics*

15.1 Metrics

Accuracy	Top-k accuracy evaluation metric.
SingleLabelMetric	alias of `mmeval.metrics.precision_recall_f1score.SingleLabelPrecisionRecallF1score`
MultiLabelMetric	alias of `mmeval.metrics.precision_recall_f1score.MultiLabelPrecisionRecallF1score`
AveragePrecision	Calculate the average precision with respect of classes.
MeanIoU	MeanIoU evaluation metric.
COCODetection	COCO object detection task evaluation metric.
ProposalRecall	Proposals recall evaluation metric.
VOCMeanAP	Pascal VOC evaluation metric.
OIDMeanAP	Open Images Dataset detection evaluation metric.
F1Score	Compute F1 scores.
HmeanIoU	HmeanIoU metric.
EndPointError	EndPointError evaluation metric.
PCKAccuracy	PCK accuracy evaluation metric, which is widely used in pose estimation.
MpiiPCKAccuracy	PCKh accuracy evaluation metric for MPII dataset.
JhmdbPCKAccuracy	PCK accuracy evaluation metric for Jhmdb dataset.
AVAMeanAP	AVA evaluation metric.
StructuralSimilarity	Calculate StructuralSimilarity (structural similarity).
SignalNoiseRatio	Signal-to-Noise Ratio.
PeakSignalNoiseRatio	Peak Signal-to-Noise Ratio.
MeanAbsoluteError	Mean Absolute Error metric for image.
MeanSquaredError	Mean Squared Error metric for image.
BLEU	Bilingual Evaluation Understudy metric.
SumAbsoluteDifferences	Sum of Absolute Differences metric for image.
GradientError	Gradient error for evaluating alpha matte prediction.

continues on next page

Table 1 – continued from previous page

MattingMeanSquaredError	Mean Squared Error metric for image matting.
ConnectivityError	Connectivity error for evaluating alpha matte prediction.
DOTAMeanAP	DOTA evaluation metric.
ROUGE	Calculate Rouge Score used for automatic summarization.
NaturalImageQualityEvaluator	Calculate Natural Image Quality Evaluator(NIQE) metric.
Perplexity	Perplexity measures how well a language model predicts a text sample.
CharRecallPrecision	Calculate the char level recall & precision.
KeypointEndPointError	EPE evaluation metric.
KeypointAUC	AUC evaluation metric.
KeypointNME	NME evaluation metric.
WordAccuracy	Calculate the word level accuracy.

15.1.1 Accuracy

class mmeval.metrics.**Accuracy**(*topk: Union[int, Sequence[int]] = (1)*, *thrs: Optional[Union[float, Sequence[Optional[float]]]] = 0.0*, ***kwargs*)

Top-k accuracy evaluation metric.

This metric computes the accuracy based on the given topk and thresholds.

Currently, this metric supports 5 kinds of inputs, i.e. `numpy.ndarray`, `torch.Tensor`, `oneflow.Tensor`, `tensorflow.Tensor` and `paddle.Tensor`, and the implementation for the calculation depends on the inputs type.

Parameters

- **topk** (*int | Sequence[int]*) – If the predictions in `topk` matches the target, the predictions will be regarded as correct ones. Defaults to 1.
- **thrs** (*Sequence[float | None] | float | None*) – Predictions with scores under the thresholds are considered negative. None means no thresholds. Defaults to 0.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import Accuracy
>>> accuracy = Accuracy()
```

Use NumPy implementation:

```
>>> import numpy as np
>>> labels = np.asarray([0, 1, 2, 3])
>>> preds = np.asarray([0, 2, 1, 3])
>>> accuracy(preds, labels)
{'top1': 0.5}
```

Use PyTorch implementation:

```
>>> import torch
>>> labels = torch.Tensor([0, 1, 2, 3])
>>> preds = torch.Tensor([0, 2, 1, 3])
>>> accuracy(preds, labels)
{'top1': 0.5}
```

Computing top-k accuracy with specified threold:

```
>>> labels = np.asarray([0, 1, 2, 3])
>>> preds = np.asarray([
    [0.7, 0.1, 0.1, 0.1],
    [0.1, 0.3, 0.4, 0.2],
    [0.3, 0.4, 0.2, 0.1],
    [0.0, 0.0, 0.1, 0.9]])
>>> accuracy = Accuracy(topk=(1, 2, 3))
>>> accuracy(preds, labels)
{'top1': 0.5, 'top2': 0.75, 'top3': 1.0}
>>> accuracy = Accuracy(topk=2, thrs=(0.1, 0.5))
>>> accuracy(preds, labels)
{'top2_thr-0.10': 0.75, 'top2_thr-0.50': 0.5}
```

Accumulate batch:

```
>>> for i in range(10):
...     labels = torch.randint(0, 4, size=(100, ))
...     predicts = torch.randint(0, 4, size=(100, ))
...     accuracy.add(predicts, labels)
>>> accuracy.compute()
```

add(*predictions: Sequence*, *labels: Sequence*) → None

Add the intermediate results to `self._results`.

Parameters

- **predictions** (*Sequence*) – Predictions from the model. It can be labels (N,), or scores of every class (N, C).
- **labels** (*Sequence*) – The ground truth labels. It should be (N,).

compute_metric(*results: List[Union[Iterable, numpy.number, torch.Tensor, tensorflow.Tensor, paddle.Tensor, jax.Array, flow.Tensor]]*) → Dict[str, float]

Compute the accuracy metric.

This method would be invoked in `BaseMetric.compute` after distributed synchronization.

Parameters **results** (*list*) – A list that consisting the correct numbers. This list has already been synced across all ranks.

Returns The computed accuracy metric.

Return type Dict[str, float]

15.1.2 SingleLabelMetric

mmeval.metrics.**SingleLabelMetric**
alias of mmeval.metrics.precision_recall_f1score.SingleLabelPrecisionRecallF1score

15.1.3 MultiLabelMetric

mmeval.metrics.**MultiLabelMetric**
alias of mmeval.metrics.precision_recall_f1score.MultiLabelPrecisionRecallF1score

15.1.4 AveragePrecision

class mmeval.metrics.**AveragePrecision**(*average: Optional[str] = 'macro'*, ***kwargs*)
Calculate the average precision with respect of classes.

Parameters

- **average** (*str, optional*) – The average method. It supports two modes:
  - ***"macro"*: Calculate metrics for each category, and calculate** the mean value over all categories.
  - *None*: Return scores of all categories.
- **to "macro".** (*Defaults*) –

References

Examples

```
>>> from mmeval import AveragePrecision
>>> average_precision = AveragePrecision()
```

Use Builtin implementation with label-format labels:

```
>>> preds = [[0.9, 0.8, 0.3, 0.2],
             [0.1, 0.2, 0.2, 0.1],
             [0.7, 0.5, 0.9, 0.3],
             [0.8, 0.1, 0.1, 0.2]]
>>> labels = [[0, 1], [1], [2], [0]]
>>> average_precision(preds, labels)
{'mAP': 70.833..}
```

Use Builtin implementation with one-hot encoding labels:

```
>>> preds = [[0.9, 0.8, 0.3, 0.2],
             [0.1, 0.2, 0.2, 0.1],
             [0.7, 0.5, 0.9, 0.3],
             [0.8, 0.1, 0.1, 0.2]]
>>> labels = [[1, 1, 0, 0],
              [0, 1, 0, 0],
              [0, 0, 1, 0],
              [1, 0, 0, 0]]
```

(continues on next page)

(continued from previous page)

```
>>> average_precision(preds, labels)
{'mAP': 70.833..}
```

Use NumPy implementation with label-format labels:

```
>>> import numpy as np
>>> preds = np.array([[0.9, 0.8, 0.3, 0.2],
                      [0.1, 0.2, 0.2, 0.1],
                      [0.7, 0.5, 0.9, 0.3],
                      [0.8, 0.1, 0.1, 0.2]])
>>> labels = [np.array([0, 1]), np.array([1]), np.array([2]), np.array([0])] # noqa
>>> average_precision(preds, labels)
{'mAP': 70.833..}
```

Use PyTorch implementation with one-hot encoding labels:

```
>>> import torch
>>> preds = torch.Tensor([[0.9, 0.8, 0.3, 0.2],
                          [0.1, 0.2, 0.2, 0.1],
                          [0.7, 0.5, 0.9, 0.3],
                          [0.8, 0.1, 0.1, 0.2]])
>>> labels = torch.Tensor([[1, 1, 0, 0],
                           [0, 1, 0, 0],
                           [0, 0, 1, 0],
                           [1, 0, 0, 0]])
>>> average_precision(preds, labels)
{'mAP': 70.833..}
```

Computing with *None* average mode:

```
>>> preds = np.array([[0.9, 0.8, 0.3, 0.2],
                      [0.1, 0.2, 0.2, 0.1],
                      [0.7, 0.5, 0.9, 0.3],
                      [0.8, 0.1, 0.1, 0.2]])
>>> labels = [np.array([0, 1]), np.array([1]), np.array([2]), np.array([0])] # noqa
>>> average_precision = AveragePrecision(average=None)
>>> average_precision(preds, labels)
{'AP_classwise': [100.0, 83.33, 100.00, 0.0]}  # rounded results
```

Accumulate batch:

```
>>> for i in range(10):
...     preds = torch.randint(0, 4, size=(100, 10))
...     labels = torch.randint(0, 4, size=(100, ))
...     average_precision.add(preds, labels)
>>> average_precision.compute()
```

add(*preds: Sequence*, *labels: Sequence*) → None

Add the intermediate results to *self._results*.

Parameters

- **preds** (*Sequence*) – Predictions from the model. It should be scores of every class (N, C).

- **labels** (*Sequence*) – The ground truth labels. It should be (N,) for label-format, or (N, C) for one-hot encoding.

compute_metric(*results: List[Union[Tuple[Union[numpy.ndarray, numpy.number], Union[numpy.ndarray, numpy.number]], Tuple[torch.Tensor, torch.Tensor], Tuple[oneflow.Tensor, oneflow.Tensor], Tuple[Union[int, Sequence[Union[int, float]]], Union[int, Sequence[int]]]]]*) → Dict[str, float]

Compute the metric.

Currently, there are 3 implementations of this method: NumPy and PyTorch and OneFlow. Which implementation to use is determined by the type of the calling parameters. e.g. *numpy.ndarray* or *torch.Tensor*, *oneflow.Tensor*.

This method would be invoked in *BaseMetric.compute* after distributed synchronization.

Parameters

- **(List[Union[NUMPY_IMPL_HINTS** (*results*) –
- **TORCH_IMPL_HINTS** –

:param : :param ONEFLOW_IMPL_HINTS]]): A list of tuples that consisting the :param prediction and label. This list has already been synced across: :param all ranks.:

Returns The computed metric.

Return type Dict[str, float]

15.1.5 MeanIoU

class mmeval.metrics.**MeanIoU**(*num_classes: Optional[int] = None*, *ignore_index: int = 255*, *nan_to_num: Optional[int] = None*, *beta: int = 1*, *classwise_results: bool = False*, ***kwargs*)

MeanIoU evaluation metric.

MeanIoU is a widely used evaluation metric for image semantic segmentation.

In addition to mean iou, it will also compute and return accuracy, mean accuracy, mean dice, mean precision, mean recall and mean f-score.

This metric supports 6 kinds of inputs, i.e. `numpy.ndarray`, `torch.Tensor`, `oneflow.Tensor`, `tensorflow.Tensor`, `paddle.Tensor``and``jax.Array`, and the implementation for the calculation depends on the inputs type.

Parameters

- **num_classes** (*int, optional*) – The number of classes. If None, it will be obtained from the 'num_classes' or 'classes' field in *self.dataset_meta*. Defaults to None.
- **ignore_index** (*int, optional*) – Index that will be ignored in evaluation. Defaults to 255.
- **nan_to_num** (*int, optional*) – If specified, NaN values will be replaced by the numbers defined by the user. Defaults to None.
- **beta** (*int, optional*) – Determines the weight of recall in the F-score. Defaults to 1.
- **classwise_results** (*bool, optional*) – Whether to return the computed results of each class. Defaults to False.
- ****kwargs** – Keyword arguments passed to `BaseMetric`.

Examples

```
>>> from mmeval import MeanIoU
>>> miou = MeanIoU(num_classes=4)
```

Use NumPy implementation:

```
>>> import numpy as np
>>> labels = np.asarray([[[0, 1, 1], [2, 3, 2]]])
>>> preds = np.asarray([[[0, 2, 1], [1, 3, 2]]])
>>> miou(preds, labels)
{'aAcc': 0.6666666666666666,
 'mIoU': 0.6666666666666666,
 'mAcc': 0.75,
 'mDice': 0.75,
 'mPrecision': 0.75,
 'mRecall': 0.75,
 'mFscore': 0.75,
 'kappa': 0.5384615384615384}
```

Use PyTorch implementation:

```
>>> import torch
>>> labels = torch.Tensor([[[0, 1, 1], [2, 3, 2]]])
>>> preds = torch.Tensor([[[0, 2, 1], [1, 3, 2]]])
>>> miou(preds, labels)
{'aAcc': 0.6666666666666666,
 'mIoU': 0.6666666666666666,
 'mAcc': 0.75,
 'mDice': 0.75,
 'mPrecision': 0.75,
 'mRecall': 0.75,
 'mFscore': 0.75,
 'kappa': 0.5384615384615384}
```

Accumulate batch:

```
>>> for i in range(10):
...     labels = torch.randint(0, 4, size=(100, 10, 10))
...     predicts = torch.randint(0, 4, size=(100, 10, 10))
...     miou.add(predicts, labels)
>>> miou.compute()
```

add(*predictions: Sequence*, *labels: Sequence*) → None

Process one batch of data and predictions.

Calculate the following 3 stuff from the inputs and store them in `self._results`:

- num_tp_per_class: the number of true positive per-class.
- num_gts_per_class: the number of ground truth per-class.
- num_preds_per_class: the number of predicition per-class.

Parameters

- **predictions** (*Sequence*) – A sequence of the predicted segmentation mask.

- **labels** (*Sequence*) – A sequence of the segmentation mask labels.

compute_confusion_matrix
Compute confusion matrix with NumPy.

Parameters

- **prediction** (*numpy.ndarray*) – The predicition.
- **label** (*numpy.ndarray*) – The ground truth.
- **num_classes** (*int*) – The number of classes.

Returns The computed confusion matrix.

Return type numpy.ndarray

compute_metric(*results: List[Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]]*) → dict
Compute the MeanIoU metric.

This method would be invoked in *BaseMetric.compute* after distributed synchronization.

Parameters **results** (*List[tuple]*) – This list has already been synced across all ranks. This is a list of tuple, and each tuple has the following elements:

- (List[numpy.ndarray]): Each element in the list is the number of true positive per-class on a sample.
- (List[numpy.ndarray]): Each element in the list is the number of ground truth per-class on a sample.
- (List[numpy.ndarray]): Each element in the list is the number of predicition per-class on a sample.

Returns

The computed metric, with following keys:

- aAcc, the overall accuracy, namely pixel accuracy.
- mIoU, the mean Intersection-Over-Union (IoU) for all classes.
- mAcc, the mean accuracy for all classes, namely mean pixel

accuracy. - mDice, the mean dice coefficient for all claases. - mPrecision, the mean precision for all classes. - mRecall, the mean recall for all classes. - mFscore, the mean f-score for all classes. - kappa, the Cohen's kappa coefficient. - classwise_result, the evaluate results of each classes. This would be returned if `self.classwise_result` is True.

Return type Dict

property num_classes: int
Returns the number of classes.

The number of classes should be set during initialization, otherwise it will be obtained from the 'classes' or 'num_classes' field in `self.dataset_meta`.

Raises **RuntimeError** – If the num_classes is not set.

Returns The number of classes.

Return type int

15.1.6 COCODetection

class mmeval.metrics.**COCODetection**(*ann_file: Optional[str] = None*, *metric: Union[str, List[str]] = 'bbox'*, *iou_thrs: Optional[Union[float, Sequence[float]]] = None*, *classwise: bool = False*, *proposal_nums: Sequence[int] = (1, 10, 100)*, *metric_items: Optional[Sequence[str]] = None*, *format_only: bool = False*, *outfile_prefix: Optional[str] = None*, *gt_mask_area: bool = True*, *backend_args: Optional[dict] = None*, *print_results: bool = True*, ***kwargs*)

COCO object detection task evaluation metric.

Evaluate AR, AP, and mAP for detection tasks including proposal/box detection and instance segmentation. Please refer to https://cocodataset.org/#detection-eval for more details.

Parameters

- **ann_file** (*str, optional*) – Path to the coco format annotation file. If not specified, ground truth annotations from the dataset will be converted to coco format. Defaults to None.
- **metric** (*str | List[str]*) – Metrics to be evaluated. Valid metrics include 'bbox', 'segm', and 'proposal'. Defaults to 'bbox'.
- **iou_thrs** (*float | List[float], optional*) – IoU threshold to compute AP and AR. If not specified, IoUs from 0.5 to 0.95 will be used. Defaults to None.
- **classwise** (*bool*) – Whether to return the computed results of each class. Defaults to False.
- **proposal_nums** (*Sequence[int]*) – Numbers of proposals to be evaluated. Defaults to (1, 10, 100).
- **metric_items** (*List[str], optional*) – Metric result names to be recorded in the evaluation result. Defaults to None.
- **format_only** (*bool*) – Format the output results without perform evaluation. It is useful when you want to format the result to a specific format and submit it to the test server. Defaults to False.
- **outfile_prefix** (*str, optional*) – The prefix of json files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Defaults to None.
- **gt_mask_area** (*bool*) – Whether to calculate GT mask area when not loading ann_file. If True, the GT instance area will be the mask area, else the bounding box area. It will not be used when loading ann_file. Defaults to True.
- **backend_args** (*dict, optional*) – Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.
- **print_results** (*bool*) – Whether to print the results. Defaults to True.
- **logger** (*Logger, optional*) – logger used to record messages. When set to `None`, the default logger will be used. Defaults to None.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> import numpy as np
>>> from mmeval import COCODetection
>>> try:
>>>     from mmeval.metrics.utils.coco_wrapper import mask_util
>>> except ImportError as e:
>>>     mask_util = None
>>>
>>> num_classes = 4
>>> fake_dataset_metas = {
...     'classes': tuple([str(i) for i in range(num_classes)])
... }
>>>
>>> coco_det_metric = COCODetection(
...     dataset_meta=fake_dataset_metas,
...     metric=['bbox', 'segm']
... )
>>> def _gen_bboxes(num_bboxes, img_w=256, img_h=256):
...     # random generate bounding boxes in 'xyxy' formart.
...     x = np.random.rand(num_bboxes, ) * img_w
...     y = np.random.rand(num_bboxes, ) * img_h
...     w = np.random.rand(num_bboxes, ) * (img_w - x)
...     h = np.random.rand(num_bboxes, ) * (img_h - y)
...     return np.stack([x, y, x + w, y + h], axis=1)
>>>
>>> def _gen_masks(bboxes, img_w=256, img_h=256):
...     if mask_util is None:
...         raise ImportError(
...             'Please try to install official pycocotools by '
...             '"pip install pycocotools"')
...     masks = []
...     for i, bbox in enumerate(bboxes):
...         mask = np.zeros((img_h, img_w))
...         bbox = bbox.astype(np.int32)
...         box_mask = (np.random.rand(
...             bbox[3] - bbox[1],
...             bbox[2] - bbox[0]) > 0.3).astype(np.int32)
...         mask[bbox[1]:bbox[3], bbox[0]:bbox[2]] = box_mask
...         masks.append(
...             mask_util.encode(
...                 np.array(mask[:, :, np.newaxis], order='F',
...                          dtype='uint8'))[0])  # encoded with RLE
...     return masks
>>>
>>> img_id = 1
>>> img_w, img_h = 256, 256
>>> num_bboxes = 10
>>> pred_boxes = _gen_bboxes(
...     num_bboxes=num_bboxes,
...     img_w=img_w,
...     img_h=img_h)
>>> pred_masks = _gen_masks(
```

(continues on next page)

(continued from previous page)

```
...     bboxes=pred_boxes,
...     img_w=img_w,
...     img_h=img_h)
>>> prediction = {
...     'img_id': img_id,
...     'bboxes': pred_boxes,
...     'scores': np.random.rand(num_bboxes, ),
...     'labels': np.random.randint(0, num_classes, size=(num_bboxes, )),
...     'masks': pred_masks
... }
>>> gt_boxes = _gen_bboxes(
...     num_bboxes=num_bboxes,
...     img_w=img_w,
...     img_h=img_h)
>>> gt_masks = _gen_masks(
...     bboxes=pred_boxes,
...     img_w=img_w,
...     img_h=img_h)
>>> groundtruth = {
...     'img_id': img_id,
...     'width': img_w,
...     'height': img_h,
...     'bboxes': gt_boxes,
...     'labels': np.random.randint(0, num_classes, size=(num_bboxes, )),
...     'masks': gt_masks,
...     'ignore_flags': np.zeros(num_bboxes)
... }
>>> coco_det_metric(predictions=[prediction, ], groundtruths=[groundtruth, ])
{'bbox_mAP': ..., 'bbox_mAP_50': ..., ...,
 'segm_mAP': ..., 'segm_mAP_50': ..., ...,
 'bbox_result': ..., 'segm_result': ..., ...}
```

add(*predictions: Sequence[Dict]*, *groundtruths: Sequence[Dict]*) → None

Add the intermediate results to *self._results*.

Parameters

- **predictions** (*Sequence[dict]*) – A sequence of dict. Each dict representing a detection result for an image, with the following keys:
  - img_id (int): Image id.
  - bboxes (numpy.ndarray): Shape (N, 4), the predicted bounding bboxes of this image, in 'xyxy' foramrt.
  - scores (numpy.ndarray): Shape (N,), the predicted scores of bounding boxes.
  - labels (numpy.ndarray): Shape (N,), the predicted labels of bounding boxes.
  - masks (list[RLE], optional): The predicted masks.
  - mask_scores (np.array, optional): Shape (N,), the predicted scores of masks.
- **groundtruths** (*Sequence[dict]*) – A sequence of dict. If load from *ann_file*, the dict inside can be empty. Else, each dict represents a groundtruths for an image, with the following keys:
  - img_id (int): Image id.

- width (int): The width of the image.
- height (int): The height of the image.
- bboxes (numpy.ndarray): Shape (K, 4), the ground truth bounding bboxes of this image, in 'xyxy' foramrt.
- labels (numpy.ndarray): Shape (K,), the ground truth labels of bounding boxes.
- masks (list[RLE], optional): The predicted masks.
- ignore_flags (numpy.ndarray, optional): Shape (K,), the ignore flags.

add_predictions(*predictions: Sequence[Dict]*) → None
Add predictions only.

If the *ann_file* has been passed, we can add predictions only.

Parameters **predictions** (*Sequence[dict]*) – Refer to *COCODetection.add*.

property classes: list
Get classes from self.dataset_meta.

compute_metric(*results: list*) → dict
Compute the COCO metrics.

Parameters **results** (*List[tuple]*) – A list of tuple. Each tuple is the prediction and ground truth of an image. This list has already been synced across all ranks.

Returns The computed metric. The keys are the names of the metrics, and the values are corresponding results.

Return type dict

gt_to_coco_json(*gt_dicts: Sequence[dict]*, *outfile_prefix: str*) → str
Convert ground truth to coco format json file.

Parameters

- **gt_dicts** (*Sequence[dict]*) – Ground truth of the dataset.
- **outfile_prefix** (*str*) – The filename prefix of the json files. If the prefix is "somepath/xxx", the json file will be named "somepath/xxx.gt.json".

Returns The filename of the json file.

Return type str

results2json(*results: Sequence[dict]*, *outfile_prefix: str*) → dict
Dump the detection results to a COCO style json file.

There are 3 types of results: proposals, bbox predictions, mask predictions, and they have different data types. This method will automatically recognize the type, and dump them to json files.

Parameters

- **results** (*Sequence[dict]*) – Testing results of the dataset.
- **outfile_prefix** (*str*) – The filename prefix of the json files. If the prefix is "somepath/xxx", the json files will be named "somepath/xxx.bbox.json", "somepath/xxx.segm.json", "somepath/xxx.proposal.json".

Returns Possible keys are "bbox", "segm", "proposal", and values are corresponding filenames.

Return type dict

xyxy2xywh(*bbox: numpy.ndarray*) → list
Convert `xyxy` style bounding boxes to `xywh` style for COCO evaluation.

Parameters **bbox** (*np.ndarray*) – The bounding boxes, shape (4,), in `xyxy` order.

Returns The converted bounding boxes, in `xywh` order.

Return type list[float]

15.1.7 ProposalRecall

class mmeval.metrics.**ProposalRecall**(*iou_thrs: Optional[Union[float, Sequence[float]]] = None*, *proposal_nums: Union[int, Sequence[int]] = (1, 10, 100, 1000)*, *use_legacy_coordinate: bool = False*, *nproc: int = 4*, *print_results: bool = True*, ***kwargs*)
Proposals recall evaluation metric.

The speed of calculating recall is faster than COCO Detection metric.

Parameters

- **iou_thrs** (*float | List[float], optional*) – IoU thresholds. If not specified, IoUs from 0.5 to 0.95 will be used. Defaults to None.
- **proposal_nums** (*Sequence[int]*) – Numbers of proposals to be evaluated. Defaults to (1, 10, 100, 1000).
- **use_legacy_coordinate** (*bool*) – Whether to use coordinate system in mmdet v1.x. which means width, height should be calculated as 'x2 - x1 + 1` and 'y2 - y1 + 1' respectively. Defaults to False.
- **nproc** (*int*) – Processes used for computing TP and FP. If nproc is less than or equal to 1, multiprocessing will not be used. Defaults to 4.
- **print_results** (*bool*) – Whether to print the results. Defaults to True.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> import numpy as np
>>> from mmeval import ProposalRecall
>>>
>>> proposal_recall = ProposalRecall()
>>> def _gen_bboxes(num_bboxes, img_w=256, img_h=256):
...     # random generate bounding boxes in 'xyxy' formart.
...     x = np.random.rand(num_bboxes, ) * img_w
...     y = np.random.rand(num_bboxes, ) * img_h
...     w = np.random.rand(num_bboxes, ) * (img_w - x)
...     h = np.random.rand(num_bboxes, ) * (img_h - y)
...     return np.stack([x, y, x + w, y + h], axis=1)
>>>
>>> prediction = {
...     'bboxes': _gen_bboxes(10),
...     'scores': np.random.rand(10, ),
... }
>>> groundtruth = {
```

(continues on next page)

(continued from previous page)

```
...     'bboxes': _gen_bboxes(10),
... }
>>> proposal_recall(predictions=[prediction, ], groundtruths=[groundtruth, ])
{'AR@1': ..., 'AR@10': ..., 'AR@100': ..., 'AR@1000': ...}
```

add(*predictions: Sequence[Dict]*, *groundtruths: Sequence[Dict]*) → None
Add the intermediate results to `self._results`.

Parameters

- **predictions** (*Sequence[dict]*) – A sequence of dict. Each dict representing a detection result for an image, with the following keys:
  - bboxes (numpy.ndarray): Shape (N, 4), the predicted bounding bboxes of this image, in 'xyxy' foramrt.
  - scores (numpy.ndarray): Shape (N, 1), the predicted scores of bounding boxes.
- **groundtruths** (*Sequence[dict]*) – A sequence of dict. Each dict represents a groundtruths for an image, with the following keys:
  - bboxes (numpy.ndarray): Shape (M, 4), the ground truth bounding bboxes of this image, in 'xyxy' foramrt.

calculate_recall(*predictions: List[dict]*, *groundtruths: List[dict]*, *pool: Optional[multiprocessing.pool.Pool]*)
Calculate recall.

Parameters

- **predictions** (*List[dict]*) – A list of dict. Each dict is the detection result of an image. Same as *ProposalRecall.add*.
- **groundtruths** (*List[dict]*) – A list of dict. Each dict is the ground truth of an image. Same as *ProposalRecall.add*.
- **pool** (*Optional[Pool]*) – A instance of `multiprocessing.Pool`. If None, do not use multiprocessing.

Returns Shape (len(proposal_nums), len(iou_thrs)), the recall results.

Return type numpy.ndarry

compute_metric(*results: list*) → dict
Compute the ProposalRecall metric.

Parameters **results** (*List[tuple]*) – A list of tuple. Each tuple is the prediction and ground truth of an image. This list has already been synced across all ranks.

Returns The computed metric. The keys are the names of the proposal numbers, and the values are corresponding results.

Return type dict

process_proposals(*predictions: List[dict]*, *groundtruths: List[dict]*) → Tuple[List, List, int]
Process the proposals(bboxes) in predictions and groundtruths.

Parameters

- **predictions** (*List[dict]*) – A list of dict. Each dict is the detection result of an image. Same as *ProposalRecall.add*.
- **groundtruths** (*list[dict]*) – Same as *ProposalRecall.add*.

Returns

- proposal_preds (List[numpy.ndarray]): The sorted proposals of the prediction.
- proposal_gts (List[numpy.ndarray]): The proposals of the groundtruths.
- num_gts (int): The total groundtruth numbers.

Return type tuple (proposal_preds, proposal_gts, num_gts)

15.1.8 VOCMeanAP

class mmeval.metrics.**VOCMeanAP**(*iou_thrs: Union[float, List[float]] = 0.5*, *scale_ranges: Optional[List[Tuple]] = None*, *num_classes: Optional[int] = None*, *eval_mode: str = 'area'*, *use_legacy_coordinate: bool = False*, *nproc: int = 4*, *drop_class_ap: bool = True*, *classwise: bool = False*, ***kwargs*)

Pascal VOC evaluation metric.

This metric computes the VOC mAP (mean Average Precision) with the given IoU thresholds and scale ranges.

Parameters

- **iou_thrs** (*float List[float]*) – IoU thresholds. Defaults to 0.5.
- **scale_ranges** (*List[tuple], optional*) – Scale ranges for evaluating mAP. If not specified, all bounding boxes would be included in evaluation. Defaults to None.
- **num_classes** (*int, optional*) – The number of classes. If None, it will be obtained from the 'classes' field in `self.dataset_meta`. Defaults to None.
- **eval_mode** (*str*) – 'area' or '11points', 'area' means calculating the area under precision-recall curve, '11points' means calculating the average precision of recalls at [0, 0.1, ..., 1]. The PASCAL VOC2007 defaults to use '11points', while PASCAL VOC2012 defaults to use 'area'. Defaults to 'area'.
- **use_legacy_coordinate** (*bool*) – Whether to use coordinate system in mmdet v1.x. which means width, height should be calculated as 'x2 - x1 + 1` and 'y2 - y1 + 1' respectively. Defaults to False.
- **nproc** (*int*) – Processes used for computing TP and FP. If nproc is less than or equal to 1, multiprocessing will not be used. Defaults to 4.
- **drop_class_ap** (*bool*) – Whether to drop the class without ground truth when calculating the average precision for each class.
- **classwise** (*bool*) – Whether to return the computed results of each class. Defaults to False.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> import numpy as np
>>> from mmeval import VOCMeanAP
>>> num_classes = 4
>>> voc_map = VOCMeanAP(num_classes=4)
>>>
>>> def _gen_bboxes(num_bboxes, img_w=256, img_h=256):
...     # random generate bounding boxes in 'xyxy' formart.
...     x = np.random.rand(num_bboxes, ) * img_w
```

(continues on next page)

(continued from previous page)

```
...     y = np.random.rand(num_bboxes, ) * img_h
...     w = np.random.rand(num_bboxes, ) * (img_w - x)
...     h = np.random.rand(num_bboxes, ) * (img_h - y)
...     return np.stack([x, y, x + w, y + h], axis=1)
>>>
>>> prediction = {
...     'bboxes': _gen_bboxes(10),
...     'scores': np.random.rand(10, ),
...     'labels': np.random.randint(0, num_classes, size=(10, ))
... }
>>> groundtruth = {
...     'bboxes': _gen_bboxes(10),
...     'labels': np.random.randint(0, num_classes, size=(10, )),
...     'bboxes_ignore': _gen_bboxes(5),
...     'labels_ignore': np.random.randint(0, num_classes, size=(5, ))
... }
>>> voc_map(predictions=[prediction, ], groundtruths=[groundtruth, ])
{'AP50': ..., 'mAP': ...}
```

add(*predictions: Sequence[Dict]*, *groundtruths: Sequence[Dict]*) → None

Add the intermediate results to `self._results`.

Parameters

- **predictions** (*Sequence[dict]*) – A sequence of dict. Each dict representing a detection result for an image, with the following keys:
  - bboxes (numpy.ndarray): Shape (N, 4), the predicted bounding bboxes of this image, in 'xyxy' foramrt.
  - scores (numpy.ndarray): Shape (N, 1), the predicted scores of bounding boxes.
  - labels (numpy.ndarray): Shape (N, 1), the predicted labels of bounding boxes.
- **groundtruths** (*Sequence[dict]*) – A sequence of dict. Each dict represents a groundtruths for an image, with the following keys:
  - bboxes (numpy.ndarray): Shape (M, 4), the ground truth bounding bboxes of this image, in 'xyxy' foramrt.
  - labels (numpy.ndarray): Shape (M, 1), the ground truth labels of bounding boxes.
  - bboxes_ignore (numpy.ndarray): Shape (K, 4), the ground truth ignored bounding bboxes of this image, in 'xyxy' foramrt.
  - labels_ignore (numpy.ndarray): Shape (K, 1), the ground truth ignored labels of bounding boxes.

calculate_class_tpfp(*predictions: List[dict]*, *groundtruths: List[dict]*, *class_index: int*, *pool: Optional[multiprocessing.pool.Pool]*) → Tuple

Calculate the tp and fp of the given class index.

Parameters

- **predictions** (*List[dict]*) – A list of dict. Each dict is the detection result of an image. Same as *VOCMeanAP.add*.
- **groundtruths** (*List[dict]*) – A list of dict. Each dict is the ground truth of an image. Same as *VOCMeanAP.add*.

- **class_index** (*int*) – The class index.
- **pool** (*Optional[Pool]*) – A instance of `multiprocessing.Pool`. If None, do not use multiprocessing.

Returns

- tp (numpy.ndarray): Shape (num_ious, num_scales, num_pred), the true positive flag of each predicted bbox for this class.
- fp (numpy.ndarray): Shape (num_ious, num_scales, num_pred), the false positive flag of each predicted bbox for this class.
- num_gts (numpy.ndarray): Shape (num_ious, num_scales), the number of ground truths.

Return type tuple (tp, fp, num_gts)

compute_metric(*results: list*) → dict
Compute the VOCMeanAP metric.

Parameters **results** (*List[tuple]*) – A list of tuple. Each tuple is the prediction and ground truth of an image. This list has already been synced across all ranks.

Returns

The computed metric, with the following keys:

- mAP, the averaged across all IoU thresholds and all class.
- AP{IoU}, the mAP of the specified IoU threshold.
- mAP@{scale_range}, the mAP of the specified scale range.
- classwise, the evaluation results of each class. This would be returned if `self.classwise` is True.

Return type dict

get_class_gts(*groundtruths: List[dict]*, *class_index: int*) → Tuple
Get prediciton gt information of a certain class index.

Parameters

- **groundtruths** (*list[dict]*) – Same as *VOCMeanAP.add*.
- **class_index** (*int*) – Index of a specific class.

Returns

- class_gts (List[numpy.ndarray]): The gt bboxes of this class.
- class_ignore_gts (List[numpy.ndarray]): The ignored gt bboxes of this class. This is necessary when counting tp and fp.

Return type tuple (class_gts, class_ignore_gts)

get_class_predictions(*predictions: List[dict]*, *class_index: int*) → List
Get prediciton results of a certain class index.

Parameters

- **predictions** (*list[dict]*) – Same as *VOCMeanAP.add*.
- **class_index** (*int*) – Index of a specific class.

Returns A list of predicted bboxes of this class. Each predicted score of the bbox is concatenated behind the predicted bbox.

Return type list[np.ndarray]

property num_classes: int

Returns the number of classes.

The number of classes should be set during initialization, otherwise it will be obtained from the 'classes' field in `self.dataset_meta`.

Returns The number of classes.

Return type int

Raises **RuntimeError** – If the num_classes is not set.

15.1.9 OIDMeanAP

class mmeval.metrics.**OIDMeanAP**(*iof_thrs: Union[float, List[float]] = 0.5*, *use_group_of: bool = True*, *get_supercategory: bool = True*, *filter_labels: bool = True*, *class_relation_matrix: Optional[numpy.ndarray] = None*, ***kwargs*)

Open Images Dataset detection evaluation metric.

The Open Images Dataset detection evaluation uses a variant of the standard PASCAL VOC 2010 mean Average Precision (mAP) at IoU > 0.5. There are some key features of Open Images annotations, which are addressed by the new metric.

For more see: https://storage.googleapis.com/openimages/web/evaluation.html

Parameters

- **iof_thrs** (*float | List[float]*) – IoF thresholds. Defaults to 0.5.
- **use_group_of** (*bool*) – Whether consider group of groud truth bboxes during evaluating. Defaults to True.
- **get_supercategory** (*bool, optional*) – Whether to get parent class of the current class. Defaults to True.
- **filter_labels** (*bool, optional*) – Whether filter unannotated classes. Defaults to True.
- **class_relation_matrix** (*numpy.ndarray, optional*) – The matrix of the corresponding relationship between the parent class and the child class. If None, it will be obtained from the 'relation_matrix' field in `self.dataset_meta`. Defaults to None.
- ****kwargs** – Keyword parameters passed to *VOCMeanAP*.

Examples

```
>>> import numpy as np
>>> from mmeval import OIDMeanAP
>>> num_classes = 4
>>> # use a fake relation_matrix
>>> relation_matrix = np.eye(num_classes, num_classes)
>>> oid_map = OIDMeanAP(
...     num_classes=4, class_relation_matrix=relation_matrix)
>>>
>>> def _gen_bboxes(num_bboxes, img_w=256, img_h=256):
...     # random generate bounding boxes in 'xyxy' formart.
```

(continues on next page)

(continued from previous page)

```
...     x = np.random.rand(num_bboxes, ) * img_w
...     y = np.random.rand(num_bboxes, ) * img_h
...     w = np.random.rand(num_bboxes, ) * (img_w - x)
...     h = np.random.rand(num_bboxes, ) * (img_h - y)
...     return np.stack([x, y, x + w, y + h], axis=1)
>>>
>>> prediction = {
...     'bboxes': _gen_bboxes(10),
...     'scores': np.random.rand(10, ),
...     'labels': np.random.randint(0, num_classes, size=(10, ))
... }
>>> instances = []
>>> for bbox in _gen_bboxes(20):
...     instances.append({
...         'bbox': bbox.tolist(),
...         'bbox_label': random.randint(0, num_classes - 1),
...         'is_group_of': random.randint(0, 1) == 0,
...     })
>>> groundtruth = {
...     'instances': instances,
...     'image_level_labels': np.random.randint(0, num_classes, size=(10, )),  #␣
↪noqa: E501
... }
>>> oid_map(predictions=[prediction, ], groundtruths=[groundtruth, ])
{'AP@50': ..., 'mAP': ...}
```

add(*predictions: Sequence[dict]*, *groundtruths: Sequence[dict]*) → None

Add the intermediate results to `self._results`.

Parameters

- **predictions** (*Sequence[dict]*) – A sequence of dict. Each dict representing a detection result for an image, with the following keys:
  - bboxes (numpy.ndarray): Shape (N, 4), the predicted bounding bboxes of this image, in 'xyxy' foramrt.
  - scores (numpy.ndarray): Shape (N, 1), the predicted scores of bounding boxes.
  - labels (numpy.ndarray): Shape (N, 1), the predicted labels of bounding boxes.
- **groundtruths** (*Sequence[dict]*) – A sequence of dict. Each dict representing a groundtruths for an image, with the following keys:
  - instances (List[dict]): Each dict representing a bounding box with the following keys:
    - bbox (list): Containing box location in 'xyxy' foramrt.
    - bbox_label (int): Box label index.
    - is_group_of (bool): Whether the box is group or not.
  - image_level_labels (numpy.ndarray): The image level labels.

calculate_class_tpfp(*predictions: List[dict]*, *groundtruths: List[dict]*, *class_index: int*, *pool: Optional[multiprocessing.pool.Pool]*) → Tuple

Calculate the tp and fp of the given class index.

This is an overridden method of *VOCMeanAP*.

Parameters

- **predictions** (*List[dict]*) – A list of dict. Each dict is the detection result of an image. Same as *VOCMeanAP.add*.
- **groundtruths** (*List[dict]*) – A list of dict. Each dict is the ground truth of an image. Same as *VOCMeanAP.add*.
- **class_index** (*int*) – The class index.
- **pool** (*Pool, optional*) – An instance of class:*multiprocessing.Pool*. If None, do not use multiprocessing.

Returns

- tp (numpy.ndarray): Shape (num_ious, num_scales, num_pred), the true positive flag of each predict bbox.
- fp (numpy.ndarray): Shape (num_ious, num_scales, num_pred), the false positive flag of each predict bbox.
- num_gts (numpy.ndarray): Shape (num_ious, num_scales), the number of ground truths.

Return type tuple (tp, fp, num_gts)

property class_relation_matrix: numpy.ndarray

Returns the class relation matrix.

The class relation matrix should be set during initialization, otherwise it will be obtained from the 'relation_matrix' field in `self.dataset_meta`.

Returns The class relation matrix.

Return type numpy.ndarray

Raises **RuntimeError** – If the class relation matrix is not set.

get_class_gts(*groundtruths: List[dict]*, *class_index: int*) → Tuple

Get prediciton gt information of a certain class index.

This is an overridden method of *VOCMeanAP*.

Parameters

- **groundtruths** (*list[dict]*) – Same as `self.add`.
- **class_index** (*int*) – Index of a specific class.

Returns gt bboxes.

Return type List[np.ndarray]

15.1.10 F1Score

class mmeval.metrics.**F1Score**(*num_classes: int*, *mode: Union[str, Sequence[str]] = 'micro'*, *cared_classes: Sequence[int] = []*, *ignored_classes: Sequence[int] = []*, ***kwargs*)

Compute F1 scores.

Parameters

- **num_classes** (*int*) – Number of labels.
- **mode** (*str or list[str]*) – There are 2 options:

  - 'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.
  - 'macro': Calculate metrics for each label, and find their unweighted mean.

  If mode is a list, then metrics in mode will be calculated separately. Defaults to 'micro'.

- **cared_classes** (*list[int]*) – The indices of the labels participated in the metric computing. If both `cared_classes` and `ignored_classes` are empty, all classes will be taken into account. Defaults to []. Note: `cared_classes` and `ignored_classes` cannot be specified together.
- **ignored_classes** (*list[int]*) – The index set of labels that are ignored when computing metrics. If both `cared_classes` and `ignored_classes` are empty, all classes will be taken into account. Defaults to []. Note: `cared_classes` and `ignored_classes` cannot be specified together.
- ****kwargs** – Keyword arguments passed to `BaseMetric`.

Warning: Only non-negative integer labels are involved in computing. All negative ground truth labels will be ignored.

Examples

```
>>> from mmeval import F1Score
>>> f1 = F1Score(num_classes=5, mode=['macro', 'micro'])
```

Use NumPy implementation:

```
>>> import numpy as np
>>> labels = np.asarray([0, 1, 4])
>>> preds = np.asarray([0, 1, 2])
>>> f1(preds, labels)
{'macro_f1': 0.4,
 'micro_f1': 0.6666666666666666}
```

Use PyTorch implementation:

```
>>> import torch
>>> labels = torch.Tensor([0, 1, 4])
>>> preds = torch.Tensor([0, 1, 2])
>>> f1(preds, labels)
{'macro_f1': 0.4,
 'micro_f1': 0.6666666666666666}
```

Accumulate batch:

```
>>> for i in range(10):
...     labels = torch.randint(0, 4, size=(20, ))
...     predicts = torch.randint(0, 4, size=(20, ))
...     f1.add(predicts, labels)
>>> f1.compute()
```

add(*predictions: Sequence[Union[Sequence[int], numpy.ndarray]]*, *labels: Sequence[Union[Sequence[int], numpy.ndarray]]*) → None
Process one batch of data and predictions.

Calculate the following 2 stuff from the inputs and store them in `self._results`:

- prediction: prediction labels.
- label: ground truth labels.

Parameters

- **predictions** (*Sequence[Sequence[int] or np.ndarray]*) – A batch of sequences of non-negative integer labels.
- **labels** (*Sequence[Sequence[int] or np.ndarray]*) – A batch of sequences of non-negative integer labels.

compute_metric(*results: Sequence[Tuple[numpy.ndarray, numpy.ndarray]]*) → Dict
Compute the metrics from processed results.

Parameters **results** (*list[(ndarray, ndarray)]*) – The processed results of each batch.

Returns The f1 scores. The keys are the names of the metrics, and the values are corresponding results. Possible keys are 'micro_f1' and 'macro_f1'.

Return type dict[str, float]

15.1.11 HmeanIoU

class mmeval.metrics.**HmeanIoU**(*match_iou_thr: float = 0.5*, *ignore_precision_thr: float = 0.5*, *pred_score_thrs: Dict = {'start': 0.3, 'step': 0.1, 'stop': 0.9}*, *strategy: str = 'vanilla'*, ***kwargs*)
HmeanIoU metric.

This method computes the hmean iou metric of polygons, and accepts parameters orgnaized as follows:

- batch_pred_polygons (Sequence[Sequence[np.ndarray]]): A batch of prediction polygons, where each element is a sequence of polygons. Each polygon is represented in the form of [x1, y1, x2, y2, ...].
- batch_pred_scores (Sequence): A batch of prediction scores, where each element is a sequence of scores.
- batch_pred_polygons (Sequence[Sequence[np.ndarray]]): A batch of ground truth polygons, where each element is a sequence of polygons. Each polygon is represented in the form of [x1, y1, x2, y2, ...].
- batch_gt_ignore_flags (Sequence): A batch of boolean flags indicating whether to ignore the corresponding ground truth polygon. Each element is a sequence of flags.

The evaluation is done in the following steps:

- Filter the prediction polygon:
  - Score is smaller than minimum prediction score threshold.
  - The proportion of the area that intersects with gt ignored polygon is greater than ignore_precision_thr.
- Computing an M x N IoU matrix, where each element indexing E_mn represents the IoU between the m-th valid GT and n-th valid prediction.
- Based on different prediction score threshold:
  - Obtain the ignored predictions according to prediction score. The filtered predictions will not be involved in the later metric computations.

  - Based on the IoU matrix, get the match metric according to `match_iou_thr`.
  - Based on different *strategy*, accumulate the match number.
- calculate H-mean under different prediction score threshold.

Parameters

- **match_iou_thr** (*float*) – IoU threshold for a match. Defaults to 0.5.
- **ignore_precision_thr** (*float*) – Precision threshold when prediction and gt ignored polygons are matched. Defaults to 0.5.
- **pred_score_thrs** (*dict*) – Best prediction score threshold searching space. Defaults to dict(start=0.3, stop=0.9, step=0.1).
- **strategy** (*str*) – Polygon matching strategy. Options are 'max_matching' and 'vanilla'. 'max_matching' refers to the optimum strategy that maximizes the number of matches. Vanilla strategy matches gt and pred polygons if both of them are never matched before. It was used in academia. Defaults to 'vanilla'.
- ****kwargs** – Keyword arguments passed to `BaseMetric`.

Examples

```
>>> from mmeval import HmeanIoU
>>> import numpy as np
>>> hmeaniou = HmeanIoU(pred_score_thrs=dict(start=0.5, stop=0.7, step=0.1))  # noqa
>>> gt_polygons = [[np.array([0, 0, 1, 0, 1, 1, 0, 1])]]
>>> pred_polygons = [[
...     np.array([0, 0, 1, 0, 1, 1, 0, 1]),
...     np.array([0, 0, 1, 0, 1, 1, 0, 1]),
... ]]
>>> pred_scores = [np.array([1, 0.5])]
>>> gt_ignore_flags = [[False]]
>>> hmeaniou(pred_polygons, pred_scores, gt_polygons, gt_ignore_flags)
{
    0.5: {'precision': 0.5, 'recall': 1.0, 'hmean': 0.6666666666666666},  # noqa
    0.6: {'precision': 1.0, 'recall': 1.0, 'hmean': 1.0},
    'best': {'precision': 1.0, 'recall': 1.0, 'hmean': 1.0}
}
```

add(*batch_pred_polygons: Sequence[Sequence[numpy.ndarray]]*, *batch_pred_scores: Sequence*, *batch_gt_polygons: Sequence[Sequence[numpy.ndarray]]*, *batch_gt_ignore_flags: Sequence*) → None

Process one batch of data and predictions.

Parameters

- **batch_pred_polygons** (*Sequence[Sequence[np.ndarray]]*) – A batch of prediction polygons, where each element is a sequence of polygons. Each polygon is represented in the form of [x1, y1, x2, y2, ...].
- **batch_pred_scores** (*Sequence*) – A batch of prediction scores, where each element is a sequence of scores.
- **batch_pred_polygons** – A batch of ground truth polygons, where each element is a sequence of polygons. Each polygon is represented in the form of [x1, y1, x2, y2, ...].

- **batch_gt_ignore_flags** (*Sequence*) – A batch of boolean flags indicating whether to ignore the corresponding ground truth polygon. Each element is a sequence of flags.

compute_metric(*results: Sequence[Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]]*) → Dict

Compute the metrics from processed results.

Parameters **results** (*list[(np.ndarray, np.ndarray, np.ndnarray, np.ndarray]*) – The processed results of each batch.

Returns Nested dicts as results. The inner dict contains the "precision", "recall", and "hmean" scores under different prediction score thresholds, which can be indexed by the corresponding threshold value from the outer dict. The outer dict also contains the "best" key, which is the result of the best hmean score.

Return type dict[float or "best", dict[str, float]]

15.1.12 EndPointError

class mmeval.metrics.**EndPointError**(***kwargs*)

EndPointError evaluation metric.

EndPointError is a widely used evaluation metric for optical flow estimation.

This metric supports 3 kinds of inputs, i.e. *numpy.ndarray* and *torch.Tensor*, *oneflow.Tensor*, and the implementation for the calculation depends on the inputs type.

Parameters ****kwargs** – Keyword arguments passed to `BaseMetric`.

Examples

```
>>> from mmeval import EndPointError
>>> epe = EndPointError()
```

Use NumPy implementation:

```
>>> import numpy as np
>>> predictions = np.array(
...     [[[10., 5.], [0.1, 3.]],
...     [[3., 15.2], [2.4, 4.5]]])
>>> labels = np.array(
...     [[[10.1, 4.8], [10, 3.]],
...     [[6., 10.2], [2.0, 4.1]]])
>>> valid_masks = np.array([[1., 1.], [1., 0.]])
>>> epe(predictions, labels, valid_masks)
{'EPE': 5.318186230865093}
```

Use PyTorch implementation:

```
>>> import torch
>>> predictions = torch.Tensor(
...     [[[10., 5.], [0.1, 3.]],
...     [[3., 15.2], [2.4, 4.5]]])
>>> labels = torch.Tensor(
...     [[[10.1, 4.8], [10, 3.]],
```

(continues on next page)

(continued from previous page)

```
...     [[6., 10.2], [2.0, 4.1]]])
>>> valid_masks = torch.Tensor([[1., 1.], [1., 0.]])
>>> epe(predictions, labels, valid_masks)
{'EPE': 5.3181863}
```

Accumulate batch:

```
>>> for i in range(10):
...     predictions = torch.randn(10, 10, 2)
...     labels = torch.randn(10, 10, 2)
...     epe.add(predictions, labels)
>>> epe.compute()
```

add(*predictions: Sequence*, *labels: Sequence*, *valid_masks: Optional[Sequence] = None*) → None
Process one batch of predictions and labels.

Parameters

- **predictions** (*Sequence*) – Predicted sequence of flow map with shape (H, W, 2).
- **labels** (*Sequence*) – The ground truth sequence of flow map with shape (H, W, 2).
- **valid_masks** (*Sequence*) – The Sequence of valid mask for labels with shape (H, W). If it is None, this function will automatically generate a map filled with 1. Defaults to None.

compute_metric(*results: List[Tuple[numpy.ndarray, int]]*) → dict
Compute the EndPointError metric.

This method would be invoked in *BaseMetric.compute* after distributed synchronization.

Parameters **results** (*List[np.ndarray]*) – This list has already been synced across all ranks. This is a list of *np.ndarray*, which is the end point error map between the prediction and the label.

Returns

The computed metric, with following key:

- EPE, the mean end point error of all pairs.

Return type Dict

end_point_error_map
Calculate end point error map.

Parameters

- **prediction** (*np.ndarray*) – Prediction with shape (H, W, 2).
- **label** (*np.ndarray*) – Ground truth with shape (H, W, 2).
- **valid_mask** (*np.ndarray, optional*) – Valid mask with shape (H, W).

Returns The mean of end point error and the numbers of valid labels.

Return type Tuple

15.1.13 PCKAccuracy

class mmeval.metrics.**PCKAccuracy**(*thr: float = 0.2*, *norm_item: Union[str, Sequence[str]] = 'bbox'*, ***kwargs*)

PCK accuracy evaluation metric, which is widely used in pose estimation.

Calculate the pose accuracy of Percentage of Correct Keypoints (PCK) for each individual keypoint and the averaged accuracy across all keypoints. PCK metric measures the accuracy of the localization of the body joints. The distances between predicted positions and the ground-truth ones are typically normalized by the person bounding box size. The threshold (thr) of the normalized distance is commonly set as 0.05, 0.1 or 0.2 etc.

Note:

- length of dataset: N
- num_keypoints: K
- number of keypoint dimensions: D (typically D = 2)

Parameters

- **thr** (*float*) – Threshold of PCK calculation. Defaults to 0.2.
- **norm_item** (*str | Sequence[str]*) – The item used for normalization. Valid items include 'bbox', 'head', 'torso', which correspond to 'PCK', 'PCKh' and 'tPCK' respectively. Defaults to `'bbox'`.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import PCKAccuracy
>>> import numpy as np
>>> num_keypoints = 15
>>> keypoints = np.random.random((1, num_keypoints, 2)) * 10
>>> predictions = [{'coords': keypoints}]
>>> keypoints_visible = np.ones((1, num_keypoints)).astype(bool)
>>> bbox_size = np.random.random((1, 2)) * 10
>>> groundtruths = [{
...     'coords': keypoints,
...     'mask': keypoints_visible,
...     'bbox_size': bbox_size,
... }]
>>> pck_metric = PCKAccuracy(thr=0.5, norm_item='bbox')
>>> pck_metric(predictions, groundtruths)
OrderedDict([('PCK@0.5', 1.0)])
```

add(*predictions: List[Dict]*, *groundtruths: List[Dict]*) → None

Process one batch of predictions and groundtruths and add the intermediate results to *self._results*.

Parameters

- **predictions** (*Sequence[dict]*) – Predictions from the model. Each prediction dict has the following keys:
  - coords (np.ndarray, [1, K, D]): predicted keypoints coordinates

- **groundtruths** (*Sequence[dict]*) – The ground truth labels. Each groundtruth dict has the following keys:
  - coords (np.ndarray, [1, K, D]): ground truth keypoints coordinates
  - mask (np.ndarray, [1, K]): ground truth keypoints_visible
  - bbox_size (np.ndarray, optional, [1, 2]): ground truth bbox size
  - head_size (np.ndarray, optional, [1, 2]): ground truth head size
  - torso_size (np.ndarray, optional, [1, 2]): ground truth torso size

compute_metric(*results: list*) → Dict[str, float]

Compute the metrics from processed results.

Parameters **results** (*list*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are the corresponding results.

Return type Dict[str, float]

15.1.14 MpiiPCKAccuracy

class mmeval.metrics.**MpiiPCKAccuracy**(*thr: float = 0.5*, *norm_item: Union[str, Sequence[str]] = 'head'*, ***kwargs*)

PCKh accuracy evaluation metric for MPII dataset.

Calculate the pose accuracy of Percentage of Correct Keypoints (PCK) for each individual keypoint and the averaged accuracy across all keypoints. PCK metric measures accuracy of the localization of the body joints. The distances between predicted positions and the ground-truth ones are typically normalized by the person bounding box size. The threshold (thr) of the normalized distance is commonly set as 0.05, 0.1 or 0.2 etc.

Note:

- length of dataset: N
- num_keypoints: K
- number of keypoint dimensions: D (typically D = 2)

Parameters

- **thr** (*float*) – Threshold of PCK calculation. Defaults to 0.5.
- **norm_item** (*str | Sequence[str]*) – The item used for normalization. Valid items include 'bbox', 'head', 'torso', which correspond to 'PCK', 'PCKh' and 'tPCK' respectively. Defaults to `'head'`.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import MpiiPCKAccuracy
>>> import numpy as np
>>> num_keypoints = 16
>>> keypoints = np.random.random((1, num_keypoints, 2)) * 10
>>> predictions = [{'coords': keypoints}]
>>> keypoints_visible = np.ones((1, num_keypoints)).astype(bool)
>>> head_size = np.random.random((1, 2)) * 10
>>> groundtruths = [{
...     'coords': keypoints + 1.0,
...     'mask': keypoints_visible,
...     'head_size': head_size,
... }]
>>> mpii_pckh_metric = MpiiPCKAccuracy(thr=0.3, norm_item='head')
>>> mpii_pckh_metric(predictions, groundtruths)
OrderedDict([('Head', 100.0), ('Shoulder', 100.0), ('Elbow', 100.0),
('Wrist', 100.0), ('Hip', 100.0), ('Knee', 100.0), ('Ankle', 100.0),
('PCKh', 100.0), ('PCKh@0.1', 100.0)])
```

compute_metric(*results: list*) → Dict[str, float]

Compute the metrics from processed results.

Parameters **results** (*list*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict[str, float]

15.1.15 JhmdbPCKAccuracy

class mmeval.metrics.**JhmdbPCKAccuracy**(*thr: float = 0.5*, *norm_item: Union[str, Sequence[str]] = 'bbox'*, ***kwargs*)

PCK accuracy evaluation metric for Jhmdb dataset.

Calculate the pose accuracy of Percentage of Correct Keypoints (PCK) for each individual keypoint and the averaged accuracy across all keypoints. PCK metric measures accuracy of the localization of the body joints. The distances between predicted positions and the ground-truth ones are typically normalized by the person bounding box size. The threshold (thr) of the normalized distance is commonly set as 0.05, 0.1 or 0.2 etc.

Note:

- length of dataset: N
- num_keypoints: K
- number of keypoint dimensions: D (typically D = 2)

Parameters

- **thr** (*float*) – Threshold of PCK calculation. Defaults to 0.5.
- **norm_item** (*str | Sequence[str]*) – The item used for normalization. Valid items include 'bbox', 'head', 'torso', which correspond to 'PCK', 'PCKh' and 'tPCK' respectively. Defaults to `'bbox'`.

- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import JhmdbPCKAccuracy
>>> import numpy as np
>>> num_keypoints = 15
>>> keypoints = np.random.random((1, num_keypoints, 2)) * 10
>>> predictions = [{'coords': keypoints}]
>>> keypoints_visible = np.ones((1, num_keypoints)).astype(bool)
>>> torso_size = np.random.random((1, 2)) * 10
>>> groundtruths = [{
...     'coords': keypoints,
...     'mask': keypoints_visible,
...     'torso_size': torso_size,
... }]
>>> jhmdb_pckh_metric = JhmdbPCKAccuracy(thr=0.2, norm_item='torso')
>>> jhmdb_pckh_metric(predictions, groundtruths)
OrderedDict([('Head tPCK', 1.0), ('Sho tPCK', 1.0), ('Elb tPCK', 1.0),
('Wri tPCK', 1.0), ('Hip tPCK', 1.0), ('Knee tPCK', 1.0),
('Ank tPCK', 1.0), ('Mean tPCK', 1.0)])
```

compute_metric(*results: list*) → Dict[str, float]

Compute the metrics from processed results.

Parameters **results** (*list*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict[str, float]

15.1.16 AVAMeanAP

class mmeval.metrics.**AVAMeanAP**(*ann_file: str*, *label_file: str*, *exclude_file: Optional[str] = None*, *num_classes: int = 81*, *custom_classes: Optional[List[int]] = None*, *verbose: bool = True*, ***kwargs*)

AVA evaluation metric.

AVA(Atomic Visual Action): https://research.google.com/ava.

This metric computes mAP using the ava evaluation toolkit provided by the author.

Parameters

- **ann_file** (*str*) – The annotation file path.
- **label_file** (*str*) – The label file path.
- **exclude_file** (*str, optional*) – The excluded timestamp file path. Defaults to None.
- **num_classes** (*int*) – Number of classes. Defaults to 81.
- **custom_classes** (*list(int), optional*) – A subset of class ids from origin dataset. Defaults to None.
- **verbose** (*bool*) – Whether to print messages in the evaluation process. Defaults to True.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import AVAMeanAP
>>> import numpy as np
>>>
>>> ann_file = 'tests/test_metrics/ava_detection_gt.csv'
>>> label_file = 'tests/test_metrics/ava_action_list.txt'
>>> num_classes = 4
>>> ava_metric = AVAMeanAP(ann_file=ann_file, label_file=label_file,
>>>                        num_classes=4)
>>>
>>> predictions = [
>>> {
>>>     'video_id': '3reY9zJKhqN',
>>>     'timestamp': 1774,
>>>     'outputs': [
>>>         np.array([[0.362, 0.156, 0.969, 0.666, 0.106],
>>>                   [0.442, 0.083, 0.721, 0.947, 0.162]]),
>>>         np.array([[0.288, 0.365, 0.766, 0.551, 0.706],
>>>                   [0.178, 0.296, 0.707, 0.995, 0.223]]),
>>>         np.array([[0.417, 0.167, 0.843, 0.939, 0.015],
>>>                   [0.35, 0.421, 0.57, 0.689, 0.427]]) ]
>>> },
>>> {
>>>     'video_id': 'HmR8SmNIoxu',
>>>     'timestamp': 1384,
>>>     'outputs': [
>>>         np.array([[0.256, 0.338, 0.726, 0.799, 0.563],
>>>                   [0.071, 0.256, 0.64, 0.75, 0.297]]),
>>>         np.array([[0.326, 0.036, 0.513, 0.991, 0.405],
>>>                   [0.351, 0.035, 0.729, 0.936, 0.945]]),
>>>         np.array([[0.051, 0.005, 0.975, 0.942, 0.424],
>>>                   [0.347, 0.05, 0.97, 0.944, 0.396]])]
>>> },
>>> {
>>>     'video_id': '5HNXoce1raG',
>>>     'timestamp': 1097,
>>>     'outputs': [
>>>         np.array([[0.39, 0.087, 0.833, 0.616, 0.447],
>>>                   [0.461, 0.212, 0.627, 0.527, 0.036]]),
>>>         np.array([[0.022, 0.394, 0.93, 0.527, 0.109],
>>>                   [0.208, 0.462, 0.874, 0.948, 0.954]]),
>>>         np.array([[0.206, 0.456, 0.564, 0.725, 0.685],
>>>                   [0.106, 0.445, 0.782, 0.673, 0.367]])]}
>>> ]
>>> ava_metric(predictions)
{'mAP@0.5IOU': 0.027777778}
```

add(*predictions: Sequence[dict]*) → None
Add detection results to the results list.

Parameters **predictions** (*Sequence[dict]*) – A list of prediction dict which contains the following keys:

- *video_id*: The id of the video, e.g., *3reY9zJKhqN*.

- *timestamp*: The timestamp of the video e.g., *1774*.
- *outputs*: A list bbox results of each class with the format of [x1, y1, x2, y2, score].

ava_eval(*result_file: str*) → dict
Perform ava evaluation.

Parameters **result_file** (*str*) – The dumped results file path.

Returns The evaluation results.

Return type dict

compute_metric(*results: list*) → dict
Compute the AVA MeanAP.

Parameters **results** (*list*) – A list of detection results.

Returns The computed ava metric.

Return type dict

read_csv(*csv_file: str*, *class_whitelist: Optional[set] = None*) → tuple
Loads boxes and class labels from a CSV file in the AVA format.

CSV file format described at https://research.google.com/ava/download.html.

Parameters

- **csv_file** (*str*) – A csv file path.
- **class_whitelist** (*set, optional*) – If provided, boxes corresponding to (integer) class labels not in this set are skipped.

Returns

- boxes (dict): A dictionary mapping each unique image key (string) to a list of boxes, given as coordinates [y1, x1, y2, x2].
- labels (dict): A dictionary mapping each unique image key (string) to a list of integer class labels, matching the corresponding box in *boxes*.
- scores (dict): A dictionary mapping each unique image key (string) to a list of score values labels, matching the corresponding label in *labels*. If scores are not provided in the csv, then they will default to 1.0.

Return type tuple (boxes, labels, scores)

read_exclusions(*exclude_file: str*) → set
Reads a CSV file of excluded timestamps.

Parameters **exclude_file** (*str*) – The path of exclude file.

Returns A set of strings containing excluded image keys, e.g. "aaaaaaaaaaa,0904" or an empty set if exclusions file is None.

Return type excluded (set)

read_label(*label_file: str*) → tuple
Reads a label mapping file.

Parameters **label_file** (*str*) – The path of label file.

Returns

- labelmap (list): The label map in the form used by the object_detection_evaluation module - a list of {"id": integer, "name": classname } dicts.

- class_ids (set): A set containing all of the valid class id integers.

Return type tuple (labelmap, class_ids)

results2csv(*results: List[dict]*, *out_file: str*) → None
Dump the results to a csv file.

Parameters

- **results** (*list[dict]*) – A list of detection results.
- **out_file** (*str*) – The output csv file path.

15.1.17 StructuralSimilarity

class mmeval.metrics.**StructuralSimilarity**(*crop_border: int = 0*, *input_order: str = 'CHW'*, *convert_to: Optional[str] = None*, *channel_order: str = 'rgb'*, ***kwargs*)
Calculate StructuralSimilarity (structural similarity).

Ref: Image quality assessment: From error visibility to structural similarity

The results are the same as that of the official released MATLAB code in https://ece.uwaterloo.ca/~z70wang/research/ssim/.

For three-channel images, StructuralSimilarity is calculated for each channel and then averaged.

Parameters

- **crop_border** (*int*) – Cropped pixels in each edges of an image. These pixels are not involved in the PeakSignalNoiseRatio calculation. Defaults to 0.
- **input_order** (*str*) – Whether the input order is 'HWC' or 'CHW'. Defaults to 'HWC'.
- **convert_to** (*str, optional*) – Whether to convert the images to other color models. If None, the images are not altered. When computing for 'Y', the images are assumed to be in BGR order. Options are 'Y' and None. Defaults to None.
- **channel_order** (*str*) – The channel order of image. Choices are 'rgb' and 'bgr'. Defaults to 'rgb'.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import StructuralSimilarity as SSIM
>>> import numpy as np
>>>
>>> ssim = SSIM(input_order='CHW', convert_to='Y', channel_order='rgb')
>>> gts = np.random.randint(0, 255, size=(3, 32, 32))
>>> preds = np.random.randint(0, 255, size=(3, 32, 32))
>>> ssim(preds, gts)
{'ssim': ...}
```

Calculate StructuralSimilarity between 2 single channel images:

```
>>> img1 = np.ones((32, 32)) * 2
>>> img2 = np.ones((32, 32))
>>> SSIM.compute_ssim(img1, img2)
0.913062377743969
```

add(*predictions: Sequence[numpy.ndarray]*, *groundtruths: Sequence[numpy.ndarray]*, *channel_order: Optional[str] = None*) → None
Add the StructuralSimilarity score of the batch to `self._results`.

For three-channel images, StructuralSimilarity is calculated for each channel and then averaged.

Parameters

- **predictions** (*Sequence[np.ndarray]*) – Predictions of the model.
- **groundtruths** (*Sequence[np.ndarray]*) – The ground truth images.
- **channel_order** (*Optional[str]*) – The channel order of the input samples. If not passed, will set as `self.channel_order`. Defaults to None.

compute_metric(*results: List[numpy.float64]*) → Dict[str, float]
Compute the StructuralSimilarity metric.

This method would be invoked in `BaseMetric.compute` after distributed synchronization.

Parameters **results** (*List[np.float64]*) – A list that consisting the StructuralSimilarity scores. This list has already been synced across all ranks.

Returns The computed StructuralSimilarity metric.

Return type Dict[str, float]

static compute_ssim(*img1: numpy.ndarray*, *img2: numpy.ndarray*) → numpy.float64
Calculate StructuralSimilarity (structural similarity) between two single channel image.

Ref: Image quality assessment: From error visibility to structural similarity

The results are the same as that of the official released MATLAB code in https://ece.uwaterloo.ca/~z70wang/research/ssim/.

Parameters

- **img1** (*np.ndarray*) – Single channels Images with range [0, 255].
- **img2** (*np.ndarray*) – Single channels Images with range [0, 255].

Returns StructuralSimilarity result.

Return type np.float64

15.1.18 SignalNoiseRatio

class mmeval.metrics.**SignalNoiseRatio**(*crop_border: int = 0*, *input_order: str = 'CHW'*, *convert_to: Optional[str] = None*, *channel_order: str = 'rgb'*, ***kwargs*)
Signal-to-Noise Ratio.

Ref: https://en.wikipedia.org/wiki/Signal-to-noise_ratio

Parameters

- **crop_border** (*int*) – Cropped pixels in each edges of an image. These pixels are not involved in the PeakSignalNoiseRatio calculation. Defaults to 0.
- **input_order** (*str*) – Whether the input order is 'HWC' or 'CHW'. Defaults to 'HWC'.
- **convert_to** (*str, optional*) – Whether to convert the images to other color models. If None, the images are not altered. When computing for 'Y', the images are assumed to be in BGR order. Options are 'Y' and None. Defaults to None.

- **channel_order** (*str*) – The channel order of image. Choices are 'rgb' and 'bgr'. Defaults to 'rgb'.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import SignalNoiseRatio
>>> import numpy as np
>>>
>>> snr = SignalNoiseRatio(crop_border=1, input_order='CHW',
...                        convert_to='Y', channel_order='rgb')
>>> gts = np.random.randint(0, 255, size=(3, 32, 32))
>>> preds = np.random.randint(0, 255, size=(3, 32, 32))
>>> snr(preds, gts)
{'snr': ...}
```

Calculate SignalNoiseRatio between 2 images:

```
>>> gts = np.ones((3, 32, 32)) * 2
>>> preds = np.ones((3, 32, 32))
>>> SignalNoiseRatio.compute_snr(preds, gts)
6.020599913279624
```

add(*predictions: Sequence[numpy.ndarray]*, *groundtruths: Sequence[numpy.ndarray]*, *channel_order: Optional[str] = None*) → None

Add SignalNoiseRatio score of batch to `self._results`

Parameters

- **predictions** (*Sequence[np.ndarray]*) – Predictions of the model.
- **groundtruths** (*Sequence[np.ndarray]*) – The ground truth images.
- **channel_order** (*Optional[str]*) – The channel order of the input samples. If not passed, will set as `self.channel_order`. Defaults to None.

compute_metric(*results: List[numpy.float64]*) → Dict[str, float]

Compute the SignalNoiseRatio metric.

This method would be invoked in `BaseMetric.compute` after distributed synchronization.

Parameters **results** (*List[np.float64]*) – A list that consisting the SignalNoiseRatio score. This list has already been synced across all ranks.

Returns The computed SignalNoiseRatio metric.

Return type Dict[str, float]

static compute_snr(*prediction: numpy.ndarray*, *groundtruth: numpy.ndarray*) → numpy.float64

Calculate SignalNoiseRatio (Signal-to-Noise Ratio).

Ref: https://en.wikipedia.org/wiki/Signal-to-noise_ratio

Parameters

- **prediction** (*np.ndarray*) – Images with range [0, 255].
- **groundtruth** (*np.ndarray*) – Images with range [0, 255].

Returns SignalNoiseRatio result.

Return type np.float64

15.1.19 PeakSignalNoiseRatio

class mmeval.metrics.**PeakSignalNoiseRatio**(*crop_border: int = 0*, *input_order: str = 'CHW'*, *convert_to: Optional[str] = None*, *channel_order: str = 'rgb'*, ***kwargs*)

Peak Signal-to-Noise Ratio.

Ref: https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

Parameters

- **crop_border** (*int*) – Cropped pixels in each edges of an image. These pixels are not involved in the PeakSignalNoiseRatio calculation. Defaults to 0.
- **input_order** (*str*) – Whether the input order is 'HWC' or 'CHW'. Defaults to 'CHW'.
- **convert_to** (*str*) – Whether to convert the images to other color models. If None, the images are not altered. When computing for 'Y', the images are assumed to be in BGR order. Options are 'Y' and None. Defaults to None.
- **channel_order** (*str*) – The channel order of image. Defaults to 'rgb'.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import PeakSignalNoiseRatio as PSNR
>>> import numpy as np
>>>
>>> psnr = PSNR(input_order='CHW', convert_to='Y', channel_order='rgb')
>>> gts = np.random.randint(0, 255, size=(3, 32, 32))
>>> preds = np.random.randint(0, 255, size=(3, 32, 32))
>>> psnr(preds, gts)
{'psnr': ...}
```

Calculate PeakSignalNoiseRatio between 2 single channel images:

```
>>> img1 = np.ones((32, 32))
>>> img2 = np.ones((32, 32))
>>> PSNR.compute_psnr(img1, img2)
49.45272242415597
```

add(*predictions: Sequence[numpy.ndarray]*, *groundtruths: Sequence[numpy.ndarray]*, *channel_order: Optional[str] = None*) → None

Add PeakSignalNoiseRatio score of batch to `self._results`

Parameters

- **predictions** (*Sequence[np.ndarray]*) – Predictions of the model.
- **groundtruths** (*Sequence[np.ndarray]*) – The ground truth images.
- **channel_order** (*Optional[str]*) – The channel order of the input samples. If not passed, will set as `self.channel_order`. Defaults to None.

compute_metric(*results: List[numpy.float64]*) → Dict[str, float]

Compute the PeakSignalNoiseRatio metric.

This method would be invoked in `BaseMetric.compute` after distributed synchronization.

Parameters **results** (*List[np.float64]*) – A list that consisting the PeakSignalNoiseRatio score. This list has already been synced across all ranks.

Returns The computed PeakSignalNoiseRatio metric.

Return type Dict[str, float]

static compute_psnr(*prediction: numpy.ndarray*, *groundtruth: numpy.ndarray*) → numpy.float64

Calculate PeakSignalNoiseRatio (Peak Signal-to-Noise Ratio).

Ref: https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

Parameters

- **prediction** (*np.ndarray*) – Images with range [0, 255].
- **groundtruth** (*np.ndarray*) – Images with range [0, 255].

Returns PeakSignalNoiseRatio result.

Return type np.float64

15.1.20 MeanAbsoluteError

class mmeval.metrics.**MeanAbsoluteError**(***kwargs*)

Mean Absolute Error metric for image.

Formula: mean(abs(a-b)).

Parameters ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import MeanAbsoluteError as MAE
>>> import numpy as np
>>>
>>> mae = MAE()
>>> gts = np.random.randint(0, 255, size=(3, 32, 32))
>>> preds = np.random.randint(0, 255, size=(3, 32, 32))
>>> mae(preds, gts)
{'mae': ...}
```

Calculate MeanAbsoluteError between 2 images with mask:

```
>>> img1 = np.ones((32, 32, 3))
>>> img2 = np.ones((32, 32, 3)) * 2
>>> mask = np.ones((32, 32, 3)) * 2
>>> mask[:16] *= 0
>>> MAE.compute_mae(img1, img2, mask)
0.003921568627
```

add(*predictions: Sequence[numpy.ndarray]*, *groundtruths: Sequence[numpy.ndarray]*, *masks: Optional[Sequence[numpy.ndarray]] = None*) → None

Add MeanAbsoluteError score of batch to `self._results`

Parameters

- **predictions** (*Sequence[np.ndarray]*) – Predictions of the model.
- **groundtruths** (*Sequence[np.ndarray]*) – The ground truth images.
- **masks** (*Sequence[np.ndarray], optional*) – Mask images. Defaults to None.

static compute_mae(*prediction: numpy.ndarray*, *groundtruth: numpy.ndarray*, *mask: Optional[numpy.ndarray] = None*) → numpy.float32

Calculate MeanAbsoluteError (Mean Absolute Error).

Parameters

- **prediction** (*np.ndarray*) – Images with range [0, 255].
- **groundtruth** (*np.ndarray*) – Images with range [0, 255].
- **mask** (*np.ndarray, optional*) – Mask of evaluation.

Returns MeanAbsoluteError result.

Return type np.float32

compute_metric(*results: List[numpy.float32]*) → Dict[str, float]

Compute the MeanAbsoluteError metric.

This method would be invoked in `BaseMetric.compute` after distributed synchronization.

Parameters **results** (*List[np.float32]*) – A list that consisting the MeanAbsoluteError score. This list has already been synced across all ranks.

Returns The computed MeanAbsoluteError metric.

Return type Dict[str, float]

15.1.21 MeanSquaredError

class mmeval.metrics.**MeanSquaredError**(***kwargs*)

Mean Squared Error metric for image.

Formula: mean((a-b)^2).

Parameters ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import MeanSquaredError as MSE
>>> import numpy as np
>>>
>>> mse = MSE()
>>> gts = np.random.randint(0, 255, size=(3, 32, 32))
>>> preds = np.random.randint(0, 255, size=(3, 32, 32))
>>> mse(preds, gts)
{'mse': ...}
```

Calculate MeanSquaredError between 2 images with mask:

```
>>> img1 = np.ones((32, 32, 3))
>>> img2 = np.ones((32, 32, 3)) * 2
>>> mask = np.ones((32, 32, 3)) * 2
```

(continues on next page)

(continued from previous page)

```
>>> mask[:16] *= 0
>>> MSE.compute_mse(img1, img2, mask)
0.000015378700496
```

add(*predictions: Sequence[numpy.ndarray]*, *groundtruths: Sequence[numpy.ndarray]*, *masks: Optional[Sequence[numpy.ndarray]] = None*) → None

Add MeanSquaredError score of batch to `self._results`

Parameters

- **predictions** (*Sequence[np.ndarray]*) – Predictions of the model.
- **groundtruths** (*Sequence[np.ndarray]*) – The ground truth images.
- **masks** (*Sequence[np.ndarray], optional*) – Mask images.

compute_metric(*results: List[numpy.float32]*) → Dict[str, float]

Compute the MeanSquaredError metric.

This method would be invoked in `BaseMetric.compute` after distributed synchronization.

Parameters **results** (*List[np.float32]*) – A list that consisting the MeanSquaredError score. This list has already been synced across all ranks.

Returns The computed MeanSquaredError metric.

Return type Dict[str, float]

static compute_mse(*prediction: numpy.ndarray*, *groundtruth: numpy.ndarray*, *mask: Optional[numpy.ndarray] = None*) → numpy.float32

Calculate MeanSquaredError (Mean Squared Error).

Parameters

- **prediction** (*np.ndarray*) – Images with range [0, 255].
- **groundtruth** (*np.ndarray*) – Images with range [0, 255].
- **mask** (*np.ndarray, optional*) – Mask of evaluation.

Returns MeanSquaredError result.

Return type np.float32

15.1.22 BLEU

class mmeval.metrics.**BLEU**(*n_gram: int = 4*, *smooth: bool = False*, *ngram_weights: Optional[Sequence[float]] = None*, *tokenizer_fn: Optional[Union[Callable, str]] = None*, ***kwargs*)

Bilingual Evaluation Understudy metric.

This metric proposed in BLEU: a Method for Automatic Evaluation of Machine Translation is a tool for evaluating the quality of machine translation. The closer the translation is to human translation, the higher the score will be.

Parameters

- **n_gram** (*int*) – The maximum number of words contained in a phrase when calculating word fragments. Defaults to 4.
- **smooth** (*bool*) – Whether or not to apply to smooth. Defaults to False.

- **ngram_weights** (*Sequence[float], optional*) – Weights used for unigrams, bigrams, etc. to calculate BLEU score. If not provided, uniform weights are used. Defaults to None.
- **tokenizer_fn** (*Union[Callable, str, None]*) – A user's own tokenizer function. Defaults to None. New in version 0.3.0.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import BLEU
>>> predictions = ['the cat is on the mat', 'There is a big tree near the park here
↪']  # noqa: E501
>>> references = [['a cat is on the mat'], ['A big tree is growing near the park␣
↪here']]  # noqa: E501
>>> bleu = BLEU()
>>> bleu_results = bleu(predictions, references)
{'bleu': 0.5226045319355426}
```

```
>>> # Calculate BLEU with smooth:
>>> from mmeval import BLEU
>>> predictions = ['the cat is on the mat', 'There is a big tree near the park here
↪']  # noqa: E501
>>> references = [['a cat is on the mat'], ['A big tree is growing near the park␣
↪here']]  # noqa: E501
>>> bleu = BLEU(smooth = True)
>>> bleu_results = bleu(predictions, references)
{'bleu': 0.566315716093867}
```

add(*predictions: Sequence[str], references: Sequence[Sequence[str]]*) → None

Add the intermediate results to `self._results`.

Parameters

- **predictions** (*Sequence[str]*) – An iterable of predicted sentences.
- **references** (*Sequence[Sequence[str]*) – An iterable of referenced sentences.

compute_metric(*results: List[Tuple[int, int, numpy.ndarray, numpy.ndarray]]*) → dict

Compute the bleu metric.

This method would be invoked in `BaseMetric.compute` after distributed synchronization.

Parameters **results** (*List[Tuple[int, int, np.ndarray, np.ndarray]]*) – A list that consisting the tuple of correct numbers. Tuple contains pred_len, references_len, precision_matches, precision_total. This list has already been synced across all ranks.

Returns The computed bleu score.

Return type Dict[str, float]

15.1.23 SumAbsoluteDifferences

class mmeval.metrics.**SumAbsoluteDifferences**(*norm_const: int = 1000*, ***kwargs*)
Sum of Absolute Differences metric for image.

This metric computes per-pixel absolute difference and sum across all pixels. i.e. sum(abs(a-b)) / norm_const

Parameters

- **norm_const** (*int*) – Divide the result to reduce its magnitude. Default to 1000.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Note: The current implementation assumes the image a numpy array with pixel values ranging from 0 to 255.

Examples

```
>>> from mmeval import SumAbsoluteDifferences as SAD
>>> import numpy as np
>>>
>>> sad = SAD()
>>> prediction = np.zeros((32, 32), dtype=np.uint8)
>>> groundtruth = np.ones((32, 32), dtype=np.uint8) * 255
>>> sad(prediction, groundtruth)
{'sad': ...}
```

add(*predictions: Sequence[numpy.ndarray]*, *groundtruths: Sequence[numpy.ndarray]*) → None
Add SumAbsoluteDifferences score of batch to `self._results`

Parameters

- **predictions** (*Sequence[np.ndarray]*) – Sequence of predicted image.
- **groundtruths** (*Sequence[np.ndarray]*) – Sequence of groundtruth image.

compute_metric(*results: List*) → Dict[str, float]
Compute the SumAbsoluteDifferences metric.

Parameters **results** (*List*) – A list that consisting the SumAbsoluteDifferences score. This list has already been synced across all ranks.

Returns The computed SumAbsoluteDifferences metric. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict[str, float]

15.1.24 GradientError

class mmeval.metrics.**GradientError**(*sigma: float = 1.4*, *norm_const: int = 1000*, ***kwargs*)
Gradient error for evaluating alpha matte prediction.

Parameters

- **sigma** (*float*) – Standard deviation of the gaussian kernel. Defaults to 1.4 .
- **norm_const** (*int*) – Divide the result to reduce its magnitude. Defaults to 1000.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Note: The current implementation assumes the image / alpha / trimap a numpy array with pixel values ranging from 0 to 255.

The pred_alpha should be masked by trimap before passing into this metric.

The trimap is the most commonly used prior knowledge. As the name implies, trimap is a ternary graph and each pixel takes one of {0, 128, 255}, representing the foreground, the unknown and the background respectively.

Examples

```
>>> from mmeval import GradientError
>>> import numpy as np
>>>
>>> gradient_error = GradientError()
>>> np.random.seed(0)
>>> pred_alpha = np.random.randn(32, 32).astype('uint8')
>>> gt_alpha = np.ones((32, 32), dtype=np.uint8) * 255
>>> trimap = np.zeros((32, 32), dtype=np.uint8)
>>> trimap[:16, :16] = 128
>>> trimap[16:, 16:] = 255
>>> gradient_error(pred_alpha, gt_alpha, trimap)
{'gradient_error': ...}
```

add(*pred_alphas: Sequence[numpy.ndarray]*, *gt_alphas: Sequence[numpy.ndarray]*, *trimaps: Sequence[numpy.ndarray]*) → None

Add GradientError score of batch to `self._results`

Parameters

- **pred_alphas** (*Sequence[np.ndarray]*) – Predict the probability that pixels belong to the foreground.
- **gt_alphas** (*Sequence[np.ndarray]*) – Probability that the actual pixel belongs to the foreground.
- **trimaps** (*Sequence[np.ndarray]*) – Broadly speaking, the trimap consists of foreground and unknown region.

compute_metric(*results: List*) → Dict[str, float]

Compute the GradientError metric.

Parameters **results** (*List*) – A list that consisting the GradientError score. This list has already been synced across all ranks.

Returns The computed GradientError metric. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict[str, float]

15.1.25 MattingMeanSquaredError

class mmeval.metrics.**MattingMeanSquaredError**(***kwargs*)

Mean Squared Error metric for image matting.

This metric computes the per-pixel squared error average across all pixels. i.e. mean((a-b)^2)

Parameters ****kwargs** – Keyword parameters passed to `BaseMetric`.

Note: The current implementation assumes the image / alpha / trimap a numpy array with pixel values ranging from 0 to 255.

The pred_alpha should be masked by trimap before passing into this metric.

The trimap is the most commonly used prior knowledge. As the name implies, trimap is a ternary graph and each pixel takes one of {0, 128, 255}, representing the foreground, the unknown and the background respectively.

Examples

```
>>> from mmeval import MattingMeanSquaredError as MattingMSE
>>> import numpy as np
>>>
>>> matting_mse = MattingMSE()
>>> pred_alpha = np.zeros((32, 32), dtype=np.uint8)
>>> gt_alpha = np.ones((32, 32), dtype=np.uint8) * 255
>>> trimap = np.zeros((32, 32), dtype=np.uint8)
>>> trimap[:16, :16] = 128
>>> trimap[16:, 16:] = 255
>>> matting_mse(pred_alpha, gt_alpha, trimap)
{'matting_mse': ...}
```

add(*pred_alphas: Sequence[numpy.ndarray]*, *gt_alphas: Sequence[numpy.ndarray]*, *trimaps: Sequence[numpy.ndarray]*) → None

Add MattingMeanSquaredError score of batch to `self._results`

Parameters

- **pred_alphas** (*Sequence[np.ndarray]*) – Predict the probability that pixels belong to the foreground.
- **gt_alphas** (*Sequence[np.ndarray]*) – Probability that the actual pixel belongs to the foreground.
- **trimaps** (*Sequence[np.ndarray]*) – Broadly speaking, the trimap consists of foreground and unknown region.

compute_metric(*results: List*) → Dict[str, float]

Compute the MattingMeanSquaredError metric.

Parameters **results** (*List*) – A list that consisting the MattingMeanSquaredError score. This list has already been synced across all ranks.

Returns The computed MattingMeanSquaredError metric. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict[str, float]

15.1.26 ConnectivityError

class mmeval.metrics.**ConnectivityError**(*step: float = 0.1*, *norm_const: int = 1000*, ***kwargs*)

Connectivity error for evaluating alpha matte prediction.

Parameters

- **step** (*float*) – Step of threshold when computing intersection between *alpha* and *pred_alpha*. Default to 0.1 .
- **norm_const** (*int*) – Divide the result to reduce its magnitude. Defaults to 1000 .
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Note: The current implementation assumes the image / alpha / trimap a numpy array with pixel values ranging from 0 to 255.

The pred_alpha should be masked by trimap before passing into this metric.

The trimap is the most commonly used prior knowledge. As the name implies, trimap is a ternary graph and each pixel takes one of {0, 128, 255}, representing the foreground, the unknown and the background respectively.

Examples

```
>>> from mmeval import ConnectivityError
>>> import numpy as np
>>>
>>> connectivity_error = ConnectivityError()
>>> pred_alpha = np.zeros((32, 32), dtype=np.uint8)
>>> gt_alpha = np.ones((32, 32), dtype=np.uint8) * 255
>>> trimap = np.zeros((32, 32), dtype=np.uint8)
>>> trimap[:16, :16] = 128
>>> trimap[16:, 16:] = 255
>>> connectivity_error(pred_alpha, gt_alpha, trimap)
{'connectivity_error': ...}
```

add(*pred_alphas: Sequence[numpy.ndarray]*, *gt_alphas: Sequence[numpy.ndarray]*, *trimaps: Sequence[numpy.ndarray]*) → None

Add ConnectivityError score of batch to `self._results`

Parameters

- **pred_alphas** (*Sequence[np.ndarray]*) – Predict the probability that pixels belong to the foreground.
- **gt_alphas** (*Sequence[np.ndarray]*) – Probability that the actual pixel belongs to the foreground.
- **trimaps** (*Sequence[np.ndarray]*) – Broadly speaking, the trimap consists of foreground and unknown region.

compute_metric(*results: List*) → Dict[str, float]

Compute the ConnectivityError metric.

Parameters **results** (*List*) – A list that consisting the ConnectivityError score. This list has already been synced across all ranks.

Returns The computed ConnectivityError metric. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict[str, float]

15.1.27 DOTAMeanAP

class mmeval.metrics.**DOTAMeanAP**(*iou_thrs: Union[float, List[float]] = 0.5*, *scale_ranges: Optional[List[Tuple]] = None*, *num_classes: Optional[int] = None*, *eval_mode: str = '11points'*, *nproc: int = 4*, *drop_class_ap: bool = True*, *classwise: bool = False*, ***kwargs*)

DOTA evaluation metric.

DOTA is a large-scale dataset for object detection in aerial images which is introduced in https://arxiv.org/abs/1711.10398. This metric computes the DOTA mAP (mean Average Precision) with the given IoU thresholds and scale ranges.

Parameters

- **iou_thrs** (*float | List[float]*) – IoU thresholds. Defaults to 0.5.
- **scale_ranges** (*List[tuple], optional*) – Scale ranges for evaluating mAP. If not specified, all bounding boxes would be included in evaluation. Defaults to None.
- **num_classes** (*int, optional*) – The number of classes. If None, it will be obtained from the 'CLASSES' field in `self.dataset_meta`. Defaults to None.
- **eval_mode** (*str*) – 'area' or '11points', 'area' means calculating the area under precision-recall curve, '11points' means calculating the average precision of recalls at [0, 0.1, ..., 1]. The PASCAL VOC2007 defaults to use '11points', while PASCAL VOC2012 defaults to use 'area'. Defaults to '11points'.
- **nproc** (*int*) – Processes used for computing TP and FP. If nproc is less than or equal to 1, multiprocessing will not be used. Defaults to 4.
- **drop_class_ap** (*bool*) – Whether to drop the class without ground truth when calculating the average precision for each class.
- **classwise** (*bool*) – Whether to return the computed results of each class. Defaults to False.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> import numpy as np
>>> from mmeval import DOTAMetric
>>> num_classes = 15
>>> dota_metric = DOTAMetric(num_classes=15)
>>>
>>> def _gen_bboxes(num_bboxes, img_w=256, img_h=256):
...     # random generate bounding boxes in 'xywha' formart.
...     x = np.random.rand(num_bboxes, ) * img_w
...     y = np.random.rand(num_bboxes, ) * img_h
...     w = np.random.rand(num_bboxes, ) * (img_w - x)
...     h = np.random.rand(num_bboxes, ) * (img_h - y)
...     a = np.random.rand(num_bboxes, ) * np.pi / 2
```

(continues on next page)

(continued from previous page)

```
...     return np.stack([x, y, w, h, a], axis=1)
>>> prediction = {
...     'bboxes': _gen_bboxes(10),
...     'scores': np.random.rand(10, ),
...     'labels': np.random.randint(0, num_classes, size=(10, ))
... }
>>> groundtruth = {
...     'bboxes': _gen_bboxes(10),
...     'labels': np.random.randint(0, num_classes, size=(10, )),
...     'bboxes_ignore': _gen_bboxes(5),
...     'labels_ignore': np.random.randint(0, num_classes, size=(5, ))
... }
>>> dota_metric(predictions=[prediction, ], groundtruths=[groundtruth, ])
{'mAP@0.5': ..., 'mAP': ...}
```

add(*predictions: Sequence[Dict]*, *groundtruths: Sequence[Dict]*) → None

Add the intermediate results to `self._results`.

Parameters

- **predictions** (*Sequence[Dict]*) – A sequence of dict. Each dict representing a detection result for an image, with the following keys: - bboxes (numpy.ndarray): Shape (N, 5) or shape (N, 8).

  bounding bboxes of this image. The box format is depend on predict_box_type. Details in Note.

  - scores (numpy.ndarray): Shape (N,), the predicted scores of bounding boxes.
  - labels (numpy.ndarray): Shape (N,), the predicted labels of bounding boxes.

- **groundtruths** (*Sequence[Dict]*) – A sequence of dict. Each dict represents a groundtruths for an image, with the following keys:
  - bboxes (numpy.ndarray): Shape (M, 5) or shape (M, 8), the groundtruth bounding bboxes of this image, The box format is depend on predict_box_type. Details in Note.
  - labels (numpy.ndarray): Shape (M,), the ground truth labels of bounding boxes.
  - bboxes_ignore (numpy.ndarray): Shape (K, 5) or shape(K, 8), the groundtruth ignored bounding bboxes of this image. The box format is depend on `self.predict_box_type`.Details in upper note.
  - labels_ignore (numpy.ndarray): Shape (K,), the ground truth ignored labels of bounding boxes.

Note: The box shape of `predictions` and `groundtruths` is depends on the predict_box_type. If predict_box_type is 'rbox', the box shape should be (N, 5) which represents the (x, y,w, h, angle), otherwise the box shape should be (N, 8) which represents the (x1, y1, x2, y2, x3, y3, x4, y4).

15.1.28 ROUGE

class mmeval.metrics.**ROUGE**(*rouge_keys: Union[List, Tuple, int, str] = (1, 2, 'L')*, *use_stemmer: bool = False*, *normalizer: Optional[Callable] = None*, *tokenizer: Optional[Union[Callable, str]] = None*, *accumulate: str = 'best'*, *lowercase: bool = True*, ***kwargs: Any*)

Calculate Rouge Score used for automatic summarization.

This metric proposed in ROUGE: A Package for Automatic Evaluation of Summaries are common evaluation indicators in the fields of machine translation, automatic summarization, question and answer generation, etc.

Parameters

- **rouge_keys** (*List or Tuple or int or str*) – A list of rouge types to calculate. Keys that are allowed are `L`, and 1 through 9. Defaults to `(1, 2, 'L')`.
- **use_stemmer** (*bool*) – Use Porter stemmer to strip word suffixes to improve matching. Defaults to False.
- **normalizer** (*Callable, optional*) – A user's own normalizer function. If this is `None`, replacing any non-alpha-numeric characters with spaces is default. Defaults to None.
- **tokenizer** (*Callable or str, optional*) – A user's own tokenizer function. Defaults to None.
- **accumulate** (*str*) – Useful in case of multi-reference rouge score. `avg` takes the average of all references with respect to predictions. `best` takes the best fmeasure score obtained between prediction and multiple corresponding references. Defaults to `best`.
- **lowercase** (*bool*) – If it is True, all characters will be lowercase. Defaults to True.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import ROUGE
>>> predictions = ['the cat is on the mat']
>>> references = [['a cat is on the mat']]
>>> metric = ROUGE(rouge_keys='L')
>>> metric.add(predictions, references)
>>> results = metric.compute_metric()
{'rougeL_fmeasure': 0.8333333,
 'rougeL_precision': 0.8333333,
 'rougeL_recall': 0.8333333}
```

add(*predictions: Sequence[str]*, *references: Sequence[Sequence[str]]*) → None

Add the intermediate results to `self._results`.

Parameters

- **predictions** (*Sequence[str]*) – An iterable of predicted sentences.
- **references** (*Sequence[Sequence[str]*) – An iterable of referenced sentences. Each predicted sentence may correspond to multiple referenced sentences.

compute_metric(*results: List[Any]*) → dict

Compute the rouge metric.

This method would be invoked in `BaseMetric.compute` after distributed synchronization.

Parameters **results** (*List*) – A list that consists correct numbers. This list has already been synced across all ranks.

Returns The computed rouge score.

Return type Dict[str, float]

15.1.29 NaturalImageQualityEvaluator

class mmeval.metrics.**NaturalImageQualityEvaluator**(*crop_border: int = 0*, *input_order: str = 'CHW'*, *convert_to: str = 'gray'*, *channel_order: str = 'rgb'*, ***kwargs*)

Calculate Natural Image Quality Evaluator(NIQE) metric.

Ref: Making a "Completely Blind" Image Quality Analyzer. This implementation could produce almost the same results as the official MATLAB codes: http://live.ece.utexas.edu/research/quality/niqe_release.zip

Parameters

- **crop_border** (*int*) – Cropped pixels in each edges of an image. These pixels are not involved in the NIQE calculation. Defaults to 0.
- **input_order** (*str*) – Whether the input order is 'HWC' or 'CHW'. Defaults to 'CHW'.
- **convert_to** (*str*) – Convert the images to other color models. Options are 'y' and 'gray'. Defaults to 'gray'.
- **channel_order** (*str*) – The channel order of image. Defaults to 'rgb'.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import NaturalImageQualityEvaluator
>>> import numpy as np
>>>
>>> niqe = NaturalImageQualityEvaluator()
>>> preds = np.random.randint(0, 255, size=(3, 32, 32))
>>> niqe(preds)
{'niqe': ...}
```

add(*predictions: Sequence[numpy.ndarray]*, *channel_order: Optional[str] = None*) → None

Add NIQE score of batch to `self._results`

Parameters

- **predictions** (*Sequence[np.ndarray]*) – Predictions of the model. Each prediction should be a 4D (as a video, not batches of images) or 3D (as an image) array.
- **channel_order** (*Optional[str]*) – The channel order of the input samples. If not passed, will set as `self.channel_order`. Defaults to None.

compute_feature(*block: numpy.ndarray*) → List

Compute features.

Parameters **block** (*np.ndarray*) – 2D Image block.

Returns Features with length of 18.

Return type List

compute_metric(*results: List[numpy.float64]*) → Dict[str, float]
Compute the NaturalImageQualityEvaluator metric.

This method would be invoked in `BaseMetric.compute` after distributed synchronization.

Parameters **results** (*List[np.float64]*) – A list that consisting the NIQE score. This list has already been synced across all ranks.

Returns The computed NIQE metric.

Return type Dict[str, float]

compute_niqe(*prediction: numpy.ndarray*, *channel_order: str*, *mu_pris_param: numpy.ndarray*, *cov_pris_param: numpy.ndarray*, *gaussian_window: numpy.ndarray*, *block_size_h: int = 96*, *block_size_w: int = 96*) → numpy.float64
Calculate NIQE (Natural Image Quality Evaluator) metric. We use the official params estimated from the pristine dataset. We use the recommended block size (96, 96) without overlaps.

Note that we do not include block overlap height and width, since they are always 0 in the official implementation.

For good performance, it is advisable by the official implementation to divide the distorted image into the same size patched as used for the construction of multivariate Gaussian model.

Parameters

- **prediction** (*np.ndarray*) – Input image whose quality to be computed. Range [0, 255] with float type.
- **channel_order** (*str*) – The channel order of image.
- **mu_pris_param** (*np.ndarray*) – Mean of a pre-defined multivariate Gaussian model calculated on the pristine dataset.
- **cov_pris_param** (*np.ndarray*) – Covariance of a pre-defined multivariate Gaussian model calculated on the pristine dataset.
- **gaussian_window** (*ndarray*) – A 7x7 Gaussian window used for smoothing the image.
- **block_size_h** (*int*) – Height of the blocks in to which image is divided. Defaults to 96.
- **block_size_w** (*int*) – Width of the blocks in to which image is divided. Defaults to 96.

Returns NIQE result.

Return type np.float64

estimate_aggd_param(*block: numpy.ndarray*) → Tuple
Estimate AGGD (Asymmetric Generalized Gaussian Distribution) parameters.

Parameters **block** (*np.ndarray*) – 2D Image block.

Returns alpha(float), beta_l(float) and beta_r(float) for the AGGD distribution (Estimating parameters in Equation 7 in the paper).

Return type Tuple

get_size_from_scale(*input_size: Tuple*, *scale_factor: List[float]*) → List[int]
Get the output size given input size and scale factor.

Parameters

- **input_size** (*Tuple*) – The size of the input image.

- **scale_factor** (*List[float]*) – The resize factor.

Returns The size of the output image.

Return type List[int]

get_weights_indices(*input_length: int*, *output_length: int*, *scale: float*, *kernel: Callable*, *kernel_width: float*) → Tuple[List[numpy.ndarray], List[numpy.ndarray]]

Get weights and indices for interpolation.

Parameters

- **input_length** (*int*) – Length of the input sequence.
- **output_length** (*int*) – Length of the output sequence.
- **scale** (*float*) – Scale factor.
- **kernel** (*Callable*) – The kernel used for resizing.
- **kernel_width** (*float*) – The width of the kernel.

Returns The weights and the indices for interpolation.

Return type Tuple[List[np.ndarray], List[np.ndarray]]

matlab_resize(*img: numpy.ndarray*, *scale: float*) → numpy.ndarray

Resize an image to the required size.

Parameters

- **img** (*np.ndarray*) – The original image.
- **scale** (*float*) – The scale factor of the resize operation.

Returns The resized image.

Return type np.ndarray

resize_along_dim(*img_in: numpy.ndarray*, *weights: numpy.ndarray*, *indices: numpy.ndarray*, *dim: int*) → numpy.ndarray

Resize along a specific dimension.

Parameters

- **img_in** (*np.ndarray*) – The input image.
- **weights** (*np.ndarray*) – The weights used for interpolation, computed from [get_weights_indices].
- **indices** (*np.ndarray*) – The indices used for interpolation, computed from [get_weights_indices].
- **dim** (*int*) – Which dimension to undergo interpolation.

Returns Interpolated (along one dimension) image.

Return type np.ndarray

15.1.30 Perplexity

class mmeval.metrics.**Perplexity**(*ignore_labels: Optional[Union[int, List[int]]] = None*, ***kwargs*)

Perplexity measures how well a language model predicts a text sample.

It is commonly used as a metric for evaluating the quality of a language model. It is defined as 2 to the power of the cross-entropy loss of the model (or the negative log-likelihood of the sample).

Parameters

- **ignore_labels** (*int or list[int], optional*) – Integer specifying a target class to ignore. If given, this class index does not contribute to the returned score. Defaults to None.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import Perplexity
>>> import numpy as np
>>>
>>> preds = np.random.rand(2, 4, 2)
>>> targets = np.random.randint(low=0, high=2, size=(2, 4))
>>> metric = Perplexity()
>>> result = metric(preds, targets)
{'perplexity': ...}
```

add(*predictions: Sequence*, *targets: Sequence*) → None

Add the intermediate results to `self._results`.

Parameters

- **predictions** (*Sequence*) – Probabilities assigned to each token in a sequence with shape [batch_size, seq_len, vocab_size].
- **targets** (*Sequence*) – Ground truth values with a shape [batch_size, seq_len].

compute_metric(*results: List[Tuple[float, int]]*) → Dict[str, float]

Compute the perplexity metric.

This method would be invoked in `BaseMetric.compute` after distributed synchronization.

Parameters **results** (*list*) – A list that consisting the total and count. This list has already been synced across all ranks.

Returns The computed perplexity metric.

Return type Dict[str, float]

15.1.31 CharRecallPrecision

class mmeval.metrics.**CharRecallPrecision**(*letter_case: str = 'unchanged'*, *invalid_symbol: str = '[^A-Za-z0-9-]'*, ***kwargs*)

Calculate the char level recall & precision.

Parameters

- **letter_case** (*str*) – There are three options to alter the letter cases
  - unchanged: Do not change prediction texts and labels.
  - upper: Convert prediction texts and labels into uppercase characters.
  - lower: Convert prediction texts and labels into lowercase characters.

  Usually, it only works for English characters. Defaults to 'unchanged'.
- **invalid_symbol** (*str*) – A regular expression to filter out invalid or not cared characters. Defaults to '[^A-Za-z0-9u4e00-u9fa5]'.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import CharRecallPrecision
>>> metric = CharRecallPrecision()
>>> metric(['helL', 'HEL'], ['hello', 'HELLO'])
{'char_recall': 0.6, 'char_precision': 0.857142857142857}
>>> metric = CharRecallPrecision(letter_case='upper')
>>> metric(['helL', 'HEL'], ['hello', 'HELLO'])
{'char_recall': 0.7, 'char_precision': 1.0}
```

add(*predictions: Sequence[str]*, *groundtruths: Sequence[str]*) → None

Process one batch of data and predictions.

Parameters

- **predictions** (*list[str]*) – The prediction texts.
- **groundtruths** (*list[str]*) – The ground truth texts.

compute_metric(*results: Sequence[Tuple[int, int, int]]*) → Dict

Compute the metrics from processed results.

Parameters **results** (*list[tuple]*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict

15.1.32 KeypointEndPointError

class mmeval.metrics.**KeypointEndPointError**(*dataset_meta: Optional[Dict] = None*, *dist_collect_mode: str = 'unzip'*, *dist_backend: Optional[str] = None*, *logger: Optional[logging.Logger] = None*)

EPE evaluation metric.

Calculate the end-point error (EPE) of keypoints.

Note:

- length of dataset: N
- num_keypoints: K
- number of keypoint dimensions: D (typically D = 2)

Examples

```
>>> from mmeval.metrics import KeypointEndPointError
>>> import numpy as np
>>> output = np.array([[[10.,  4.],
...     [10., 18.],
...     [ 0.,  0.],
...     [40., 40.],
...     [20., 10.]]])
>>> target = np.array([[[10.,  0.],
...     [10., 10.],
...     [ 0., -1.],
...     [30., 30.],
...     [ 0., 10.]]])
>>> keypoints_visible = np.array([[True, True, False, True, True]])
>>> predictions = [{'coords': output}]
>>> groundtruths = [{'coords': target, 'mask': keypoints_visible}]
>>> epe_metric = KeypointEndPointError()
>>> epe_metric(predictions, groundtruths)
{'EPE': 11.535533905029297}
```

add(*predictions: Sequence[Dict]*, *groundtruths: Sequence[Dict]*) → None

Process one batch of predictions and groundtruths and add the intermediate results to *self._results*.

Parameters

- **predictions** (*Sequence[dict]*) – Predictions from the model. Each prediction dict has the following keys:
  - coords (np.ndarray, [1, K, D]): predicted keypoints coordinates
- **groundtruths** (*Sequence[dict]*) – The ground truth labels. Each groundtruth dict has the following keys:
  - coords (np.ndarray, [1, K, D]): ground truth keypoints coordinates
  - mask (np.ndarray, [1, K]): ground truth keypoints_visible

compute_metric(*results: list*) → Dict[str, float]

Compute the metrics from processed results.

Parameters **results** (*list*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict[str, float]

15.1.33 KeypointAUC

class mmeval.metrics.**KeypointAUC**(*norm_factor: float = 30*, *num_thrs: int = 20*, ***kwargs*)

AUC evaluation metric.

Calculate the Area Under Curve (AUC) of keypoint PCK accuracy.

By altering the threshold percentage in the calculation of PCK accuracy, AUC can be generated to further evaluate the pose estimation algorithms.

Note:

- length of dataset: N
- num_keypoints: K
- number of keypoint dimensions: D (typically D = 2)

Parameters

- **norm_factor** (*float*) – AUC normalization factor, Defaults to 30 (pixels).
- **num_thrs** (*int*) – Number of thresholds to calculate AUC. Defaults to 20.
- ****kwargs** – Keyword parameters passed to `mmeval.BaseMetric`. Must include `dataset_meta` in order to compute the metric.

Examples

```
>>> from mmeval import KeypointAUC
>>> import numpy as np
>>> auc_metric = KeypointAUC(norm_factor=20, num_thrs=4)
>>> output = np.array([[[10.,  4.],
...     [10., 18.],
...     [ 0.,  0.],
...     [40., 40.],
...     [20., 10.]]])
>>> target = np.array([[[10.,  0.],
...     [10., 10.],
...     [ 0., -1.],
...     [30., 30.],
...     [ 0., 10.]]])
>>> keypoints_visible = np.array([[True, True, False, True, True]])
>>> num_keypoints = 15
>>> prediction = {'coords': output}
>>> groundtruth = {'coords': target, 'mask': keypoints_visible}
>>> predictions = [prediction]
>>> groundtruths = [groundtruth]
```

(continues on next page)

(continued from previous page)

```
>>> auc_metric(predictions, groundtruths)
OrderedDict([('AUC@4', 0.375)])
```

add(*predictions: List[Dict]*, *groundtruths: List[Dict]*) → None
Process one batch of predictions and groundtruths and add the intermediate results to *self._results*.

Parameters

- **predictions** (*Sequence[dict]*) – Predictions from the model. Each prediction dict has the following keys:
  - coords (np.ndarray, [1, K, D]): predicted keypoints coordinates
- **groundtruths** (*Sequence[dict]*) – The ground truth labels. Each groundtruth dict has the following keys:
  - coords (np.ndarray, [1, K, D]): ground truth keypoints coordinates
  - mask (np.ndarray, [1, K]): ground truth keypoints_visible

compute_metric(*results: list*) → Dict[str, float]
Compute the metrics from processed results.

Parameters **results** (*list*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict[str, float]

15.1.34 KeypointNME

class mmeval.metrics.**KeypointNME**(*norm_mode: str*, *norm_item: Optional[str] = None*, *keypoint_indices: Optional[Sequence[int]] = None*, ***kwargs*)
NME evaluation metric.

Calculate the normalized mean error (NME) of keypoints.

Note:

- length of dataset: N
- num_keypoints: K
- number of keypoint dimensions: D (typically D = 2)

Parameters

- **norm_mode** (*str*) – The normalization mode, which should be one of the following options:
  - `'use_norm_item'`: Should specify the argument *norm_item*, which represents the item in the datainfo that will be used as the normalization factor.
  - `'keypoint_distance'`: Should specify the argument *keypoint_indices* that are used to calculate the keypoint distance as the normalization factor.

- **norm_item** (*str, optional*) – The item used as the normalization factor. For example, *'box_size'* in *'AFLWDataset'*. Only valid when `norm_mode` is `use_norm_item`. Defaults to `None`.
- **keypoint_indices** (*Sequence[int], optional*) – The keypoint indices used to calculate the keypoint distance as the normalization factor. Only valid when `norm_mode` is `keypoint_distance`. If set as None, will use the default `keypoint_indices` in *DEFAULT_KEYPOINT_INDICES* for specific datasets, else use the given `keypoint_indices` of the dataset. Defaults to `None`.
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval.metrics import KeypointNME
>>> import numpy as np
>>> aflw_dataset_meta = {
...     'dataset_name': 'aflw',
...     'num_keypoints': 19,
...     'sigmas': np.array([]),
... }
>>> nme_metric = KeypointNME(
...     norm_mode='use_norm_item',
...     norm_item=norm_item,
...     dataset_meta=aflw_dataset_meta)
>>> batch_size = 2
>>> predictions = [{
...     'coords': np.zeros((1, 19, 2))
... } for _ in range(batch_size)]
>>> groundtruths = [{
...     'coords': np.zeros((1, 19, 2)) + 0.5,
...     'mask': np.ones((1, 19)).astype(bool),
...     'box_size': np.ones((1, 1)) * i * 20
... } for i in range(batch_size)]
>>> norm_item = 'box_size'
>>> nme_metric(predictions, groundtruths)
OrderedDict([('NME', 0.03535533892480951)])
```

add(*predictions: Sequence[Dict]*, *groundtruths: Sequence[Dict]*) → None

Add the intermediate results to *self._results*.

Parameters

- **predictions** (*Sequence[dict]*) – A sequence of dict. Each prediction dict has the following keys:
  - coords (np.ndarray, [1, K, D]): predicted keypoints coordinates
- **groundtruths** (*Sequence[dict]*) – The ground truth labels. Each groundtruth dict has the following keys:
  - coords (np.ndarray, [1, K, D]): ground truth keypoints coordinates
  - mask (np.ndarray, [1, K]): ground truth keypoints_visible

  There are some optional keys as well:
  - bboxes: it is necessary when `self.norm_item` is *'bbox_size'*

- `self.norm_item`: it is necessary when `self.norm_item` is neither `None` nor *'bbox_size'*

compute_metric(*results: list*) → Dict[str, float]
Compute the metrics from processed results.

Parameters **results** (*list*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are the corresponding results.

Return type Dict[str, float]

15.1.35 WordAccuracy

class mmeval.metrics.**WordAccuracy**(*mode: Union[str, Sequence[str]] = 'ignore_case_symbol'*, *invalid_symbol: str = '[^A-Za-z0-9-]'*, ***kwargs*)
Calculate the word level accuracy.

Parameters

- **mode** (*str or list[str]*) – Options are:
  - 'exact': Accuracy at word level.
  - 'ignore_case': Accuracy at word level, ignoring letter case.
  - 'ignore_case_symbol': Accuracy at word level, ignoring letter case and symbol. (Default metric for academic evaluation)

  If mode is a list, then metrics in mode will be calculated separately. Defaults to 'ignore_case_symbol'.
- **invalid_symbol** (*str*) – A regular expression to filter out invalid or not cared characters. Defaults to '[^A-Za-z0-9u4e00-u9fa5]'
- ****kwargs** – Keyword parameters passed to `BaseMetric`.

Examples

```
>>> from mmeval import WordAccuracy
>>> metric = WordAccuracy()
>>> metric(['hello', 'hello', 'hello'], ['hello', 'HELLO', '$HELLO$'])
{'ignore_case_symbol_accuracy': 1.0}
>>> metric = WordAccuracy(mode=['exact', 'ignore_case',
>>>                             'ignore_case_symbol'])
>>> metric(['hello', 'hello', 'hello'], ['hello', 'HELLO', '$HELLO$'])
{'accuracy': 0.333333333,
 'ignore_case_accuracy': 0.666666667,
 'ignore_case_symbol_accuracy': 1.0}
```

add(*predictions: Sequence[str]*, *groundtruths: Sequence[str]*) → None
Process one batch of data and predictions.

Parameters

- **predictions** (*list[str]*) – The prediction texts.
- **groundtruths** (*list[str]*) – The ground truth texts.

compute_metric(*results: List[Tuple[int, int, int]]*) → Dict
Compute the metrics from processed results.

Parameters **results** (*list[float]*) – The processed results of each batch.

Returns

Nested dicts as results. Provided keys are:

- accuracy (float): Accuracy at word level.
- ignore_case_accuracy (float): Accuracy at word level, ignoring letter case.
- ignore_case_symbol_accuracy (float): Accuracy at word level, ignoring letter case and symbol.

Return type dict[str, float]

CHAPTER

SIXTEEN

MMEVAL.UTILS

mmeval.utils

- *misc*

16.1 misc

try_import	Try to import a module.
has_method	Check whether the object has a method.
is_seq_of	Check whether it is a sequence of some type.
is_list_of	Check whether it is a list of some type.
is_tuple_of	Check whether it is a tuple of some type.
is_filepath	Check if the given object is Path-like.

16.1.1 mmeval.utils.try_import

mmeval.utils.**try_import**(*name: str*) → Optional[module]

Try to import a module.

Parameters **name** (*str*) – Specifies what module to import in absolute or relative terms (e.g. either pkg.mod or ..mod).

Returns If importing successfully, returns the imported module, otherwise returns None.

Return type ModuleType or None

16.1.2 mmeval.utils.has_method

mmeval.utils.**has_method**(*obj: object*, *method: str*) → bool

Check whether the object has a method.

Parameters

- **method** (*str*) – The method name to check.
- **obj** (*object*) – The object to check.

Returns True if the object has the method else False.

Return type bool

16.1.3 mmeval.utils.is_seq_of

mmeval.utils.**is_seq_of**(*seq: Any*, *expected_type: Union[Type, tuple]*, *seq_type: Optional[Type] = None*) → bool

Check whether it is a sequence of some type.

Parameters

- **seq** (*Sequence*) – The sequence to be checked.
- **expected_type** (*type or tuple*) – Expected type of sequence items.
- **seq_type** (*type, optional*) – Expected sequence type. Defaults to None.

Returns Return True if `seq` is valid else False.

Return type bool

Examples

```
>>> from mmeval.utils import is_seq_of
>>> seq = ['a', 'b', 'c']
>>> is_seq_of(seq, str)
True
>>> is_seq_of(seq, int)
False
```

16.1.4 mmeval.utils.is_list_of

mmeval.utils.**is_list_of**(*seq*, *expected_type*)

Check whether it is a list of some type.

A partial method of *is_seq_of()*.

16.1.5 mmeval.utils.is_tuple_of

mmeval.utils.**is_tuple_of**(*seq*, *expected_type*)

Check whether it is a tuple of some type.

A partial method of *is_seq_of()*.

16.1.6 mmeval.utils.is_filepath

mmeval.utils.**is_filepath**(*x*)

Check if the given object is Path-like.

Parameters **x** (*object*) – Any object.

Returns Returns True if the given is a str or `pathlib.Path`. Otherwise, returns False.

Return type bool

CHAPTER

SEVENTEEN

CHANGELOG OF V0.X

CHAPTER

EIGHTEEN

ENGLISH

CHAPTER

NINETEEN

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

D

E

N

O

P

R

S

T

V

W

X

Y