Quantus

Anna Hedström

Mar 25, 2024

INSTALLATION

Quantus is an eXplainable AI toolkit for responsible evaluation of neural network explanations.

This documentation is complementary to the README.md in the Quantus repository and provides documentation for how to *install* Quantus, how to *contribute* and details on the API. For further guidance on what to think about when applying Quantus, please read the *user guidelines*. Do you want to get started? Please have a look at our simple *toy example* with PyTorch using MNIST data. For more examples, check the tutorials folder.

Quantus can be installed from PyPI (this way assumes that you have either PyTorch or Tensorflow installed on your machine):

```
pip install quantus
```

For a more in-depth guide on how to install Quantus, read more here. This includes instructions for how to install a desired deep learning framework such as PyTorch or tensorflow together with Quantus.

CHAPTER

ONE

CONTENTS

1.1 Quick installation

You can install Quantus in various ways. The different options are listed in the following.

1.1.1 Installing via PyPI

If you already have PyTorch or TensorFlow installed on your machine, the most light-weight version of Quantus can be obtained from PyPI as follows (no additional explainability functionality or deep learning framework will be included):

```
pip install quantus
```

Alternatively, you can simply add the desired deep learning framework (in brackets) to have the package installed together with Quantus. To install Quantus with PyTorch, please run:

```
pip install "quantus[torch]"
```

For TensorFlow, please run:

```
pip install "quantus[tensorflow]"
```

1.1.2 Installing additional XAI Library support (PyPI only)

Most evaluation metrics in Quantus allow for a choice of either providing pre-computed explanations directly as an input, or instead making use of several wrappers implemented in `quantus.explain` around common explainability libraries. The following XAI Libraries are currently supported:

Captum

To enable the use of wrappers around Captum, you can run:

```
pip install "quantus[captum]"
```

tf-explain

To enable the use of wrappers around tf.explain, you can run:

```
pip install "quantus[tf-explain]"
```

Zennit

To use Quantus with support for the Zennit library, you can run:

```
pip install "quantus[zennit]"
```

Note that the three options above will also install the required frameworks (i.e., PyTorch or TensorFlow) respectively, if they are not already installed in your environment. Note also, that not all explanation methods offered in **Captum** and **tf-explain** are included in `quantus.explain`.

1.1.3 Installing tutorial requirements

The Quantus tutorials have more requirements than the base package, which you can install by running

```
pip install "quantus[tutorials]"
```

1.1.4 Full installation

To simply install all of the above, you can run

```
pip install "quantus[full]"
```

1.1.5 Package requirements

The package requirements are as follows:

```
python>=3.8.0
torch>=1.11.0
tensorflow>=2.5.0
```

Please note that the exact PyTorch and/ or TensorFlow versions to be installed depends on your Python version (3.8-3.11) and platform (`darwin`, `linux`, ...). See `[project.optional-dependencies]` section in the `pyproject.toml` file.

1.2 Getting Started

The following will give a short introduction to how to get started with Quantus.

Note that this example is based on the PyTorch framework, but we also support TensorFlow, which would differ only in the *preliminaries* i.e., the loading of model, data and explanations.

1.2.1 Preliminaries

Quantus implements methods for the quantitative evaluation of XAI methods. Generally, in order to apply these, you will need:

- A model (`model`), inputs (`x_batch`) and labels (`y_batch`)
- Some explanations you want to evaluate (`a_batch`)

Step 1. Load data and model

Let's first load the data and model. In this example, a pre-trained LeNet available from Quantus for the purpose of this tutorial is loaded, but generally, you might use any Pytorch (or TensorFlow) model instead.

```
import quantus
from quantus import LeNet
import torch
import torchvision

# Enable GPU.
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Load a pre-trained LeNet classification model (architecture at quantus/helpers/models).
model = LeNet()
model.load_state_dict(torch.load("tests/assets/mnist_model"))

# Load datasets and make loaders.
test_set = torchvision.datasets.MNIST(root='./sample_data', download=True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

# Load a batch of inputs and outputs to use for XAI evaluation.
x_batch, y_batch = iter(test_loader).next()
x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()
```

Step 2. Load explanations

We still need some explanations to evaluate. For this, there are two possibilities in Quantus. You can provide either:

1. a set of re-computed attributions (`np.ndarray`)
2. any arbitrary explanation function (`callable`), e.g., the built-in method `quantus.explain` or your own customised function

We describe the different options in detail below.

a) Using pre-computed explanations

Quantus allows you to evaluate explanations that you have pre-computed, assuming that they match the data you provide in `x_batch`. Let's say you have explanations for Saliency and Integrated Gradients already pre-computed.

In that case, you can simply load these into corresponding variables `a_batch_saliency` and `a_batch_intgrad`:

```
a_batch_saliency = load("path/to/precomputed/saliency/explanations")
a_batch_saliency = load("path/to/precomputed/intgrad/explanations")
```

Another option is to simply obtain the attributions using one of many XAI frameworks out there, such as Captum, Zennit, tf.explain, or iNNvestigate. The following code example shows how to obtain explanations (Saliency and Integrated Gradients, to be specific) using Captum:

```
import captum
from captum.attr import Saliency, IntegratedGradients

# Generate Integrated Gradients attributions of the first batch of the test set.
```

(continues on next page)

(continued from previous page)

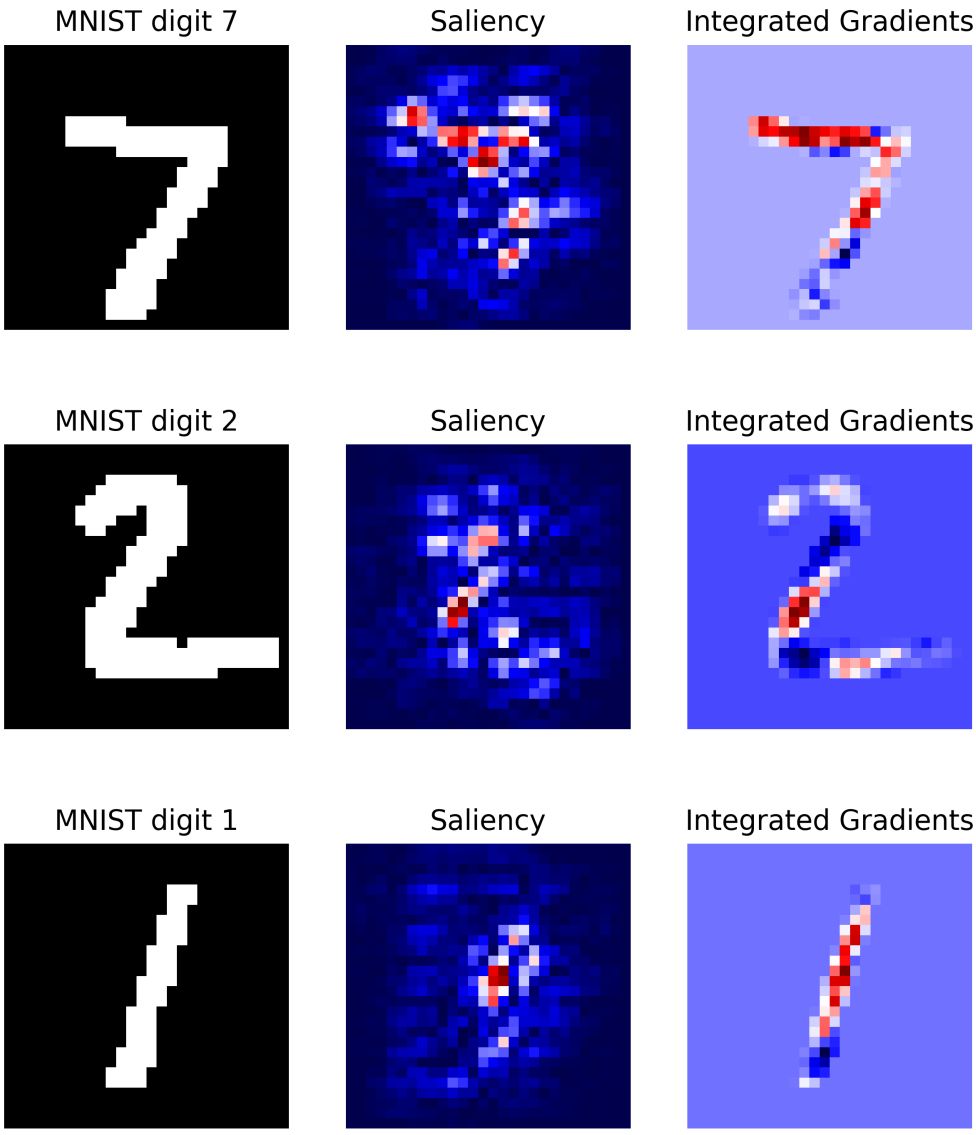
```
a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).
↪sum(axis=1).cpu().numpy()
a_batch_intgrad = IntegratedGradients(model).attribute(inputs=x_batch, target=y_batch,␣
↪baselines=torch.zeros_like(x_batch)).sum(axis=1).cpu().numpy()

# Save x_batch and y_batch as numpy arrays that will be used to call metric instances.
x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

# Quick assert.
assert [isinstance(obj, np.ndarray) for obj in [x_batch, y_batch, a_batch_saliency, a_
↪batch_intgrad]]
```

b) Passing an explanation function

If you don't have a pre-computed set of explanations but rather want to pass an explanation function that you wish to evaluate with Quantus, this option exists.

For this, you can rely on the built-in `quantus.explain` function, which includes some popular explanation methods (please run `quantus.available_methods()` to see which ones). Note, however, that the set of explanation methods offered in `quantus.explain` are limited — `quantus.explain` is a wrapper around Captum, Zennit, and tf.explain but does not support every explanation method offered in the respective libraries.

If you want to use Quantus to evaluate any arbitrary explanation method (e.g., your newly built explanation function), you can simply provide your own function (`callable`, see also *Extending quantus*). Examples of how to use `quantus.explain` or your own customised explanation function are included in the next section.



As seen in the above image, the qualitative aspects of explanations may look fairly uninterpretable — since we lack ground truth of what the explanations should be looking like, it is hard to draw conclusions about the explainable evidence.

1.2.2 Step 3. Evaluate explanations with Quantus

To gather quantitative evidence for the quality of the different explanation methods, we can apply Quantus.

Initialise metrics

Quantus implements XAI evaluation metrics from different categories, e.g., Faithfulness, Localisation and Robustness etc which all inherit from the base `quantus.Metric` class. To apply a metric to your setting (e.g., Max-Sensitivity) it first needs to be instantiated:

```
metric = quantus.MaxSensitivity()
```

and then applied to your model, data, and (pre-computed) explanations:

```
scores = metric(
    model=model,
    x_batch=x_batch,
    y_batch=y_batch,
    device=device,
    explain_func=quantus.explain,
    explain_func_kwargs={"method": "Saliency"}
)
```

Since a re-computation of the explanations is necessary for robustness evaluation, in this example, we also pass an explanation function (`explain_func`) to the metric call. Here, we rely on the built-in `quantus.explain` function to recompute the explanations. The hyperparameters are set with the `explain_func_kwargs` dictionary. Please find more details on how to use `quantus.explain` at API documentation.

You can alternatively use your own customised explanation function (assuming it returns an `np.ndarray` in a shape that matches the input `x_batch`). This is done as follows:

```
def your_own_callable(model, inputs, targets, **kwargs) -> np.ndarray:
  """Logic goes here to compute the attributions and return an
  explanation  in the same shape as x_batch (np.array),
  (flatten channels if necessary)."""
  return explanation(model, inputs, targets)

scores = metric(
    model=model,
    x_batch=x_batch,
    y_batch=y_batch,
    device=device,
    explain_func=your_own_callable
)
```

Customising metrics

The metrics for evaluating XAI methods are often quite sensitive to their respective hyperparameters. For instance, how explanations are normalised or whether signed or unsigned explanations are considered can have a significant impact on the results of the evaluation. However, some metrics require normalisation or unsigned values, while others are more flexible.

Therefore, different metrics can have different hyperparameters or default values in Quantus, which are documented in detail here. We encourage users to read the respective documentation before applying each metric, to gain an understanding of the implications of altering each hyperparameter.

To get an overview of the hyperparameters for a specific metric, please run:

```
metric.get_params
```

Nevertheless, for the purpose of robust evaluation, it makes sense to vary especially those hyperparameters that metrics tend to be sensitive to. Generally, hyperparameters for each metric are separated as follows:

- Hyperparameters affecting the metric function itself are set in the `__init__` method of each metric. Extending the above example of MaxSensitivity, various init hyperparameters can be set as follows:

```
max_sensitivity = quantus.MaxSensitivity(
    nr_samples=10,
    lower_bound=0.2,
    norm_numerator=quantus.fro_norm,
    norm_denominator=quantus.fro_norm,
    perturb_func=quantus.uniform_noise,
    similarity_func=quantus.difference
)
```

- Hyperparameters affecting the inputs (data, model, explanations) to each metric are set in the `__call__` method of each metric. Extending the above example of MaxSensitivity, various call hyperparameters can be set as follows:

```
result = max_sensitivity(
    model=model,
    x_batch=x_batch,
    y_batch=y_batch,
    device=device,
    explain_func=quantus.explain,
    explain_func_kwargs={"method": "Saliency"},
    softmax=False
)
```

Large-scale evaluations

Quantus also provides high-level functionality to support large-scale evaluations, e.g., multiple XAI methods, multi-faceted evaluation through several metrics, or a combination thereof.

To utilise `quantus.evaluate()`, you simply need to define two things:

1. The **Metrics** you would like to use for evaluation (each `__init__` parameter configuration counts as its own metric):

```
metrics = {
    "max-sensitivity-10": quantus.MaxSensitivity(nr_samples=10),
    "max-sensitivity-20": quantus.MaxSensitivity(nr_samples=20),
    "region-perturbation": quantus.RegionPerturbation(),
}
```

2. The **XAI methods** you would like to evaluate, as a `dict` with pre-computed attributions:

```
xai_methods = {
    "Saliency": a_batch_saliency,
    "IntegratedGradients": a_batch_intgrad
}
```

or as a `dict` but with explanation functions:

```
xai_methods = {
    "Saliency": saliency_callable,
    "IntegratedGradients": saliency_callable
}
```

or as a `dict` with keys with the name of the Quantus built-in explanation methods (see `quantus.explain`), and the values are associated hyperparameters (as a `dict`):

```
xai_methods = {
    "Saliency": {},
    "IntegratedGradients": {}
}
```

You can then simply run a large-scale evaluation as follows (this aggregates the result by `np.mean` averaging):

```
import numpy as np
results = quantus.evaluate(
      metrics=metrics,
      xai_methods=xai_methods,
      agg_func=np.mean,
      model=model,
      x_batch=x_batch,
      y_batch=y_batch,
      call_kwargs={"0": {"softmax": False,},}
)
```

Please see Getting started tutorial to run code similar to this example.

1.2.3 Extending Quantus

With Quantus, one can flexibly extend the library's functionality, e.g., to adopt a customised explainer function `explain_func` or to replace a function that perturbs the input `perturb_func` with a user-defined one. If you are extending or replacing a function within the Quantus framework, make sure that your new function:

- has the same **return type**
- expects the same **arguments**

as the function, you're intending to replace.

Details on what datatypes and arguments should be used for the different functions can be found in the respective function typing in *quantus.helpers*. For example, if you want to replace `similarity_func` in your evaluation, you can do as follows.

```
import scipy
import numpy as np

def my_similarity_func(a: np.array, b: np.array, **kwargs) -> float:
    """Calculate the similarity of a and b by subtraction."""
    return a - b

# Simply initialise the metric with your own function.
metric = quantus.LocalLipschitzEstimate(similarity_func=my_similar_func)
```

Similarly, if you are replacing or extending metrics, make sure they inherit from the `Metric` class in *quantus.metrics.base*. Each metric at least needs to implement the `Metric.evaluate_instance` method.

1.2.4 Miscellaneous

There are several miscellaneous helpers built into Quantus as follows:

```
# Interpret scores of a given metric.
metric_instance.interpret_scores

# Understand the hyperparameters of a metric.
sensitivity_scorer.get_params

# To list available metrics (and their corresponding categories).
quantus.AVAILABLE_METRICS

# To list available explainable methods with tf-explain.
quantus.AVAILABLE_XAI_METHODS_TF

# To list available explainable methods with captum.
quantus.AVAILABLE_XAI_METHODS_CAPTUM

# To list available perturbation functions.
quantus.AVAILABLE_SIMILARITY_FUNCTIONS

# To list available similarity functions.
quantus.AVAILABLE_PERTURBATION_FUNCTIONS

# To list available normalisation function.
quantus.AVAILABLE_NORMALISATION_FUNCTIONS

# To get the scores of the last evaluated batch.
metric_instance_called.evaluation_scores
```

Per default, warnings are printed to shell with each metric initialisation in order to make the user attentive to the hyperparameters of the metric which may have great influence on the evaluation outcome. If you are running evaluation iteratively you might want to disable warnings, then set:

`disable_warnings = True`

in the params of the metric initalisation. Additionally, if you want to track progress while evaluating your explanations set:

`display_progressbar = True`

If you want to return an aggregate score for your test samples you can set the following hyperparameter:

`return_aggregate = True`

for which you can specify an `aggregate_func` e.g., `np.mean` to use while aggregating the score for a given metric.

1.2.5 Tutorials

Further tutorials are available that showcase the many types of analysis that can be done using Quantus. For this purpose, please see notebooks in the tutorials folder which includes examples such as:

- All Metrics ImageNet Example: shows how to instantiate the different metrics for ImageNet dataset
- Metric Parameterisation Analysis: explores how sensitive a metric could be to its hyperparameters
- Robustness Analysis Model Training: measures robustness of explanations as model accuracy increases
- Full Quantification with Quantus: example of benchmarking explanation methods
- Tabular Data Example: example of how to use Quantus with tabular data
- Quantus and TensorFlow Data Example: showcases how to use Quantus with TensorFlow

... and more.

1.3 quantus package

1.3.1 Subpackages

quantus.functions package

Submodules

quantus.functions.discretise_func module

This module holds a collection of explanation discretisation functions i.e., methods to split continuous explanation spaces into discrete counterparts.

`quantus.functions.discretise_func.`**`floating_points`**(*a: array*, ***kwargs*) → float

Rounds input to have n floating-points representation

Parameters

a: np.ndarray
Numpy array with shape (x,).

kwargs: optional

Keyword arguments.

n: integer Number of floating point digits.

Returns

float
Returns the hash values of the resulting array.

quantus.functions.discretise_func.**rank**(*a: array*, ***kwargs*) → float

Calculates indices that would sort the array in order of importance.

Parameters

a: np.ndarray
Numpy array with shape (x,).

kwargs: optional
Keyword arguments.

Returns

float
Returns the hash values of the resulting array.

quantus.functions.discretise_func.**sign**(*a: array*, ***kwargs*) → float

Calculates element-wise signs of the array.

Parameters

a: np.ndarray
Numpy array with shape (x,).

kwargs: optional
Keyword arguments.

Returns

float
Returns the hash values of the resulting array.

quantus.functions.discretise_func.**top_n_sign**(*a: array*, ***kwargs*) → float

Calculates top n element-wise signs of the array.

Parameters

a: np.ndarray
Numpy array with shape (x,).

kwargs: optional

Keyword arguments.

n: integer Number of floating point digits.

Returns

float
Returns the hash values of the resulting array.

quantus.functions.explanation_func module

This modules contains explainer functions which can be used in conjunction with the metrics in the library.

quantus.functions.explanation_func.**explain**(*model*, *inputs*, *targets*, ***kwargs*) → ndarray

Explain inputs given a model, targets and an explanation method. Expecting inputs to be shaped such as (batch_size, nr_channels, ...) or (batch_size, ..., nr_channels).

Parameters

model: torch.nn.Module, tf.keras.Model
A model that is used for explanation.

inputs: np.ndarray
The inputs that ought to be explained.

targets: np.ndarray
The target lables that should be used in the explanation.

kwargs: optional
Keyword arguments. Pass as "explain_func_kwargs" dictionary when working with a metric class. Pass as regular kwargs when using the stnad-alone function.

xai_lib: string, optional
XAI library: captum, tf-explain or zennit.

method: string, optional
XAI method (used with captum and tf-explain libraries).

attributor: string, optional
XAI method (used with zennit).

xai_lib_kwargs: dictionary, optional
Keyword arguments to be passed to the attribution function.

softmax: boolean, optional
Indicated whether softmax activation in the last layer shall be removed.

channel_first: boolean, optional
Indicates if the image dimensions are channel first, or channel last. Inferred from the input shape if None.

reduce_axes: tuple
Indicates the indices of dimensions of the output explanation array to be summed. For example, an input array of shape (8, 28, 28, 3) with keepdims=True and reduce_axes = (-1,) will return an array of shape (8, 28, 28, -1). Passing "()" will keep the original dimensions.

keepdims: boolean
Indicated if the reduced axes shall be preserved (True) or removed (False).

Returns

explanation: np.ndarray
Returns np.ndarray of same shape as inputs.

quantus.functions.explanation_func.**generate_captum_explanation**(*model*, *inputs: ndarray*, *targets: ndarray*, *device: str | None = None*, ***kwargs*) → ndarray

Generate explanation for a torch model with captum. Parameters ———- model: torch.nn.Module

A model that is used for explanation.

inputs: np.ndarray
The inputs that ought to be explained.

targets: np.ndarray
The target lables that should be used in the explanation.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments. Pass as "explain_func_kwargs" dictionary when working with a metric class. Pass as regular kwargs when using the stnad-alone function. May include xai_lib_kwargs dictionary which includes keyword arguments for a method call.

xai_lib: string
XAI library: captum, tf-explain or zennit.

method: string
XAI method.

xai_lib_kwargs: dict
Keyword arguments to be passed to the attribution function.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

reduce_axes: tuple
Indicates the indices of dimensions of the output explanation array to be summed. For example, an input array of shape (8, 28, 28, 3) with keepdims=True and reduce_axes = (-1,) will return an array of shape (8, 28, 28, -1). Passing "()" will keep the original dimensions.

keepdims: boolean
Indicated if the reduced axes shall be preserved (True) or removed (False).

Returns

explanation: np.ndarray
Returns np.ndarray of same shape as inputs.

quantus.functions.explanation_func.**generate_tf_explanation**(*model*, *inputs: array*, *targets: array*, ***kwargs*) → ndarray

Generate explanation for a tf model with tf_explain. Assumption: Currently only normalised absolute values of explanations supported.

Parameters

model: tf.keras.Model
A model that is used for explanation.

inputs: np.ndarray
The inputs that ought to be explained.

targets: np.ndarray
The target lables that should be used in the explanation.

kwargs: optional
Keyword arguments. Pass as "explain_func_kwargs" dictionary when working with a metric class. Pass as regular kwargs when using the stnad-alone function.

method: string, optional
XAI method.

xai_lib_kwargs: dictionary, optional
Keyword arguments to be passed to the attribution function.

softmax: boolean, optional
Indicated whether softmax activation in the last layer shall be removed.

channel_first: boolean, optional
Indicates if the image dimensions are channel first, or channel last. Inferred from the input shape if None.

reduce_axes: tuple
Indicates the indices of dimensions of the output explanation array to be summed. For example, an input array of shape (8, 28, 28, 3) with keepdims=True and reduce_axes = (-1,) will return an array of shape (8, 28, 28, -1). Passing "()" will keep the original dimensions.

keepdims: boolean
Indicated if the reduced axes shall be preserved (True) or removed (False).

Returns
——-
explanation: np.ndarray
Returns np.ndarray of same shape as inputs.

quantus.functions.explanation_func.**generate_zennit_explanation**(*model*, *inputs: ndarray*, *targets: ndarray*, *device: str | None = None*, ***kwargs*) → ndarray

Generate explanation for a torch model with zennit.

Parameters

model: torch.nn.Module
A model that is used for explanation.

inputs: np.ndarray
The inputs that ought to be explained.

targets: np.ndarray
The target lables that should be used in the explanation.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments. Pass as "explain_func_kwargs" dictionary when working with a metric class. Pass as regular kwargs when using the stnad-alone function.

attributor: string, optional
XAI method.

xai_lib_kwargs: dictionary, optional
Keyword arguments to be passed to the attribution function.

softmax: boolean, optional
Indicated whether softmax activation in the last layer shall be removed.

channel_first: boolean, optional
Indicates if the image dimensions are channel first, or channel last. Inferred from the input shape if None.

reduce_axes: tuple
Indicates the indices of dimensions of the output explanation array to be summed. For example, an input array of shape (8, 28, 28, 3) with keepdims=True and reduce_axes = (-1,) will return an array of shape (8, 28, 28, -1). Passing "()" will keep the original dimensions.

keepdims: boolean
Indicated if the reduced axes shall be preserved (True) or removed (False).

Returns
——-
explanation: np.ndarray
Returns np.ndarray of same shape as inputs.

quantus.functions.explanation_func.**get_explanation**(*model*, *inputs*, *targets*, ***kwargs*)

Generate explanation array based on the type of input model and user specifications. For tensorflow models, tf.explain is used. For pytorch models, either captum or zennit is used, depending on which module is installed. If both are installed, captum is used per default. Setting the xai_lib kwarg to "zennit" uses zennit instead.

Parameters

model: torch.nn.Module, tf.keras.Model
A model that is used for explanation.

inputs: np.ndarray
The inputs that ought to be explained.

targets: np.ndarray
The target lables that should be used in the explanation.

kwargs: optional
Keyword arguments. Pass as "explain_func_kwargs" dictionary when working with a metric class. Pass as regular kwargs when using the stnad-alone function.

xai_lib: string, optional
XAI library: captum, tf-explain or zennit.

method: string, optional
XAI method (used with captum and tf-explain libraries).

attributor: string, optional
XAI method (used with zennit).

xai_lib_kwargs: dictionary, optional
Keyword arguments to be passed to the attribution function.

softmax: boolean, optional
Indicated whether softmax activation in the last layer shall be removed.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

reduce_axes: tuple
Indicates the indices of dimensions of the output explanation array to be summed. For example, an input array of shape (8, 28, 28, 3) with keepdims=True and reduce_axes = (-1,) will return an array of shape (8, 28, 28, -1). Passing "()" will keep the original dimensions.

keepdims: boolean
Indicated if the reduced axes shall be preserved (True) or removed (False).

Returns

———-

explanation: np.ndarray
Returns np.ndarray of same shape as inputs.

quantus.functions.loss_func module

This module holds a collection of loss functions i.e., ways to measure the loss between two inputs.

quantus.functions.loss_func.**mse**(*a: array*, *b: array*, ***kwargs*) → float

Calculate Mean Squared Error between two images (or explanations).

Parameters

a: np.ndarray
Array to calculate MSE with.

b: np.ndarray
Array to calculate MSE with.

kwargs: optional

Keyword arguments.

normalise_mse: boolean
Indicates whether to returned a normalised MSE calculation or not.

Returns

float:
A floating point of MSE.

quantus.functions.mosaic_func module

This module contains a collection of mosaic creation functions, i.e., group images within a grid structure.

quantus.functions.mosaic_func.**build_single_mosaic**(*mosaic_images_list: List[ndarray]*) → ndarray

Frame a list of 4 images into a 2x2 mosaic image.

Parameters

mosaic_images_list: List[np.array]
A list of four images.

Returns

mosaic: np.ndarray
The single 2x2 mosaic built from a list of images.

quantus.functions.mosaic_func.**mosaic_creation**(*images: ndarray*, *labels: ndarray*, *mosaics_per_class: int*, *seed: int | None = None*) → Tuple[Any, List[Tuple[Any, ...]], List[Tuple[Any, ...]], List[Tuple[int, ...]], List[Any]]

Build a mosaic dataset from an image dataset (images). Each mosaic corresponds to a 2x2 grid. Each one is composed by four images: two belonging to the target class and the other two are chosen randomly from the rest of the classes.

Parameters

images: np.ndarray
A np.ndarray which contains the input data.

labels: np.ndarray
A np.ndarray which contains the labels from the input data.

mosaics_per_class: integer
An integer indicating the number of mosaics per class.

seed: integer
An integer used to generate a random number (optional)..

Returns

all_mosaics: np.ndarray
a np.ndarray which contains the mosaic data

mosaic_indices_list: a List[Tuple[int, int, int, int]] which contains the image indices corresponding to the images
composing each mosaic

mosaic_labels_list (List[Tuple[Union[int, str], ...]]): a List[Tuple[Union[int, str], ...]] which contains the labels of the images composing each
mosaic. Each tuple contains four values referring to (top_left_label, top_right_label, bottom_left_label, bottom_right_label)

p_batch_list (List[Tuple[int, int, int, int]]): a List[Tuple[int, int, int, int]] which contains the positions of the target class within the mosaic.
Each tuple contains 0 and 1 values (non_target_class and target_class) referring to (top_left, top_right, bottom_left, bottom_right).

target_list (List[Union[int, str]]): a List[Union[int, str]] which contains the target class of each mosaic.

quantus.functions.norm_func module

This module contains a collection of norm functions i.e., ways to measure the norm of a input- (or explanation) vector.

quantus.functions.norm_func.**fro_norm**(*a: array*) → float

Calculate Frobenius norm for an array.

Parameters

a: np.ndarray
The array to calculate the Frobenius on.

Returns

float
The norm.

quantus.functions.norm_func.**l2_norm**(*a: array*) → float

Calculate L2 norm for an array.

Parameters

a: np.ndarray
The array to calculate the L2 on

Returns

float
The norm.

quantus.functions.norm_func.**linf_norm**(*a: array*) → float

Calculate L-inf norm for an array.

Parameters

a: np.ndarray
The array to calculate the L-inf on.

Returns

float
The norm.

quantus.functions.normalise_func module

This module provides some basic functionality to normalise and denormalise images.

quantus.functions.normalise_func.**denormalise**(*a: ndarray*, *mean: ndarray*, *std: ndarray*) → ndarray

Parameters

a: np.ndarray
the array to normalise, e.g., an image or an explanation.

mean: np.ndarray
The mean points to sample from, len(mean) = nr_channels.

std: np.ndarray
The standard deviations to sample from, len(mean) = nr_channels.

kwargs: optional
Keyword arguments.

Returns

np.ndarray
A denormalised array.

quantus.functions.normalise_func.**normalise_by_average_second_moment_estimate**(*a: ndarray*, *normalise_axes: Sequence[int] | None = None*) → ndarray

Normalise attributions by dividing the attribution map by the square-root of its average second moment estimate (that is, similar to the standard deviation, but centered around zero instead of the data mean).

This normalisation function does not normalise the attributions into a fixed range. Instead, it ensures that each score in the attribution map has an average squared distance to zero that is equal to one. This is not meant for visualisation purposes, rather it is meant to preserve a quantity that is useful for the comparison of distances between different attribution methods.

References:
1) Binder et al., (2022): "Shortcomings of Top-Down Randomization-Based Sanity Checks for Evaluations of Deep Neural Network Explanations." arXiv: https://arxiv.org/abs/2211.12486.

Parameters

a: np.ndarray
the array to normalise, e.g., an image or an explanation.

normalise_axes: optional, sequence
the axes to normalise over.

kwargs: optional
Keyword arguments.

Returns

a: np.ndarray
a normalised array.

quantus.functions.normalise_func.**normalise_by_max**(*a: ndarray*, *normalise_axes: Sequence[int] | None = None*) → ndarray

Normalise attributions by the maximum absolute value of the explanation.

Parameters

a: np.ndarray
the array to normalise, e.g., an image or an explanation.

normalise_axes: optional, sequence
the axes to normalise over.

kwargs: optional
Keyword arguments.

Returns

a: np.ndarray
a normalised array.

quantus.functions.normalise_func.**normalise_by_negative**(*a: ndarray*, *normalise_axes: Sequence[int] | None = None*) → ndarray

Normalise attributions between [-1, 1].

Parameters

a: np.ndarray
the array to normalise, e.g., an image or an explanation.

normalise_axes: optional, sequence
the axes to normalise over.

kwargs: optional
Keyword arguments.

Returns

a: np.ndarray
a normalised array.

quantus.functions.perturb_func module

This modules holds a collection of perturbation functions i.e., ways to perturb an input or an explanation.

quantus.functions.perturb_func.**baseline_replacement_by_blur**(*arr: array*, *indices: Tuple[array]*, *indexed_axes: Sequence[int]*, *blur_kernel_size: int | Sequence[int] = 15*, ***kwargs*) → array

Replace array at indices by a blurred version, performed via convolution.

Parameters

arr: np.ndarray
Array to be perturbed.

indices: int, sequence, tuple
Array-like, with a subset shape of arr.

indexed_axes: sequence

The dimensions of arr that are indexed. These need to be consecutive,
and either include the first or last dimension of array.

blur_kernel_size: int, sequence
Controls the kernel-size of that convolution (Default is 15).

kwargs: optional
Keyword arguments.

Returns

arr_perturbed: np.ndarray
The array which some of its indices have been perturbed.

quantus.functions.perturb_func.**baseline_replacement_by_indices**(*arr: array*, *indices: Tuple[slice, ...]*, *indexed_axes: Sequence[int]*, *perturb_baseline: float | int | str | array*, ***kwargs*) → array

Replace indices in an array by a given baseline.

Parameters

arr: np.ndarray
Array to be perturbed.

indices: int, sequence, tuple
Array-like, with a subset shape of arr.

indexed_axes: sequence

The dimensions of arr that are indexed. These need to be consecutive,
and either include the first or last dimension of array.

perturb_baseline: float, int, str, np.ndarray
The baseline values to replace arr at indices with.

kwargs: optional
Keyword arguments.

Returns

arr_perturbed: np.ndarray
The array which some of its indices have been perturbed.

quantus.functions.perturb_func.**baseline_replacement_by_shift**(*arr: array*, *indices: Tuple[slice, ...]*, *indexed_axes: Sequence[int]*, *input_shift: float | int | str | array*, ***kwargs*) → array

Shift values at indices in an image.

Parameters

arr: np.ndarray
Array to be perturbed.

indices: int, sequence, tuple
Array-like, with a subset shape of arr.

indexed_axes: sequence
The dimensions of arr that are indexed. These need to be consecutive, and either include the first or last dimension of array.

input_shift: float, int, str, np.ndarray
Value to shift arr at indices with.

kwargs: optional
Keyword arguments.

Returns

arr_perturbed: np.ndarray
The array which some of its indices have been perturbed.

quantus.functions.perturb_func.**gaussian_noise**(*arr: array*, *indices: Tuple[slice, ...]*, *indexed_axes: Sequence[int]*, *perturb_mean: float = 0.0*, *perturb_std: float = 0.01*, ***kwargs*) → array

Add gaussian noise to the input at indices.

Parameters

arr: np.ndarray
Array to be perturbed.

indices: int, sequence, tuple
Array-like, with a subset shape of arr.

indexed_axes: sequence
The dimensions of arr that are indexed. These need to be consecutive, and either include the first or last dimension of array.

perturb_mean (float):
The mean for gaussian noise.

perturb_std (float):
The standard deviation for gaussian noise.

kwargs: optional
Keyword arguments.

Returns

arr_perturbed: np.ndarray
The array which some of its indices have been perturbed.

quantus.functions.perturb_func.**no_perturbation**(*arr: array*, ***kwargs*) → array

Apply no perturbation to input.

Parameters

arr: np.ndarray
Array to be perturbed.

kwargs: optional
Keyword arguments.

Returns

arr: np.ndarray
Array unperturbed.

quantus.functions.perturb_func.**noisy_linear_imputation**(*arr: array*, *indices: Sequence[int] | Tuple[array]*, *noise: float = 0.01*, ***kwargs*) → array

Calculates noisy linear imputation for the given array and a list of indices indicating which elements are not included in the mask.

Adapted from: https://github.com/tleemann/road_evaluation.

Parameters

arr: np.ndarray
Array to be perturbed.

indices: int, sequence, tuple
Array-like, with a subset shape of arr.

indexed_axes: sequence

The dimensions of arr that are indexed. These need to be consecutive,
and either include the first or last dimension of array.

noise: float
The amount of noise added.

kwargs: optional
Keyword arguments.

Returns

arr_perturbed: np.ndarray
The array which some of its indices have been perturbed.

quantus.functions.perturb_func.**perturb_batch**(*perturb_func: Callable*, *arr: ndarray*, *indices: ndarray | None = None*, *inplace: bool = False*, ***kwargs*) → ndarray | None

Use a perturb function and make perturbation on the full batch.

Parameters

perturb_func: callable

Input perturbation function.

arr: np.ndarray
Array to be perturbed.

indices: int, sequence, tuple
Array-like, with a subset shape of arr.

inplace: boolean
Indicates if the array should be copied or not.

kwargs: optional
Keyword arguments.

Returns

None, array

quantus.functions.perturb_func.**rotation**(*arr: array*, *perturb_angle: float = 10*, ***kwargs*) → array

Rotate array by some given angle, assumes image type data and channel first layout.

Parameters

arr: np.ndarray
Array to be perturbed.

perturb_angle: float
The angle to perturb.

kwargs: optional
Keyword arguments.

Returns

arr_perturbed: np.ndarray
The array which some of its indices have been perturbed.

quantus.functions.perturb_func.**translation_x_direction**(*arr: array*, *perturb_baseline: float | int | str | array*, *perturb_dx: int = 10*, ***kwargs*) → array

Translate array by some given value in the x-direction, assumes image type data and channel first layout.

Parameters

arr: np.ndarray
Array to be perturbed.

perturb_baseline: float, int, str, np.ndarray
The baseline values to replace arr at indices with.

perturb_dx: integer
The translation length in features, e.g., pixels.

kwargs: optional
Keyword arguments.

Returns

arr_perturbed: np.ndarray
The array which some of its indices have been perturbed.

quantus.functions.perturb_func.**translation_y_direction**(*arr: array*, *perturb_baseline: float | int | str | array*, *perturb_dy: int = 10*, ***kwargs*) → array

Translate array by some given value in the y-direction, assumes image type data and channel first layout.

Parameters

arr: np.ndarray
Array to be perturbed.

perturb_baseline: float, int, str, np.ndarray
The baseline values to replace arr at indices with.

perturb_dy: integer
The translation length in features, e.g., pixels.

kwargs: optional
Keyword arguments.

Returns

arr_perturbed: np.ndarray
The array which some of its indices have been perturbed.

quantus.functions.perturb_func.**uniform_noise**(*arr: array*, *indices: Tuple[slice, ...]*, *indexed_axes: Sequence[int]*, *lower_bound: float = 0.02*, *upper_bound: None | float = None*, ***kwargs*) → array

Add noise to the input at indices as sampled uniformly random from [-lower_bound, lower_bound]. if upper_bound is None, and [lower_bound, upper_bound] otherwise.

Parameters

arr: np.ndarray
Array to be perturbed.

indices: int, sequence, tuple
Array-like, with a subset shape of arr.

indexed_axes: sequence

The dimensions of arr that are indexed. These need to be consecutive,
and either include the first or last dimension of array.

lower_bound: float
The lower bound for uniform sampling.

upper_bound: float, optional
The upper bound for uniform sampling.

kwargs: optional
Keyword arguments.

Returns

arr_perturbed: np.ndarray
The array which some of its indices have been perturbed.

quantus.functions.similarity_func module

This module holds a collection of similarity functions i.e., ways to measure the distance between two inputs (or explanations).

quantus.functions.similarity_func.**abs_difference**(*a: array*, *b: array*, ***kwargs*) → float

Calculate the mean of the absolute differences between two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.functions.similarity_func.**correlation_kendall_tau**(*a: array*, *b: array*, ***kwargs*) → float

Calculate Kendall Tau correlation of two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.functions.similarity_func.**correlation_pearson**(*a: array*, *b: array*, ***kwargs*) → float

Calculate Pearson correlation of two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.functions.similarity_func.**correlation_spearman**(*a: array*, *b: array*, ***kwargs*) → float

Calculate Spearman rank of two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.functions.similarity_func.**cosine**(*a: array*, *b: array*, ***kwargs*) → float

Calculate Cosine of two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.functions.similarity_func.**difference**(*a: array*, *b: array*, ***kwargs*) → array

Calculate the difference between two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

np.array
The difference in each element.

quantus.functions.similarity_func.**distance_chebyshev**(*a: array*, *b: array*, ***kwargs*) → float

Calculate Chebyshev distance of two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.functions.similarity_func.**distance_euclidean**(*a: array*, *b: array*, ***kwargs*) → float

Calculate Euclidean distance of two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.functions.similarity_func.**distance_manhattan**(*a: array*, *b: array*, ***kwargs*) → float

Calculate Manhattan distance of two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.functions.similarity_func.**lipschitz_constant**(*a: array*, *b: array*, *c: array | None*, *d: array | None*, ***kwargs*) → float

Calculate non-negative local Lipschitz abs(||a-b||/||c-d||), where a,b can be f(x) or a(x) and c,d is x.

For numerical stability, a small value is added to division.

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

c: np.ndarray
The third array to use for similarity scoring.

d: np.ndarray
The fourth array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.functions.similarity_func.**squared_difference**(*a: array*, *b: array*, ***kwargs*) → float

Calculate the sqaured differences between two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.functions.similarity_func.**ssim**(*a: array*, *b: array*, ***kwargs*) → float

Calculate Structural Similarity Index Measure of two images (or explanations).

Parameters

a: np.ndarray
The first array to use for similarity scoring.

b: np.ndarray
The second array to use for similarity scoring.

kwargs: optional
Keyword arguments.

Returns

float
The similarity score.

quantus.helpers package

Subpackages

quantus.helpers.model package

Submodules

quantus.helpers.model.model_interface module

This model implements the basics for the ModelInterface class.

class quantus.helpers.model.model_interface.**ModelInterface**(*model: M*, *channel_first: bool | None = True*, *softmax: bool = False*, *model_predict_kwargs: Dict[str, Any] | None = None*)

Bases: ABC, Generic[M]

Base ModelInterface for torch and tensorflow models.

Attributes

get_ml_framework_name
Identify the framework of the underlying model (PyTorch or TensorFlow).

random_layer_generator_length
Count number of randomisable layers for *Model Parameter Randomisation*.

Methods

add_mean_shift_to_first_layer(input_shift, shape)	Consider the first layer neuron before non-linearity: z = w^T * x1 + b1.
get_hidden_representations(x[, layer_names, ...])	Compute the model's internal representation of input x.
get_model()	Get the original torch/tf model.
get_random_layer_generator([order, seed])	In every iteration yields a copy of the model with one additional layer's parameters randomized.
get_softmax_arg_model()	Returns model with last layer adjusted accordingly to softmax argument.
predict(x, **kwargs)	Predict on the given input.
shape_input(x, shape[, channel_first, batched])	Reshape input into model expected input.
state_dict()	Get a dictionary of the model's learnable parameters.

__init__(*model: M*, *channel_first: bool | None = True*, *softmax: bool = False*, *model_predict_kwargs: Dict[str, Any] | None = None*)

Initialisation of ModelInterface class.

Parameters

model: torch.nn.Module, tf.keras.Model
A model this will be wrapped in the ModelInterface:

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

abstract add_mean_shift_to_first_layer(*input_shift: int | float*, *shape: tuple*)

Consider the first layer neuron before non-linearity: z = w^T * x1 + b1. We update the bias b1 to b2:= b1 - w^T * m. The operation is necessary for Input Invariance metric.

Parameters

input_shift: Union[int, float]
Shift to be applied.

shape: tuple
Model input shape.

Returns

new_model: torch.nn
The resulting model with a shifted first layer.

abstract get_hidden_representations(*x: ndarray*, *layer_names: List[str] | None = None*, *layer_indices: List[int] | None = None*) → ndarray

Compute the model's internal representation of input x. In practice, this means, executing a forward pass and then, capturing the output of layers (of interest). As the exact definition of "internal model representation" is left out in the original paper (see: https://arxiv.org/pdf/2203.06877.pdf), we make the implementation flexible. It is up to the user whether all layers are used, or specific ones should be selected. The user can therefore select a layer by providing 'layer_names' (exclusive) or 'layer_indices'.

Parameters

x: np.ndarray
4D tensor, a batch of input datapoints

layer_names: List[str]
List with names of layers, from which output should be captured.

layer_indices: List[int]
List with indices of layers, from which output should be captured. Intended to use in case, when layer names are not unique, or unknown.

Returns

L: np.ndarray
2D tensor with shape (batch_size, None)

property get_ml_framework_name: str

Identify the framework of the underlying model (PyTorch or TensorFlow).

Returns

str
A string indicating the framework ('PyTorch', 'TensorFlow', or 'Unknown').

abstract get_model()

Get the original torch/tf model.

abstract get_random_layer_generator(*order: str = 'top_down'*, *seed: int = 42*) → Generator[Tuple[str, M], None, None]

In every iteration yields a copy of the model with one additional layer's parameters randomized. For cascading randomization, set order (str) to 'top_down'. For independent randomization, set it to 'independent'. For bottom-up order, set it to 'bottom_up'.

abstract get_softmax_arg_model()

Returns model with last layer adjusted accordingly to softmax argument. If the original model has softmax activation as the last layer and softmax=false, the layer is removed.

abstract predict(*x: array*, ***kwargs*)

Predict on the given input.

Parameters

x: np.ndarray
A given input that the wrapped model predicts on.

kwargs: optional
Keyword arguments.

`abstract property random_layer_generator_length: int`

Count number of randomisable layers for *Model Parameter Randomisation*. This property is needed to avoid *len(model.get_random_layer_generator())*, because meterializing bigger models *num_layers* times in memory at ones has shown to cause OOM errors.

Returns

n:
Number of layers in model, which can be randomised.

`abstract shape_input`(*x: array*, *shape: Tuple[int, ...]*, *channel_first: bool | None = None*, *batched: bool = False*)

Reshape input into model expected input.

Parameters

x: np.ndarray
A given input that is shaped.

shape: Tuple[int...]
The shape of the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

`abstract state_dict`()

Get a dictionary of the model's learnable parameters.

quantus.helpers.model.models module

This module contains example LeNets and other simple architectures for PyTorch and tensorflow.

quantus.helpers.model.pytorch_model module

quantus.helpers.model.tf_model module

Submodules

quantus.helpers.asserts module

This module holds a collection of asserts functionality that is used across the Quantus library to avoid undefined behaviour.

`quantus.helpers.asserts.assert_attributions`(*x_batch: array*, *a_batch: array*) → None

Asserts on attributions, assumes channel first layout.

Parameters

x_batch: np.ndarray
The batch of input to compare the shape of the attributions with.

a_batch: np.ndarray
The batch of attributions.

Returns

None

quantus.helpers.asserts.**assert_attributions_order**(*order: str*) → None

Assert that order is in pre-defined list.

Parameters

order: string
The different orders that attributions could be ranked in.

Returns

None

quantus.helpers.asserts.**assert_explain_func**(*explain_func: Callable*) → None

Asser thta the explanation function is a callable.

Parameters

explain_func: callable
An plot function input, asusmed to be a Callable.

Returns

None

quantus.helpers.asserts.**assert_features_in_step**(*features_in_step: int*, *input_shape: Tuple[int, ...]*) → None

Assert that features in step is compatible with the image size.

Parameters

features_in_step: integer
The number of features e.g., pixels included in each iteration.

input_shape: Tuple[int...]
The shape of the input.

Returns

None

quantus.helpers.asserts.**assert_indexed_axes**(*arr: array*, *indexed_axes: Sequence[int]*) → None

Checks that indexed_axes fits the given array.

Parameters

arr: np.ndarray
A given array that we want to check indexed_axes against.

indexed_axes: sequence
The sequence with indices, with axes.

Returns

None

quantus.helpers.asserts.**assert_layer_order**(*layer_order: str*) → None

Assert that layer order is in pre-defined list.

Parameters

layer_order: string
The various ways that a model's weights of a layer can be randomised.

Returns

None

quantus.helpers.asserts.**assert_nr_segments**(*nr_segments: int*) → None

Assert that the number of segments given the segmentation algorithm is more than one.

Parameters

nr_segments: integer
The number of segments that the segmentaito algorithm produced.

Returns

None

quantus.helpers.asserts.**assert_patch_size**(*patch_size: int | tuple*, *shape: Tuple[int, ...]*) → None

Assert that patch size is compatible with given image shape.

Parameters

patch_size: integer
The size of the patch_size, assumed to tbe squared.

input_shape: Tuple[int...]
the shape of the input.

Returns

None

quantus.helpers.asserts.**assert_plot_func**(*plot_func: Callable*) → None

Assert that the plot function is a callable.

Parameters

plot_func: callable
An plot function input, asusmed to be a Callable.

Returns

None

quantus.helpers.asserts.**assert_segmentations**(*x_batch: array*, *s_batch: array*) → None

Asserts on segmentations, assumes channel first layout.

Parameters

x_batch: np.ndarray
The batch of input to compare the shape of the attributions with.

s_batch: np.ndarray
The batch of segmentations.

Returns

None

quantus.helpers.asserts.**assert_value_smaller_than_input_size**(*x: ndarray*, *value: int*, *value_name: str*) → None

Checks if value is smaller than input size, assumes batch and channel first dimension.

Parameters

x: np.ndarray
The input to check the value against.

value: integer
The value that must be smaller than input size.

value_name: string
The hyperparameter to check, e.g., "k" for TopKIntersection.

Returns

None

quantus.helpers.constants module

This module contains constants and simple methods to retreive the available metrics, perturbation-, similarity-, normalisation- functions and explanation methods in Quantus.

quantus.helpers.constants.**available_categories**() → List[str]

Retrieve the available metric categories in Quantus.

Returns

List[str]
With the available metric categories in Quantus.

quantus.helpers.constants.**available_methods_captum**() → List[str]

Retrieve the available explanation methods in Quantus.

Returns

List[str]
With the available explanation methods in Quantus.

quantus.helpers.constants.**available_methods_tf_explain**() → List[str]

Retrieve the available explanation methods in Quantus.

Returns

List[str]
With the available explanation methods in Quantus.

quantus.helpers.constants.**available_metrics**() → Dict[str, List[str]]

Retrieve the available metrics in Quantus.

Returns

Dict[str, str]
With the available metrics, under each category in Quantus.

quantus.helpers.constants.**available_normalisation_functions**() → List[str]

Retrieve the available normalisation functions in Quantus.

Returns

List[str]
With the available normalisation functions in Quantus.

quantus.helpers.constants.**available_perturbation_functions**() → List[str]

Retrieve the available perturbation functions in Quantus.

Returns

List[str]
With the available perturbation functions in Quantus.

quantus.helpers.constants.**available_similarity_functions**() → List[str]

Retrieve the available similarity functions in Quantus.

Returns

List[str]
With the available similarity functions in Quantus.

quantus.helpers.enums module

class quantus.helpers.enums.**DataType**(*value*)

Bases: Enum

This enum represents the different types of data that a metric implementation currently supports.

- IMAGE: Represents image data.
- TABULAR: Represents tabular data.
- TEXT: Represents text data.

IMAGE = 'image'

TABULAR = 'tabular'

TEXT = 'text'

TIMESERIES = 'time-series'

class quantus.helpers.enums.**EvaluationCategory**(*value*)

Bases: Enum

This enum represents different categories of explanation quality for XAI algorithms.

- FAITHFULNESS: Indicates how well the explanation reflects the true features used by the model.
- ROBUSTNESS: Represents the degree to which the explanation remains consistent under small perturbations in the input.
- RANDOMISATION: Measures the quality of the explanation in terms of difference in explanation when randomness is introduced.
- COMPLEXITY: Refers to how easy it is to understand the explanation. Lower complexity is usually better.
- LOCALISATION: Refers to how consistently the explanation points out the parts of the input as defined in a ground-truth segmentation mask.
- AXIOMATIC: Represents the quality of the explanation in terms of well-defined axioms.

AXIOMATIC = 'Axiomatic'

COMPLEXITY = 'Complexity'

FAITHFULNESS = 'Faithfulness'

LOCALISATION = 'Localisation'

NONE = 'None'

RANDOMISATION = 'Randomisation'

ROBUSTNESS = 'Robustness'

class quantus.helpers.enums.**ModelType**(*value*)

Bases: Enum

This enum represents the different types of models that a metric can work with.

- TORCH: Represents PyTorch models.
- TF: Represents TensorFlow models.

TF = 'tensorflow'

TORCH = 'torch'

class quantus.helpers.enums.**ScoreDirection**(*value*)

Bases: Enum

This enum represents the direction that the score of a metric should go in for better results.

- HIGHER: Higher scores are better.
- LOWER: Lower scores are better.

HIGHER = 'higher'

LOWER = 'lower'

quantus.helpers.perturbation_utils module

quantus.helpers.perturbation_utils.**changed_prediction_indices**(*model: ModelInterface*, *x_batch: np.ndarray*, *x_perturbed: np.ndarray*, *return_nan_when_prediction_changes: bool*) → List[int]

Find indices in batch, for which predicted label has changed after applying perturbation. If metric *return_nan_when_prediction_changes* is False, will return empty list.

Parameters

return_nan_when_prediction_changes:
Instance attribute of perturbation metrics.

model:
x_batch:
Batch of original inputs provided by user.

x_perturbed:
Batch of inputs after applying perturbation.

Returns

changed_idx:
List of indices in batch, for which predicted label has changed afer.

quantus.helpers.perturbation_utils.**make_changed_prediction_indices_func**(*return_nan_when_prediction_changes: bool*) → Callable[[*ModelInterface*, np.ndarray, np.ndarray], List[int]]

A utility function to improve static analysis.

quantus.helpers.perturbation_utils.**make_perturb_func**(*perturb_func: PerturbFunc*, *perturb_func_kwargs: Mapping[str, ...] | None*, ***kwargs*) → PerturbFunc | functools.partial

A utility function to save few lines of code during perturbation metric initialization.

quantus.helpers.plotting module

This module provides some plotting functionality.

quantus.helpers.plotting.**plot_focus**(*results: Dict[str, List[float]]*, **args*, ***kwargs*) → None

Plot the Focus experiment as done in the paper:

References:
1) Arias-Duart, Anna, et al. 'Focus! Rating XAI Methods and Finding Biases. arXiv:2109.15035 (2022)"

Parameters

results: dict
A dictionary with the Focus scores obtained using different feature attribution methods.

args: optional
Arguments.

kwargs: optional
Keyword arguments.

Returns

None

quantus.helpers.plotting.**plot_model_parameter_randomisation_experiment**(*results: Dict[str, dict]*, *methods=None*, **args*, ***kwargs*) → None

Plot the model parameter randomisation experiment as done in paper:

References:
1) Adebayo, J., Gilmer, J., Muelly, M., Goodfellow, I., Hardt, M., and Kim, B. "Sanity Checks for Saliency Maps." arXiv preprint, arXiv:1810.073292v3 (2018)

Parameters

results: list, dict
The results fromm the Selectivity experiment(s).

args: optional
Arguments.

kwargs: optional
Keyword arguments.

Returns

None

quantus.helpers.plotting.**plot_pixel_flipping_experiment**(*y_batch: ndarray*, *scores: List[Any]*, *single_class: int | None = None*, **args*, ***kwargs*) → None

Plot the pixel-flipping experiment as done in paper:

References:
1) Bach, Sebastian, et al. "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation." PloS one 10.7 (2015): e0130140.

Parameters

y_batch: np.ndarray
The list of true labels.

scores: list
The list of evalution scores.

single_class: integer, optional
An integer to specify the label to plot.

args: optional
Arguments.

kwargs: optional
Keyword arguments.

Returns

None

quantus.helpers.plotting.**plot_region_perturbation_experiment**(*results: Dict[str, List[Any]]*, **args*, ***kwargs*) → None

Plot the region perturbation experiment as done in paper:

References:
1) Samek, Wojciech, et al. "Evaluating the visualization of what a deep neural network has learned." IEEE transactions on neural networks and learning systems 28.11 (2016): 2660-2673.

Parameters

results: list, dict
The results fromm the Selectivity experiment(s).

args: optional
Arguments.

kwargs: optional
Keyword arguments.

Returns

None

quantus.helpers.plotting.**plot_selectivity_experiment**(*results: Dict[str, List[Any]]*, **args*, ***kwargs*) → None

Plot the selectivity experiment as done in paper:

References:
1) Montavon, Grégoire, Wojciech Samek, and Klaus-Robert Müller. "Methods for interpreting and understanding deep neural networks." Digital Signal Processing 73 (2018): 1-15.

Parameters

results: list, dict
The results fromm the Selectivity experiment(s).

args: optional
Arguments.

kwargs: optional
Keyword arguments.

Returns

None

quantus.helpers.plotting.**plot_sensitivity_n_experiment**(*results: List[float] | Dict[str, List[float]]*, **args*, ***kwargs*) → None

Plot the sensitivity n experiment as done in paper:

References:
1) Ancona, Marco, et al. "Towards better understanding of gradient-based attribution methods for deep neural networks." arXiv preprint arXiv:1711.06104 (2017).

Parameters

results: list, dict
The results fromm the Selectivity experiment(s).

args: optional
Arguments.

kwargs: optional
Keyword arguments.

Returns

None

quantus.helpers.utils module

This module contains the utils functions of the library.

quantus.helpers.utils.**blur_at_indices**(*arr: array*, *kernel: array*, *indices: int | Sequence[int] | Tuple[array] | Tuple[slice, ...]*, *indexed_axes: Sequence[int]*) → array

Creates a version of arr that is blurred at indices.

Parameters

arr: np.ndarray
Array to be perturbed.

kernel: np.ndarray
Kernel used for blurring.

indices: int, sequence, tuple
Array-like, with a subset shape of arr.

indexed_axes: sequence

The dimensions of arr that are indexed. These need to be consecutive and either
include the first or last dimension of array.

Returns

np.ndarray
A version of arr that is blurred at indices.

quantus.helpers.utils.**calculate_auc**(*values: array*, *dx: int = 1*)

Calculate area under the curve using the composite trapezoidal rule.

Parameters

values: np.ndarray
Input array.

dx: integer
The spacing between sample points. The default is 1.

Returns

np.ndarray
Definite integral of values.

quantus.helpers.utils.**create_patch_slice**(*patch_size: int | Sequence[int]*, *coords: Sequence[int]*) → Tuple[slice, ...]

Create a patch slice from patch size and coordinates.

Parameters

patch_size: int, sequence
One- or multidimensional patch size.

coords: sequence
Coordinates for creating patches.

Returns

tuple
Patches at all provided coordinates.

quantus.helpers.utils.**expand_attribution_channel**(*a_batch: ndarray*, *x_batch: ndarray*)

Expand additional channel dimension(s) for attributions if needed.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

a_batch: np.ndarray
An array which contains pre-computed attributions i.e., explanations.

Returns

np.ndarray
A x_batch with dimensions matching those of a_batch.

quantus.helpers.utils.**expand_indices**(*arr: array*, *indices: int | Sequence[int] | Tuple[array] | Tuple[slice, ...]*, *indexed_axes: Sequence[int]*) → Tuple

Expands indices to fit array shape. Returns expanded indices.

–> if indices are a sequence of ints, they are interpreted as indices to the flattened arr,
and subsequently expanded

–> if indices contains only slices and 1D sequences for arr, everything is interpreted as slices –> if indices contains already expanded indices, they are returned as is

Parameters

arr: np.ndarray
The input to the expanded.

indices: int, sequence, tuple
A list of indices.

indexed_axes: sequence
Refers to all axes that are not indexed by slice(None).

Returns

tuple
Expanded indices.

quantus.helpers.utils.**filter_compatible_patch_sizes**(*perturb_patch_sizes: list*, *img_size: int*) → list

Remove patch sizes that are not compatible with input size.

Parameters

perturb_patch_sizes: list:
Patch sizes for perturbation.

img_size: integer
A single dimension of an image array.

Returns

list:
All integers within perturb_patch_sizes which are compatible with the image.

quantus.helpers.utils.**get_baseline_dict**(*arr: ndarray*, *patch: ndarray | None = None*, ***kwargs*) → dict

Make a dictionary of baseline approaches depending on the input x (or patch of input).

Parameters

arr: np.ndarray
CxWxH image array used to calculate baseline values, i.e. for "mean", "black" and "white" methods.

patch: np.ndarray, optional
CxWxH patch array to calculate baseline values, necessary for "neighbourhood_mean" and "neighbourhood_random_min_max" methods.

kwargs: optional
Keyword arguments..

Returns

fill_dict: dict
Maps all available baseline methods to baseline values.

quantus.helpers.utils.**get_baseline_value**(*value: float | int | str | array*, *arr: ndarray*, *return_shape: Tuple*, *patch: ndarray | None = None*, ***kwargs*) → array

Get the baseline value to fill the array with, in the shape of return_shape.

Parameters

value: float, int, str, np.ndarray
Either the value (float, int) to fill the array with, a method (string) used to construct baseline array ("mean", "uniform", "black", "white", "neighbourhood_mean" or "neighbourhood_random_min_max"), or the array (np.array) to be returned.

arr: np.ndarray
CxWxH image array used to calculate baseline values, i.e. for "mean", "black" and "white" methods.

return_shape: tuple
CxWxH shape to be returned.

patch: np.ndarray, optional
CxWxH patch array to calculate baseline values. Necessary for "neighbourhood_mean" and "neighbourhood_random_min_max" methods.

kwargs: optional
Keyword arguments.

Returns

np.ndarray
Baseline array in return_shape.

quantus.helpers.utils.**get_features_in_step**(*max_steps_per_input: int*, *input_shape: Tuple[int, ...]*)

Get the number of features in the iteration.

Parameters

max_steps_per_input: integer
The number of repeated iterations on an image.

input_shape: tuple
Input shape.

Returns

float
Product of the input shape divided by the maximum number of steps.

quantus.helpers.utils.**get_leftover_shape**(*arr: array*, *axes: Sequence[int]*) → Tuple

Gets the shape of the arr dimensions not included in axes.

Parameters

arr: np.ndarray
The input to the expanded.

axes: sequence
A sequence of ints containing the axes.

Returns

leftover_shape: tuple
The leftover shape.

quantus.helpers.utils.**get_name**(*name: str*)

Get the name of the Metric class object.

Parameters

name: string
A metric name.

Returns

name: string
A cleaned version of the Metric name.

quantus.helpers.utils.**get_nr_patches**(*patch_size: int | Sequence[int]*, *shape: Tuple[int, ...]*, *overlap: bool = False*) → int

Get number of patches for given shape.

Parameters

patch_size: int, sequence
One- or multidimensional patch size.

shape (shape: Tuple[int, ...]): The image shape.
overlap: boolean
Indicates whether overlapping patches is used or not.

Returns

integer
Number of patches that fit into the image.

quantus.helpers.utils.**get_superpixel_segments**(*img: ndarray*, *segmentation_method: str*) → ndarray

Given an image, return segments or so-called 'super-pixels' segments i.e., an 2D mask with segment labels.

Parameters

img: np.ndarray
CxWxH image array.

segmentation_method: string
Indicates the segmentation method, i.e. "slic" or "felzenszwalb".

Returns

img: np.ndarray
CxWxH segmented image array.

quantus.helpers.utils.**get_wrapped_model**(*model*, *channel_first: bool*, *softmax: bool*, *device: str | None = None*, *model_predict_kwargs: Dict[str, Any] | None = None*) → *ModelInterface*

Identifies the type of a model object and wraps the model in an appropriate interface.

Parameters

model: torch.nn.Module, tf.keras.Model
A model this will be wrapped in the ModelInterface:

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

Returns

model: ModelInterface
A wrapped ModelInterface model.

quantus.helpers.utils.**identity**(*x: T*) → T

quantus.helpers.utils.**infer_attribution_axes**(*a_batch: ndarray*, *x_batch: ndarray*) → Sequence[int]

Infers the axes in x_batch that are covered by a_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

a_batch: np.ndarray
An array which contains pre-computed attributions i.e., explanations.

Returns

np.ndarray
The axes inferred.

quantus.helpers.utils.**infer_channel_first**(*x: array*) → bool

Infer if the channels are first.

Assumes:

For 1D input:
nr_channels < sequence_length

For 2D input:
nr_channels < img_width and nr_channels < img_height

For higher dimensional input an error is raised. For input in n_features x n_batches format True is returned (no channel).

Parameters

x: np.ndarray
Input image.

Returns

For 1D input:
True for input shape (nr_batch, nr_features).

For 2D input:
True if input shape is (nr_batch, nr_channels, sequence_length). False if input shape is (nr_batch, sequence_length, nr_channels). An error is raised if the two last dimensions are equal.

For 3D input:
True if input shape is (nr_batch, nr_channels, img_width, img_height). False if input shape is (nr_batch, img_width, img_height, nr_channels). An error is raised if the three last dimensions are equal.

quantus.helpers.utils.**make_channel_first**(*x: array*, *channel_first: bool = False*)

Reshape batch to channel first.

Parameters

x: np.ndarray
The input image.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

Returns

np.ndarray
Image in CxWxH format.

quantus.helpers.utils.**make_channel_last**(*x: array*, *channel_first: bool = True*)

Reshape batch to channel last.

Parameters

x: np.ndarray
The input image.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

Returns

np.ndarray
Image in WxHxC format.

quantus.helpers.utils.**offset_coordinates**(*indices: list | Sequence[int] | Tuple[Any]*, *offset: tuple*, *img_shape: tuple*)

Checks if offset coordinates are within the image frame.

Adapted from: https://github.com/tleemann/road_evaluation.

Parameters

indices: list
List of indices to be offset.

offset: tuple
Offset for the coordinates, e.g. offset (1,1) adds 1 to both coordinates.

img_shape: tuple
Image shape in (channels, height, width) format.

Returns

list
Offset coordinates for valid indices and the list of booleans which identifies valid ids.

quantus.helpers.warn module

This modules holds a collection of perturbation functions i.e., ways to perturb an input or an explanation.

quantus.helpers.warn.**check_kwargs**(*kwargs*)

Check that no additional kwargs are passed, i.e. the kwargs dict is empty. Raises an exception with helpful suggestions to fix the issue.

Parameters

kwargs: optional
Keyword arguments.

Returns

None

quantus.helpers.warn.**deprecation_warnings**(*kwargs: dict*) → None

Run deprecation warnings.

Parameters

kwargs: optional
Keyword arguments.

Returns

None

quantus.helpers.warn.**warn_absolute_operation**(*word: str = ''*) → None

Warn if an absolute operation is applied, where the metric is defined otherwise.

Parameters

word: string
A string for which is '' or 'not '.

Returns

None

quantus.helpers.warn.**warn_different_array_lengths**() → None

Warn if the array lengths are different, for plotting.

Returns

None

quantus.helpers.warn.**warn_empty_segmentation**() → None

Warn if the segmentation mask is empty.

Returns

None

quantus.helpers.warn.**warn_iterations_exceed_patch_number**(*n_iterations: int*, *n_patches: int*) → None

Warn if the number of non-overlapping patches is lower than the number of iterations specified for this metric.

Parameters

n_iterations: integer
The number of iterations specified in the metric.

n_patches: integer
The number of patches specified in the metric.

Returns

None

quantus.helpers.warn.**warn_max_size**() → None

Warns if the ratio is smaller than the maximum size, for attribution_localisaiton metric. Returns ——- None

quantus.helpers.warn.**warn_noise_zero**(*noise: float*) → None

Warn if noise is zero.

Parameters

noise: float
The amount of noise.

Returns

None

quantus.helpers.warn.**warn_normalise_operation**(*word: str = ''*) → None

Warn if a normalisation operation is applied, where the metric is defined otherwise.

Parameters

word: string
A string for which is '' or 'not '.

Returns

None

quantus.helpers.warn.**warn_parameterisation**(*metric_name: str = 'Metric'*, *sensitive_params: str = 'X, Y and Z.'*, *data_domain_applicability: str = ''*, *citation: str = 'INSERT CITATION'*)

Warn the parameterisation of the metric.

Parameters

metric_name: string
The metric name.

sensitive_params: string
The sensitive parameters of the metric.

data_domain_applicability string
The applicability when it comes to data domains, default = "".

citation: string
The citation.

Returns

None

quantus.helpers.warn.**warn_perturbation_caused_no_change**(*x: ndarray*, *x_perturbed: ndarray*) → None

Warn that perturbation applied to input caused no change so that input and perturbed input is the same.

Parameters

x: np.ndarray
The original input that is considered unperturbed.

x_perturbed: np.ndarray
The perturbed input.

Returns

None

quantus.helpers.warn.**warn_segmentation**(*inside_attribution: float*, *total_attribution: float*) → None

Warn if the inside explanation is greater than total explanation.

Parameters

inside_attribution: float
The size of inside attribution.

total_attribution: float
The size of total attribution.

Returns

None

quantus.metrics package

Subpackages

quantus.metrics.axiomatic package

Submodules

quantus.metrics.axiomatic.completeness module

This module contains the implementation of the Completeness metric.

final class quantus.metrics.axiomatic.completeness.**Completeness**(*abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *output_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func: Callable | None = None*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Completeness test by Sundararajan et al., 2017, also referred to as Summation to Delta by Shrikumar et al., 2017 and Conservation by Montavon et al., 2018.

Attribution completeness asks that the total attribution is proportional to the explainable evidence at the output/ or some function of the model output. Or, that the attributions add up to the difference between the model output F at the input x and the baseline b.

Assumptions:

- This implementation does completeness test against logits, not softmax.

References:

1) Completeness - Mukund Sundararajan et al.: "Axiomatic attribution for deep networks." International Conference on Machine Learning. PMLR, 2017. 2) Summation to delta - Avanti Shrikumar et al.: "Learning important features through propagating activation differences." International Conference on Machine Learning. PMLR, 2017. 3) Conservation - Grégoire Montavon et al.: "Methods for interpreting and understanding deep neural networks." Digital Signal Processing 73 (2018): 1-15.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *output_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func: Callable | None = None*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable

Attribution normalisation function applied in case normalise=True.
If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

output_func: callable

Function applied to the difference between the model output at the input and the
baseline before metric calculation. If output_func=None, the default value is used, default=lambda x: x.

perturb_baseline: string

Indicates the type of baseline: "mean", "random", "uniform", "black" or "white",
default="black".

perturb_func: callable

Input perturbation function. If None, the default value is used,
default=baseline_replacement_by_indices.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction, default=False.

return_aggregate: boolean
Indicates if an aggregated score should be produced over all instances.

aggregate_func: callable
A Callable to aggregate the scores per instance to one float.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[bool]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInterface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

evaluate_instance(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → bool

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

score: boolean
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Axiomatic'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Completeness'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.axiomatic.input_invariance module

This module contains the implementation of the Input Invariance metric.

final class quantus.metrics.axiomatic.input_invariance.**InputInvariance**(*abs: bool = False*, *normalise: bool = False*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *input_shift: int | float = -1*, *perturb_func: Callable | None = None*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Completeness test by Kindermans et al., 2017.

To test for input invariance, we add a constant shift to the input data and a mean shift to the model bias, so that the output of the original model on the original data is equal to the output of the changed model on the shifted data. The metric returns True if batch attributions stayed unchanged too. Currently only supporting constant values for the shift.

References:
Pieter-Jan Kindermans et al.: "The (Un)reliability of Saliency Methods." Explainable AI (2019): 267-280

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.

- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch, a_batch[, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(**kwargs)	Additional explain_func assert, as the one in prepare() won't be executed when *a_batch != None*.
evaluate_batch(model, x_batch, y_batch, ...)	Evaluates model and attributes on a single data batch and returns the batched evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray | None*, *a_batch: ndarray | None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict[str, Any] | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = None*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*abs: bool = False*, *normalise: bool = False*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *input_shift: int | float = -1*, *perturb_func: Callable | None = None*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=False.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

input_shift: float, int
The value used to shift the input and the model bias as per the paper, default=-1.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be produced over all instances.

aggregate_func: callable
A Callable to aggregate the scores per instance to one float.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(***kwargs*) → None

Additional explain_func assert, as the one in prepare() won't be executed when *a_batch != None*.

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → ndarray

Evaluates model and attributes on a single data batch and returns the batched evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

Returns

scores_batch: np.ndarray
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Axiomatic'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Input Invariance'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.axiomatic.non_sensitivity module

This module contains the implementation of the Non-Sensitivity metric.

final class quantus.metrics.axiomatic.non_sensitivity.**NonSensitivity**(*eps: float = 1e-05, n_samples: int = 100, features_in_step: int = 1, abs: bool = True, normalise: bool = True, normalise_func: ~typing.Callable[[~numpy.ndarray], ~numpy.ndarray] | None = None, normalise_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, perturb_baseline: str = 'black', perturb_func: ~typing.Callable | None = None, perturb_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, return_aggregate: bool = False, aggregate_func: ~typing.Callable = <function mean>, default_plot_func: ~typing.Callable | None = None, disable_warnings: bool = False, display_progressbar: bool = False, **kwargs*)

Bases: *Metric*[List[float]]

Implementation of NonSensitivity by Nguyen at el., 2020.

Non-sensitivity measures if zero-importance is only assigned to features, that the model is not functionally dependent on.

References:

1) An-phi Nguyen and María Rodríguez Martínez.: "On quantitative aspects of model interpretability." arXiv preprint arXiv:2007.07584 (2020). 2) Marco Ancona et al.: "Explaining Deep Neural Networks with a Polynomial Time Algorithm for Shapley Values Approximation." ICML (2019): 272-281. 3) Grégoire Montavon et al.: "Methods for interpreting and understanding deep neural networks." Digital Signal Processing 73 (2018): 1-15.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*eps: float = 1e-05, n_samples: int = 100, features_in_step: int = 1, abs: bool = True, normalise: bool = True, normalise_func: ~typing.Callable[[~numpy.ndarray], ~numpy.ndarray] | None = None, normalise_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, perturb_baseline: str = 'black', perturb_func: ~typing.Callable | None = None, perturb_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, return_aggregate: bool = False, aggregate_func: ~typing.Callable = <function mean>, default_plot_func: ~typing.Callable | None = None, disable_warnings: bool = False, display_progressbar: bool = False, **kwargs*)

Parameters

eps: float
Attributions threshold, default=1e-5.

n_samples: integer
The number of samples to iterate over, default=100.

features_in_step: integer
The step size, default=1.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=True.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="black".

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be produced over all instances.

aggregate_func: callable
A Callable to aggregate the scores per instance to one float.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`custom_preprocess`(*x_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

kwargs:
Unused.

Returns

None

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[int]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInterface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

`evaluate_instance`(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → int

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

s: np.ndarray
The segmentation to be evaluated on an instance-basis.

Returns

integer:
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Axiomatic'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Non-Sensitivity'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'lower'

quantus.metrics.complexity package

Submodules

quantus.metrics.complexity.complexity module

This module contains the implementation of the Complexity metric.

final class quantus.metrics.complexity.complexity.**Complexity**(*abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Complexity metric by Bhatt et al., 2020.

Complexity of attributions is defined as the entropy of the fractional contribution of feature x_i to the total magnitude of the attribution. A complex explanation is one that uses all features in its explanation to explain

some decision. Even though such an explanation may be faithful to the model output, if the number of features is too large it may be too difficult for the user to understand the explanations, rendering it useless.

References:

1) Umang Bhatt et al.: "Evaluating and aggregating feature-based model explanations." IJCAI (2020): 3016-3022.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(x_batch, a_batch, **kwargs)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(x, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=True.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

default_plot_func: callable
Callable that plots the metrics result.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*x_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

static evaluate_instance(*x: ndarray*, *a: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

x: np.ndarray
The input to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Complexity'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Complexity'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'lower'

quantus.metrics.complexity.effective_complexity module

This module contains the implementation of the Effective Complexity metric.

final class quantus.metrics.complexity.effective_complexity.**EffectiveComplexity**(*eps: float = 1e-05*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Effective complexity metric by Nguyen at el., 2020.

Effective complexity measures how many attributions in absolute values are exceeding a certain threshold (eps) where a value above the specified threshold implies that the features are important and under indicates it is not.

References:

1) An-phi Nguyen and María Rodríguez Martínez.: "On quantitative aspects of model interpretability." arXiv preprint arXiv:2007.07584 (2020).

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.

- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(a_batch, **kwargs)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*eps: float = 1e-05*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

eps: float
Attributions threshold, default=1e-5.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=True.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

default_plot_func: callable
Callable that plots the metrics result.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*a_batch: ndarray*, ***kwargs*) → List[int]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

evaluate_instance(*a: ndarray*) → int

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

integer
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Complexity'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Effective Complexity'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'lower'

quantus.metrics.complexity.sparseness module

This module contains the implementation of the Sparseness metric.

final class quantus.metrics.complexity.sparseness.**Sparseness**(*abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Sparseness metric by Chalasani et al., 2020.

Sparseness is quantified using the Gini Index applied to the vector of the absolute values of attributions. The test asks that features that are truly predictive of the output F(x) should have significant contributions, and similarly, that irrelevant (or weakly-relevant) features should have negligible contributions.

Assumptions:

- Based on the implementation of the authors as found on the following link:

<https://github.com/jfc43/advex/blob/master/DNN-Experiments/Fashion-MNIST/utils.py>.

References:

1) Prasad Chalasani et al.: "Concise explanations of neural networks using adversarial training." International Conference on Machine Learning. PMLR, 2020.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(x_batch, a_batch, **kwargs)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(x, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=True.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

default_plot_func: callable
Callable that plots the metrics result.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction, default=False.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*x_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

`static evaluate_instance`(*x: ndarray*, *a: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

x: np.ndarray
The input to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

`evaluation_category: ClassVar[EvaluationCategory] = 'Complexity'`

`evaluation_scores: Any`

`explain_func: Callable`

`explain_func_kwargs: Dict[str, Any]`

`model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}`

`name: ClassVar[str] = 'Sparseness'`

`normalise_func: Callable[[np.ndarray], np.ndarray] | None`

`score_direction: ClassVar[ScoreDirection] = 'higher'`

quantus.metrics.faithfulness package

Submodules

quantus.metrics.faithfulness.faithfulness_correlation module

This module contains the implementation of the Faithfulness Correlation metric.

final class quantus.metrics.faithfulness.faithfulness_correlation.**FaithfulnessCorrelation**(*similarity_func: Callable | None = None*, *nr_runs: int = 100*, *subset_size: int = 224*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarra ndarray] | None = None*, *normalise_func_kwa Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwarg Dict[str, Any] | None = None*, *

Bases: *Metric*[List[float]]

Implementation of faithfulness correlation by Bhatt et al., 2020.

The Faithfulness Correlation metric intend to capture an explanation's relative faithfulness (or 'fidelity') with respect to the model behaviour.

Faithfulness correlation scores shows to what extent the predicted logits of each modified test point and the average explanation attribution for only the subset of features are (linearly) correlated, taking the average over multiple runs and test samples. The metric returns one float per input-attribution pair that ranges between -1 and 1, where higher scores are better.

For each test sample, |S| features are randomly selected and replace them with baseline values (zero baseline or average of set). Thereafter, Pearson's correlation coefficient between the predicted logits of each modified test point and the average explanation attribution for only the subset of features is calculated. Results is average over multiple runs and several test samples.

References:

1) Umang Bhatt et al.: "Evaluating and aggregating feature-based model explanations." IJCAI (2020): 3016-3022.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, *custom_batch: Any | None = None*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

custom_batch: any
Any object that can be passed to the evaluation process. Gives flexibility to the user to adapt for implementing their own metric.

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*similarity_func: Callable | None = None*, *nr_runs: int = 100*, *subset_size: int = 224*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = True*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

similarity_func: callable
Similarity function applied to compare input and perturbed input. If None, the default value is used, default=correlation_pearson.

nr_runs: integer
The number of runs (for each input and explanation pair), default=100.

subset_size: integer
The size of subset, default=224.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: “mean”, “random”, “uniform”, “black” or “white”, default=”black”.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(*x_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

kwargs:
Unused.

Returns

tuple
In addition to the x_batch, y_batch, a_batch, s_batch and custom_batch, returning a custom preprocess batch (custom_preprocess_batch).

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInterface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

evaluate_instance(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

```
evaluation_category: ClassVar[EvaluationCategory] = 'Faithfulness'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Faithfulness Correlation'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[ScoreDirection] = 'higher'
```

quantus.metrics.faithfulness.faithfulness_estimate module

This module contains the implementation of the Faithfulness Estimate metric.

final class quantus.metrics.faithfulness.faithfulness_estimate.**FaithfulnessEstimate**(*similarity_func: Callable | None = None*, *features_in_step: int = 1*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable*

Bases: *Metric*[List[float]]

Implementation of Faithfulness Estimate by Alvares-Melis at el., 2018a and 2018b.

Computes the correlations of probability drops and the relevance scores on various points, showing the aggregate statistics.

References:

1) David Alvarez-Melis and Tommi S. Jaakkola.: "Towards robust interpretability with self-explaining neural networks." NeurIPS (2018): 7786-7795.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, *custom_batch: Any | None = None*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

custom_batch: any
Any object that can be passed to the evaluation process. Gives flexibility to the user to adapt for implementing their own metric.

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*similarity_func: Callable | None = None*, *features_in_step: int = 1*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

similarity_func: callable

Similarity function applied to compare input and perturbed input.
If None, the default value is used, default=correlation_spearman.

features_in_step: integer
The size of the step, default=1.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: “mean”, “random”, “uniform”, “black” or “white”, default=”black”.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(*x_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

kwargs:
Unused.

Returns

tuple
In addition to the x_batch, y_batch, a_batch, s_batch and custom_batch, returning a custom preprocess batch (custom_preprocess_batch).

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInterface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

evaluate_instance(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Faithfulness'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Faithfulness Estimate'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.faithfulness.infidelity module

This module contains the implementation of the Infidelity metric.

final class quantus.metrics.faithfulness.infidelity.**Infidelity**(*loss_func: str | Callable = 'mse', perturb_patch_sizes: List[int] | None = None, n_perturb_samples: int = 10, abs: bool = False, normalise: bool = False, normalise_func: Callable[[ndarray], ndarray] | None = None, normalise_func_kwargs: Dict[str, Any] | None = None, perturb_func: Callable | None = None, perturb_baseline: str = 'black', perturb_func_kwargs: Dict[str, Any] | None = None, return_aggregate: bool = False, aggregate_func: Callable | None = None, default_plot_func: Callable | None = None, disable_warnings: bool = False, display_progressbar: bool = False, **kwargs*)

Bases: *Metric*[List[float]]

Implementation of Infidelity by Yeh et al., 2019.

Explanation infidelity represents the expected mean square error between 1) a dot product of an attribution and input perturbation and 2) difference in model output after significant perturbation.

Assumptions:

- The original implementation (https://github.com/chihkuanyeh/saliency_evaluation/

blob/master/infid_sen_utils.py) supports perturbation of Gaussian noise and squared patches. In this implementation, we use squared patches as the default option. - Since we use squared patches in this implementation, the metric is only applicable to 3-dimensional (image) data. To extend the applicability to other data domains, adjustments to the current implementation might be necessary.

References:

1) Chih-Kuan Yeh et al.: "On the (In)fidelity and Sensitivity of Explanations." 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*loss_func: str | Callable = 'mse'*, *perturb_patch_sizes: List[int] | None = None*, *n_perturb_samples: int = 10*, *abs: bool = False*, *normalise: bool = False*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

loss_func: string
Loss function, default="mse".

perturb_patch_sizes: list
List of patch sizes to be perturbed. If None, the defaul is used, default=[4].

features_in_step: integer
The size of the step, default=1.

n_perturb_samples: integer
The number of samples to be perturbed, default=10.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=False.

normalise_func: callable
Attribution normalisation function applied in case normalise=False. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="black".

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(*x_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

kwargs:
Unused.

Returns

None

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInterface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

evaluate_instance(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInterface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Faithfulness'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Infidelity'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'lower'

quantus.metrics.faithfulness.irof module

This module contains the implementation of the Iterative Removal of Features metric.

final class quantus.metrics.faithfulness.irof.**IROF**(*segmentation_method: str = 'slic'*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'mean'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = True*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of IROF (Iterative Removal of Features) by Rieger at el., 2020.

The metric computes the area over the curve per class for sorted mean importances of feature segments (super-pixels) as they are iteratively removed (and prediction scores are collected), averaged over several test samples.

Assumptions:

- The original metric definition relies on image-segmentation functionality. Therefore, only apply the metric to 3-dimensional (image) data. To extend the applicablity to other data domains, adjustments to the current implementation might be necessary.

References:

1) Laura Rieger and Lars Kai Hansen. "Irof: a low resource evaluation metric for explanation methods." arXiv preprint arXiv:2003.08747 (2020).

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.

- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_aoc_score
Calculate the area over the curve (AOC) score for several test samples.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: array*, *y_batch: array*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*segmentation_method: str = 'slic'*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'mean'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = True*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

segmentation_method: string
Image segmentation method:'slic' or 'felzenszwalb', default="slic".

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="mean".

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`custom_preprocess`(*x_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model e.g., torchvision.models that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

custom_batch: any
Gives flexibility ot the user to use for evaluation, can hold any variable.

Returns

None

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInterface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

`evaluate_instance`(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

——-

float
The evaluation results.

`evaluation_category: ClassVar[EvaluationCategory] = 'Faithfulness'`

`evaluation_scores: Any`

`explain_func: Callable`

`explain_func_kwargs: Dict[str, Any]`

`property get_aoc_score`

Calculate the area over the curve (AOC) score for several test samples.

`model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}`

`name: ClassVar[str] = 'IROF'`

`normalise_func: Callable[[np.ndarray], np.ndarray] | None`

`score_direction: ClassVar[ScoreDirection] = 'higher'`

quantus.metrics.faithfulness.monotonicity module

This module contains the implementation of the Monotonicity metric.

final class quantus.metrics.faithfulness.monotonicity.**Monotonicity**(*features_in_step: int = 1*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Monotonicity metric by Arya at el., 2019.

Monotonicity tests if adding more positive evidence increases the probability of classification in the specified class.

It captures attributions' faithfulness by incrementally adding each attribute in order of increasing importance and evaluating the effect on model performance. As more features are added, the performance of the model is expected to increase and thus result in monotonically increasing model performance.

References:

1) Vijay Arya et al.: "One explanation does not fit all: A toolkit and taxonomy of ai explainability techniques." arXiv preprint arXiv:1909.03012 (2019). 2) Ronny Luss et al.: "Generating contrastive explanations with monotonic attribute functions." arXiv preprint arXiv:1905.12698 (2019).

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

`__call__`(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
`batch_preprocess`(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
`custom_batch_preprocess`(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
`custom_postprocess`(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
`custom_preprocess`(x_batch, **kwargs)	Implementation of custom_preprocess_batch.
`evaluate_batch`(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
`evaluate_instance`(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
`explain_batch`(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
`general_preprocess`(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
`generate_batches`(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
`interpret_scores`()	Get an interpretation of the scores.
`plot`([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*features_in_step: int = 1*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

features_in_step: integer
The size of the step, default=1.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=True.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="black".

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(*x_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

kwargs:
Unused.

Returns

None

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInterface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

`evaluate_instance`(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

`evaluation_category: ClassVar[EvaluationCategory] = 'Faithfulness'`

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Monotonicity'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.faithfulness.monotonicity_correlation module

This module contains the implementation of the Monotonicity metric.

final class quantus.metrics.faithfulness.monotonicity_correlation.**MonotonicityCorrelation**(*similarity_func: Callable | None = None, eps: float = 1e-05, nr_samples: int = 100, features_in_step: int = 1, abs: bool = True, normalise: bool = True, normalise_func: Callable[[ndarra ndarray] | None = None, normalise_func_kwa Dict[str, Any] | None = None, perturb_func: Callable | None = None, perturb_baseline: str = 'uniform', per-*

Bases: *Metric*[List[float]]

Implementation of Monotonicity Correlation metric by Nguyen at el., 2020.

Monotonicity measures the (Spearman's) correlation coefficient of the absolute values of the attributions and the uncertainty in probability estimation. The paper argues that if attributions are not monotonic then they are not providing the correct importance of the feature.

References:

1) An-phi Nguyen and María Rodríguez Martínez.: "On quantitative aspects of model interpretability." arXiv preprint arXiv:2007.07584 (2020).

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, *custom_batch: Any | None = None*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

custom_batch: any
Any object that can be passed to the evaluation process. Gives flexibility to the user to adapt for implementing their own metric.

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*similarity_func: Callable | None = None*, *eps: float = 1e-05*, *nr_samples: int = 100*, *features_in_step: int = 1*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'uniform'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

similarity_func: callable
Similarity function applied to compare input and perturbed input. If None, the default value is used, default=correlation_spearman.

eps: float
Attributions threshold, default=1e-5.

nr_samples: integer
The number of samples to iterate over, default=100.

features_in_step: integer
The size of the step, default=1.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=True.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="uniform".

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`custom_preprocess`(*x_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

kwargs:
Unused.

Returns

None

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A model that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

`evaluate_instance`(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

```
evaluation_category: ClassVar[EvaluationCategory] = 'Faithfulness'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Monotonicity'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[ScoreDirection] = 'higher'
```

quantus.metrics.faithfulness.pixel_flipping module

This module contains the implementation of the Pixel-Flipping metric.

final class quantus.metrics.faithfulness.pixel_flipping.**PixelFlipping**(*features_in_step: int = 1*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *return_auc_per_sample: bool = False*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[Union[float, List[float]]]

Implementation of Pixel-Flipping experiment by Bach et al., 2015.

The basic idea is to compute a decomposition of a digit for a digit class and then flip pixels with highly positive, highly negative scores or pixels with scores close to zero and then to evaluate the impact of these flips onto the prediction scores (mean prediction is calculated).

References:

1) Sebastian Bach et al.: "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation." PloS one 10.7 (2015): e0130140.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_auc_score
Calculate the area under the curve (AUC) score for several test samples.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*features_in_step: int = 1*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *return_auc_per_sample: bool = False*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

features_in_step: integer
The size of the step, default=1.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="black".

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

return_auc_per_sample: boolean
Indicates if an AUC score should be computed over the curve and returned.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(*x_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

kwargs:
Unused.

Returns

None

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[float | List[float]]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

evaluate_instance(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → float | List[float]

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

list
The evaluation results.

`evaluation_category: ClassVar[EvaluationCategory] = 'Faithfulness'`

`evaluation_scores: Any`

`explain_func: Callable`

`explain_func_kwargs: Dict[str, Any]`

`property get_auc_score`

Calculate the area under the curve (AUC) score for several test samples.

`model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}`

`name: ClassVar[str] = 'Pixel-Flipping'`

`normalise_func: Callable[[np.ndarray], np.ndarray] | None`

`score_direction: ClassVar[ScoreDirection] = 'lower'`

quantus.metrics.faithfulness.region_perturbation module

This module contains the implementation of the Region Perturbation metric.

final class quantus.metrics.faithfulness.region_perturbation.**RegionPerturbation**(*patch_size: int = 8*, *order: str = 'morf'*, *regions_evaluation: int = 100*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Region Perturbation by Samek et al., 2015.

Consider a greedy iterative procedure that consists of measuring how the class encoded in the image (e.g. as measured by the function f) disappears when we progressively remove information from the image x, a process referred to as region perturbation, at the specified locations.

Assumptions:

- The original metric definition relies on image-patch functionality. Therefore, only apply the

metric to 3-dimensional (image) data. To extend the applicablity to other data domains, adjustments to the current implementation might be necessary.

References:
1) Wojciech Samek et al.: "Evaluating the visualization of what a deep neural network has learned." IEEE transactions on neural networks and learning systems 28.11 (2016): 2660-2673.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_auc_score`
Calculate the area under the curve (AUC) score for several test samples.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*patch_size: int = 8*, *order: str = 'morf'*, *regions_evaluation: int = 100*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

patch_size: integer
The patch size for masking, default=8.

regions_evaluation: integer
The number of regions to evaluate, default=100.

order: string

Indicates whether attributions are ordered randomly ("random"),
according to the most relevant first ("morf"), or least relevant first ("lerf"), default="morf".

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="black".

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[List[float]]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

evaluate_instance(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → List[float]

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

s: np.ndarray
The segmentation to be evaluated on an instance-basis.

Returns

: list
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Faithfulness'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

property get_auc_score

Calculate the area under the curve (AUC) score for several test samples.

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Region-Perturbation'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'lower'

quantus.metrics.faithfulness.road module

This module contains the implementation of the ROAD metric.

final class quantus.metrics.faithfulness.road.**ROAD**(*percentages: List[float] | None = None*, *noise: float = 0.01*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of ROAD evaluation strategy by Rong et al., 2022.

The ROAD approach measures the accuracy of the model on the provided test set at each step of an iterative process of removing k most important pixels. At each step k most relevant pixels (MoRF order) are replaced with noisy linear imputations which removes bias.

Assumptions:

- The original metric definition relies on perturbation functionality suited only for images.

Therefore, only apply the metric to 3-dimensional (image) data. To extend the applicablity to other data domains, adjustments to the current implementation might be necessary.

References:

1) Leemann Rong et al.: "Evaluating Feature Attribution: An Information-Theoretic Perspective." arXiv preprint arXiv:2202.00449 (2022).

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(a_batch, **kwargs)	ROAD requires *a_size* property to be set to *image_height * image_width* of an explanation.
custom_postprocess(**kwargs)	Post-process the evaluation results.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*percentages: List[float] | None = None*, *noise: float = 0.01*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

percentages (list): The list of percentages of the image to be removed, default=list(range(1, 100, 2)).
noise (noise): Noise added, default=0.01.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_batch_preprocess(*a_batch: ndarray*, ***kwargs*) → None

ROAD requires *a_size* property to be set to *image_height * image_width* of an explanation.

custom_postprocess(***kwargs*) → None

Post-process the evaluation results.

Parameters

kwargs:
Unused.

Returns

None

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[List[float]]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

evaluate_instance(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → List[float]

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

list:
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Faithfulness'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'ROAD'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'lower'

quantus.metrics.faithfulness.selectivity module

This module contains the implementation of the Selectivity metric.

final class quantus.metrics.faithfulness.selectivity.**Selectivity**(*patch_size: int = 8*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Selectivity test by Montavon et al., 2018.

At each iteration, a patch (e.g., of size 4 x 4) corresponding to the region with highest relevance is set to black. The plot keeps track of the function value as the features are being progressively removed and computes an average over a large number of examples.

Assumptions:

- The original metric definition relies on perturbation functionality suited only for images.

Therefore, only apply the metric to 3-dimensional (image) data. To extend the applicablity to other data domains, adjustments to the current implementation might be necessary.

References:

1) Grégoire Montavon et al.: "Methods for interpreting and understanding deep neural networks." Digital Signal Processing 73 (2018): 1-15.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_auc_score`
Calculate the area under the curve (AUC) score for several test samples.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*patch_size: int = 8*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

patch_size: integer
The patch size for masking, default=8.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="black".

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[List[float]]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

evaluate_instance(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → List[float]

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

: list
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Faithfulness'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

property get_auc_score

Calculate the area under the curve (AUC) score for several test samples.

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Selectivity'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'lower'

quantus.metrics.faithfulness.sensitivity_n module

This module contains the implementation of the Sensitivity-N metric.

final class quantus.metrics.faithfulness.sensitivity_n.**SensitivityN**(*similarity_func: Callable | None = None*, *n_max_percentage: float = 0.8*, *features_in_step: int = 1*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = True*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Sensitivity-N test by Ancona et al., 2019.

An attribution method satisfies Sensitivity-n when the sum of the attributions for any subset of features of cardinality n is equal to the variation of the output Sc caused removing the features in the subset. The test computes the correlation between sum of attributions and delta output.

Pearson correlation coefficient (PCC) is computed between the sum of the attributions and the variation in the target output varying n from one to about 80% of the total number of features, where an average across a thousand of samples is reported. Sampling is performed using a uniform probability distribution over the features.

References:

1) Marco Ancona et al.: "Towards better understanding of gradient-based attribution methods for deep neural networks." ICLR (Poster) (2018).

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.

- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(x_batch, **kwargs)	Post-process the evaluation results.
custom_preprocess(x_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It com-

pletes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*similarity_func: Callable | None = None*, *n_max_percentage: float = 0.8*, *features_in_step: int = 1*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_baseline: str = 'black'*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = True*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

similarity_func: callable
Similarity function applied to compare input and perturbed input, default=correlation_pearson.

n_max_percentage: float
The percentage of features to iteratively evaluatede, fault=0.8.

features_in_step: integer
The size of the step, default=1.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="black".

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`custom_postprocess`(*x_batch: ndarray*, ***kwargs*) → None

Post-process the evaluation results.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

kwargs:
Unused.

Returns

None

`custom_preprocess`(*x_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

kwargs:
Unused.

Returns

None

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[Dict[str, List[float]]]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch:
The evaluation results.

`evaluate_instance`(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → Dict[str, List[float]]

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

(Dict[str, List[float]]): The evaluation results.

`evaluation_category: ClassVar[EvaluationCategory] = 'Faithfulness'`

`evaluation_scores: Any`

`explain_func: Callable`

`explain_func_kwargs: Dict[str, Any]`

`model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}`

`name: ClassVar[str] = 'Sensitivity-N'`

`normalise_func: Callable[[np.ndarray], np.ndarray] | None`

`score_direction: ClassVar[ScoreDirection] = 'higher'`

quantus.metrics.faithfulness.sufficiency module

This module contains the implementation of the Sufficiency metric.

final class quantus.metrics.faithfulness.sufficiency.**Sufficiency**(*threshold: float = 0.6*, *distance_func: str = 'seuclidean'*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Sufficiency test by Dasgupta et al., 2022.

The (global) sufficiency metric measures the expected local sufficiency. Local sufficiency measures the probability of the prediction label for a given datapoint coinciding with the prediction labels of other data points that the same explanation applies to. For example, if the explanation of a given image is "contains zebra", the local sufficiency metric measures the probability a different that contains zebra having the same prediction label.

Assumptions:

- We assume that a given explanation applies to 'anothers' data point if the distance between

the explanation and the explanations of the data point is under the user-defined threshold.

References:

1) Sanjoy Dasgupta et al.: "Framework for Evaluating Faithfulness of Local Explanations." ICML (2022): 4794-4815.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(model, x_batch, ...)	Compute additional arguments required for Sufficiency evaluation on batch-level.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(i_batch, a_sim_vector_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(i, a_sim_vector, ...)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*threshold: float = 0.6*, *distance_func: str = 'seuclidean'*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

threshold: float
Distance threshold, default=0.6.

distance_func: string
Distance function, default = "seuclidean".

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=baseline_replacement_by_indices.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="black".

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_batch_preprocess(*model:* ModelInterface, *x_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → Dict[str, ndarray]

Compute additional arguments required for Sufficiency evaluation on batch-level.

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*i_batch: ndarray*, *a_sim_vector_batch: ndarray*, *y_pred_classes: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

i_batch:
The index of the current instance.

a_sim_vector_batch:
The custom input to be evaluated on an instance-basis.

y_pred_classes:
The class predictions of the complete input dataset.

kwargs:
Unused.

Returns

evaluation_scores:
List of measured sufficiency for each entry in the batch.

static evaluate_instance(*i: int*, *a_sim_vector: ndarray*, *y_pred_classes: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

i: int
The index of the current instance.

a_sim_vector: any
The custom input to be evaluated on an instance-basis.

y_pred_classes: np,ndarray
The class predictions of the complete input dataset.

Returns

float
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Faithfulness'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Sufficiency'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.localisation package

Submodules

quantus.metrics.localisation.attribution_localisation module

This module contains the implementation of the Attribution Localisation metric.

final class quantus.metrics.localisation.attribution_localisation.**AttributionLocalisation**(*weighted: bool = False*, *max_size: float = 1.0*, *positive_attributions: bool = False*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable | None = None*, *normalise_func_kwa Dict | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *display_progressbar bool*

Bases: *Metric*[List[float]]

Implementation of the Attribution Localization by Kohlbrenner et al., 2020.

Attribution Localization implements the ratio of positive attributions within the target to the overall attribution. High scores are desired, as it means, that the positively attributed pixels belong to the targeted object class.

References:

1) Max Kohlbrenner et al., "Towards Best Practice in Explaining Neural Network Decisions with LRP." IJCNN (2020): 1-7.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, s_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(x_batch, a_batch, s_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(x, a, s)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

`__init__`(*weighted: bool = False*, *max_size: float = 1.0*, *positive_attributions: bool = False*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable | None = None*, *normalise_func_kwargs: Dict | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *display_progressbar: bool = False*, *disable_warnings: bool = False*, ***kwargs*)

Parameters

weighted: boolean
Indicates whether the weighted variant of the inside-total relevance ratio is used, default=False.

max_size: float
The maximum ratio for the size of the bounding box to image, default=1.0.

positive_attributions: boolean
Indicates whether only positive attributions should be used, i.e., clipping, default=False.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=True.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`custom_preprocess`(*x_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

kwargs:
Unused.

Returns

———-

None

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*x_batch: ndarray*, *a_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

a_batch: np.ndarray
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray
A np.ndarray which contains segmentation masks that matches the input.

kwargs:
Unused.

Returns

scores_batch:
Evaluation result for batch.

evaluate_instance(*x: ndarray*, *a: ndarray*, *s: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

x: np.ndarray
The input to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

s: np.ndarray
The segmentation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Localisation'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Attribution Localisation'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.localisation.auc module

This module contains the implementation of the AUC metric.

final class quantus.metrics.localisation.auc.**AUC**(*abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of AUC metric by Fawcett et al., 2006.

AUC is a ranking metric and compares the ranking between attributions and a given ground-truth mask

References:

1) Tom Fawcett: 'An introduction to ROC analysis' "Pattern Recognition Letters" Vol 27, Issue 8, 2006

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, s_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(a_batch, s_batch, **kwargs)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(a, s)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.
s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.
channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list

a list of Any with the evaluation scores of the concerned batch.

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(*x_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

s_batch: np.ndarray
A Union[np.ndarray, None] which contains segmentation masks that matches the input.

kwargs:
Unused.

Returns

None

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*a_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

a_batch:
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch:
A np.ndarray which contains segmentation masks that matches the input.

kwargs:
Unused.

Returns

scores_batch:
Evaluation result for batch.

static evaluate_instance(*a: ndarray*, *s: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

a: np.ndarray
The explanation to be evaluated on an instance-basis.

s: np.ndarray
The segmentation to be evaluated on an instance-basis.

Returns

: float
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Localisation'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'AUC'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.localisation.focus module

This module contains the implementation of the AUC metric.

final class quantus.metrics.localisation.focus.**Focus**(*mosaic_shape: Any | None = None*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Focus evaluation strategy by Arias et. al. 2022

The Focus is computed through mosaics of instances from different classes, and the explanations these generate. Each mosaic contains four images: two images belonging to the target class (the specific class the feature attribution method is expected to explain) and the other two are chosen randomly from the rest of classes. Thus, the Focus estimates the reliability of feature attribution method's output as the probability of the sampled pixels lying on an image of the target class of the mosaic. This is equivalent to the proportion of positive relevance lying on those images.

References:

1) Anna Arias-Duart et al.: "Focus! Rating XAI Methods and Finding Biases" FUZZ-IEEE (2022): 1-8.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(model, x_batch, y_batch, ...)	Implementation of custom_preprocess_batch.
evaluate_batch(a_batch, c_batch, **kwargs)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(a, c)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

quadrant_bottom_left
quadrant_bottom_right
quadrant_top_left
quadrant_top_right

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, *custom_batch: Any | None = None*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

For this metric to run we need to get the positions of the target class within the mosaic. This should be a

np.ndarray containing one tuple per sample, representing the positions of the target class within the mosaic (where each tuple contains 0/1 values referring to (top_left, top_right, bottom_left, bottom_right).

An example:
>> custom_batch=[(1, 1, 0, 0), (0, 0, 1, 1), (1, 0, 1, 0), (0, 1, 0, 1)]

How to initialise the metric and evaluate explanations by calling the metric instance?
>> metric = Focus() >> scores = {method: metric(**init_params)(model=model,

x_batch=x_mosaic_batch, y_batch=y_mosaic_batch, a_batch=None, custom_batch=p_mosaic_batch, **{"explain_func": explain,

"explain_func_kwargs": { "method": "LayerGradCAM", "gc_layer": "model._modules.get('conv_2')", "pos_only": True, "interpolate": (2*28, 2*28), "interpolate_mode": "bilinear",} "device": device}) for method in ["LayerGradCAM", "IntegratedGradients"]}

Plot example! >> metric.plot(results=scores)

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

custom_batch: any
Any object that can be passed to the evaluation process. Gives flexibility to the user to adapt for implementing their own metric.

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*mosaic_shape: Any | None = None*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`custom_preprocess`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *custom_batch: ndarray*, ***kwargs*) → Dict[str, Any]

Implementation of custom_preprocess_batch.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model e.g., torchvision.models that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

custom_batch: any
Gives flexibility ot the user to use for evaluation, can hold any variable.

kwargs:
Unused.

Returns

dictionary[str, np.ndarray]
Output dictionary with two items: 1) 'c_batch' as key and custom_batch as value. 2) 'custom_batch' as key and None as value. This results in the keyword argument 'c' being passed to *evaluate_instance()*.

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE}`

`evaluate_batch`(*a_batch: ndarray*, *c_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

a_batch:
A np.ndarray which contains pre-computed attributions i.e., explanations.

c_batch:
The custom input to be evaluated on an batch-basis.

kwargs:
Unused.

Returns

score_batch:
Evaluation result for batch.

evaluate_instance(*a: ndarray*, *c: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

a: np.ndarray
The explanation to be evaluated on an instance-basis.

c: any
The custom input to be evaluated on an instance-basis.

Returns

float
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Localisation'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Focus'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

quadrant_bottom_left(*a: ndarray*) → ndarray

quadrant_bottom_right(*a: ndarray*) → ndarray

quadrant_top_left(*a: ndarray*) → ndarray

quadrant_top_right(*a: ndarray*) → ndarray

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.localisation.pointing_game module

This module contains the implementation of the Pointing-Game metric.

final class quantus.metrics.localisation.pointing_game.**PointingGame**(*weighted: bool = False*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of the Pointing Game by Zhang et al., 2018.

The Pointing Game implements a check whether the point of maximal attribution is on target, denoted by a binary mask. High scores are desired as it means, that the maximal attributed pixel belongs to an object of the specified class.

References:

1) Jianming Zhang et al.: "Top-Down Neural Attention by Excitation Backprop." International Journal of Computer Vision (2018) 126:1084-1102.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, s_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(*args, a_batch, s_batch, **kwargs)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(a, s)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*weighted: bool = False*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

weighted: boolean
Indicates whether output score is weighted by size of segmentation map, default=False.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(*x_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

Returns

None

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(**args*, *a_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

a_batch:
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch:
A np.ndarray which contains segmentation masks that matches the input.

args:
Unused.

kwargs:
Unused.

Returns

scores_batch:
Evaluation result for batch.

evaluate_instance(*a: ndarray*, *s: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

a: np.ndarray
The explanation to be evaluated on an instance-basis.

s: np.ndarray
The segmentation to be evaluated on an instance-basis.

Returns

boolean
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Localisation'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Pointing-Game'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.localisation.relevance_mass_accuracy module

This module contains the implementation of the Relevance Mass Accuracy metric.

final class quantus.metrics.localisation.relevance_mass_accuracy.**RelevanceMassAccuracy**(*abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of the Relevance Mass Accuracy by Arras et al., 2021.

The Relevance Mass Accuracy computes the ratio of attributions inside the bounding box to the sum of overall positive attributions. High scores are desired, as the pixels with the highest positively attributed scores should be within the bounding box of the targeted object.

References:

1) Leila Arras et al.: "CLEVR-XAI: A benchmark dataset for the ground truth evaluation of neural network explanations." Inf. Fusion 81 (2022): 14-40.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, s_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(a_batch, s_batch, **kwargs)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(a, s)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(*x_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

kwargs:
Unused.

Returns
———-
None

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*a_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

a_batch:
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch:
A np.ndarray which contains segmentation masks that matches the input.

kwargs:
Unused.

Returns

scores_batch:
A list of Any with the evaluation scores for the batch.

static evaluate_instance(*a: ndarray*, *s: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

a: np.ndarray
The explanation to be evaluated on an instance-basis.

s: np.ndarray
The segmentation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Localisation'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Relevance Mass Accuracy'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.localisation.relevance_rank_accuracy module

This module contains the implementation of the Relevance Rank Accuracy metric.

final class quantus.metrics.localisation.relevance_rank_accuracy.**RelevanceRankAccuracy**(*abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of the Relevance Rank Accuracy by Arras et al., 2021.

The Relevance Rank Accuracy measures the ratio of high intensity relevances within the ground truth mask GT. With P_top-k being the set of pixels sorted by there relevance in decreasing order until the k-th pixels, the rank accuracy is computed as: rank accuracy = (**|P_top-k intersect GT|**) / **|GT|**. High scores are desired, as the pixels with the highest positively attributed scores should be within the bounding box of the targeted object.

References:

1) Leila Arras et al.: "CLEVR-XAI: A benchmark dataset for the ground truth evaluation of neural network explanations." Inf. Fusion 81 (2022): 14-40.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, s_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(a_batch, s_batch, **kwargs)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(a, s)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(*x_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

kwargs:
Unused.

Returns
———-
None

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*a_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

a_batch:
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch:
A np.ndarray which contains segmentation masks that matches the input.

kwargs:
Unused

Returns

scores_batch:
Evaluation result for batch.

static evaluate_instance(*a: ndarray*, *s: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

a: np.ndarray
The explanation to be evaluated on an instance-basis.

s: np.ndarray
The segmentation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Localisation'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Relevance Rank Accuracy'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.localisation.top_k_intersection module

This module contains the implementation of the Top-K Intersection metric.

final class quantus.metrics.localisation.top_k_intersection.**TopKIntersection**(*k: int = 1000*, *concept_influence: bool = False*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of the top-k intersection by Theiner et al., 2021.

The TopKIntersection implements the pixel-wise intersection between a ground truth target object mask and an "explainer" mask, the binarized version of the explanation. High scores are desired, as the overlap between the ground truth object mask and the attribution mask should be maximal.

References:

1) Jonas Theiner et al.: "Interpretable Semantic Photo Geolocalization." arXiv preprint arXiv:2104.14995 (2021).

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, s_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(a_batch, s_batch, **kwargs)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(a, s)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*k: int = 1000*, *concept_influence: bool = False*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

k: integer
Top k attributions values to use, default=1000.

concept_influence: boolean
Indicates whether concept influence metric is used, default=False.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(*x_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

kwargs:
Unused.

Returns

None

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*a_batch: ndarray*, *s_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

a_batch:
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch:
A np.ndarray which contains segmentation masks that matches the input.

kwargs:
Unused.

Returns

scores_batch:
Evaluation result for batch.

evaluate_instance(*a: ndarray*, *s: ndarray*)

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

a: np.ndarray
The explanation to be evaluated on an instance-basis.

s: np.ndarray
The segmentation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Localisation'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Top-K Intersection'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'higher'

quantus.metrics.randomisation package

Submodules

quantus.metrics.randomisation.model_parameter_randomisation module

quantus.metrics.randomisation.random_logit module

This module contains the implementation of the Random Logit metric.

final class quantus.metrics.randomisation.random_logit.**RandomLogit**(*similarity_func: Callable | None = None*, *num_classes: int = 1000*, *seed: int = 42*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of the Random Logit Metric by Sixt et al., 2020.

The Random Logit Metric computes the distance between the original explanation and a reference explanation of a randomly chosen non-target class.

References:

1) Leon Sixt et al.: "When Explanations Lie: Why Many Modified BP Attributions Fail." ICML (2020): 9046-9057.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.

- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(**kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y, a)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*similarity_func: Callable | None = None*, *num_classes: int = 1000*, *seed: int = 42*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

similarity_func: callable
Similarity function applied to compare input and perturbed input, default=ssim.

num_classes: integer
Number of prediction classes in the input, default=1000.

seed: integer
Seed used for the random generator, default=42.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_preprocess(***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

kwargs:
Unused.

Returns

None

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model:
A model that is subject to explanation.

x_batch:
A np.ndarray which contains the input data that are explained.

y_batch:
A np.ndarray which contains the output labels that are explained.

a_batch:
A np.ndarray which contains pre-computed attributions i.e., explanations.

kwargs:
Unused.

Returns

scores_batch:
Evaluation results.

evaluate_instance(*model:* ModelInterface, *x: ndarray*, *y: ndarray*, *a: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

a: np.ndarray
The explanation to be evaluated on an instance-basis.

Returns

float
The evaluation results.

```
evaluation_category: ClassVar[EvaluationCategory] = 'Randomisation'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Random Logit'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[ScoreDirection] = 'lower'
```

quantus.metrics.robustness package

Submodules

quantus.metrics.robustness.avg_sensitivity module

This module contains the implementation of the Avg-Sensitivity metric.

final class quantus.metrics.robustness.avg_sensitivity.**AvgSensitivity**(*similarity_func: ~typing.Callable | None = None, norm_numerator: ~typing.Callable | None = None, norm_denominator: ~typing.Callable | None = None, nr_samples: int = 200, abs: bool = False, normalise: bool = False, normalise_func: ~typing.Callable[[~numpy.ndarray], ~numpy.ndarray] | None = None, normalise_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, perturb_func: ~typing.Callable = <function uniform_noise>, lower_bound: float = 0.2, upper_bound: None | float = None, perturb_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, return_aggregate: bool = False, aggregate_func: ~typing.Callable | None = None, default_plot_func: ~typing.Callable | None = None, disable_warnings: bool = False, display_progressbar: bool = False, return_nan_when_prediction_changes: bool = False, **kwargs*)

Bases: *Metric*[List[float]]

Implementation of Avg-Sensitivity by Yeh at el., 2019.

Using Monte Carlo sampling-based approximation while measuring how explanations change under slight perturbation - the average sensitivity is captured.

References:

1) Chih-Kuan Yeh et al. "On the (in) fidelity and sensitivity for explanations." NeurIPS (2019): 10965-10976. 2) Umang Bhatt et al.: "Evaluating and aggregating feature-based model explanations." IJCAI (2020): 3016-3022.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(**kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	Evaluates model and attributes on a single data batch and returns the batched evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*similarity_func: ~typing.Callable | None = None, norm_numerator: ~typing.Callable | None = None, norm_denominator: ~typing.Callable | None = None, nr_samples: int = 200, abs: bool = False, normalise: bool = False, normalise_func: ~typing.Callable[[~numpy.ndarray], ~numpy.ndarray] | None = None, normalise_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, perturb_func: ~typing.Callable = <function uniform_noise>, lower_bound: float = 0.2, upper_bound: None | float = None, perturb_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, return_aggregate: bool = False, aggregate_func: ~typing.Callable | None = None, default_plot_func: ~typing.Callable | None = None, disable_warnings: bool = False, display_progressbar: bool = False, return_nan_when_prediction_changes: bool = False, **kwargs*)

Parameters

similarity_func: callable
Similarity function applied to compare input and perturbed input. If None, the default value is used, default=difference.

norm_numerator: callable
Function for norm calculations on the numerator. If None, the default value is used, default=fro_norm

norm_denominator: callable
Function for norm calculations on the denominator. If None, the default value is used, default=fro_norm

nr_samples: integer
The number of samples iterated, default=200.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=gaussian_noise.

perturb_std: float
The amount of noise added, default=0.1.

perturb_mean: float
The mean of noise added, default=0.0.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

return_nan_when_prediction_changes: boolean
When set to true, the metric will be evaluated to NaN if the prediction changes after the perturbation is applied.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`custom_preprocess`(***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

kwargs:
Unused.

Returns

None

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → ndarray

Evaluates model and attributes on a single data batch and returns the batched evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on an instance-basis.

y_batch: np.ndarray
The output to be evaluated on an instance-basis.

a_batch: np.ndarray
The explanation to be evaluated on an instance-basis.

kwargs:
Unused.

Returns

scores_batch: np.ndarray
The batched evaluation results.

`evaluation_category: ClassVar[EvaluationCategory] = 'Robustness'`

`evaluation_scores: Any`

`explain_func: Callable`

`explain_func_kwargs: Dict[str, Any]`

`model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}`

`name: ClassVar[str] = 'Avg-Sensitivity'`

`normalise_func: Callable[[np.ndarray], np.ndarray] | None`

`score_direction: ClassVar[ScoreDirection] = 'lower'`

quantus.metrics.robustness.consistency module

This module contains the implementation of the Consistency metric.

final class quantus.metrics.robustness.consistency.**Consistency**(*discretise_func: Callable | None = None*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of the Consistency metric which measures the expected local consistency, i.e., the probability of the prediction label for a given datapoint coinciding with the prediction labels of other data points that the same explanation is being attributed to. For example, if the explanation of a given image is "contains zebra", the local consistency metric measures the probability a different image that the explanation "contains zebra" is being attributed to having the same prediction label.

Assumptions:

- A used-defined discreization function is used to discretize continuous explanation spaces.

References:

1) Sanjoy Dasgupta et al.: "Framework for Evaluating Faithfulness of Local Explanations." ICML (2022): 4794-4815.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(model, x_batch, ...)	Compute additional arguments required for Consistency on batch-level.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(a_batch, i_batch, ...)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(a, i, a_label, y_pred_classes)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*discretise_func: Callable | None = None*, *abs: bool = True*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, ***kwargs*)

Parameters

discretise_func: callable
Discretisation function applied to explantions. If None, the default value is used, default=top_n_sign.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

kwargs: optional
Keyword arguments.

a_axes: Sequence[int]

all_evaluation_scores: Any

custom_batch_preprocess(*model:* ModelInterface, *x_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → Dict[str, ndarray]

Compute additional arguments required for Consistency on batch-level.

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*a_batch: ndarray*, *i_batch: ndarray*, *a_label_batch: ndarray*, *y_pred_classes: ndarray*, ***kwargs*) → List[float]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

a_batch:
Batch of explanation to be evaluated.

i_batch:
Batch of segmentations to be evaluated.

a_label_batch:
Batch of discretised attribution labels.

y_pred_classes:
The class predictions of the complete input dataset.

kwargs:
Unused.

Returns

scores_batch:
Evaluation results.

`static evaluate_instance`(*a: ndarray*, *i: int*, *a_label: ndarray*, *y_pred_classes: ndarray*) → float

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

a: np.ndarray
The explanation to be evaluated on an instance-basis.

i: int
The index of the current instance.

a_label: np.ndarray
The discretised attribution labels.

y_pred_classes: np,ndarray
The class predictions of the complete input dataset.

Returns

float
The evaluation results.

`evaluation_category: ClassVar[EvaluationCategory] = 'Robustness'`

`evaluation_scores: Any`

`explain_func: Callable`

`explain_func_kwargs: Dict[str, Any]`

`model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}`

`name: ClassVar[str] = 'Consistency'`

`normalise_func: Callable[[np.ndarray], np.ndarray] | None`

`score_direction: ClassVar[ScoreDirection] = 'lower'`

quantus.metrics.robustness.continuity module

This module contains the implementation of the Continuity metric.

final class quantus.metrics.robustness.continuity.**Continuity**(*similarity_func: ~typing.Callable | None = None, nr_steps: int = 28, patch_size: int = 7, abs: bool = True, normalise: bool = True, normalise_func: ~typing.Callable[[~numpy.ndarray], ~numpy.ndarray] | None = None, normalise_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, perturb_func: ~typing.Callable | None = None, perturb_baseline: str = 'black', perturb_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, return_aggregate: bool = False, aggregate_func: ~typing.Callable | None = <function mean>, default_plot_func: ~typing.Callable | None = None, disable_warnings: bool = False, display_progressbar: bool = False, return_nan_when_prediction_changes: bool = False, **kwargs*)

Bases: *Metric*[List[float]]

Implementation of the Continuity test by Montavon et al., 2018.

The test measures the strongest variation of the explanation in the input domain i.e., ||R(x) - R(x')||_1 / ||x - x'||_2 where R(x) is the explanation for input x and x' is the perturbed input.

Assumptions:

- The original metric definition relies on perturbation functionality suited only for images.

Therefore, only apply the metric to 3-dimensional (image) data. To extend the applicablity to other data domains, adjustments to the current implementation might be necessary.

References:

1) Grégoire Montavon et al.: "Methods for interpreting and understanding deep neural networks." Digital Signal Processing 73 (2018): 1-15.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

aggregated_score
Implements a continuity correlation score (an addition to the original method) to evaluate the relationship between change in explanation and change in function output.

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(x_batch, **kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, **kwargs)	This method performs XAI evaluation on a single batch of explanations.
evaluate_instance(model, x, y)	Evaluate instance gets model and data for a single instance as input and returns the evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It com-

pletes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*similarity_func: ~typing.Callable | None = None, nr_steps: int = 28, patch_size: int = 7, abs: bool = True, normalise: bool = True, normalise_func: ~typing.Callable[[~numpy.ndarray], ~numpy.ndarray] | None = None, normalise_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, perturb_func: ~typing.Callable | None = None, perturb_baseline: str = 'black', perturb_func_kwargs: ~typing.Dict[str, ~typing.Any] | None = None, return_aggregate: bool = False, aggregate_func: ~typing.Callable | None = <function mean>, default_plot_func: ~typing.Callable | None = None, disable_warnings: bool = False, display_progressbar: bool = False, return_nan_when_prediction_changes: bool = False, **kwargs*)

Parameters

similarity_func: callable
Similarity function applied to compare input and perturbed input. If None, the default value is used, default=difference.

patch_size: integer
The patch size for masking, default=7.

nr_steps: integer
The number of steps to iterate over, default=28.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=True.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=translation_x_direction.

perturb_baseline: string
Indicates the type of baseline: "mean", "random", "uniform", "black" or "white", default="black".

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

default_plot_func: callable
Callable that plots the metrics result.

return_nan_when_prediction_changes: boolean
When set to true, the metric will be evaluated to NaN if the prediction changes after the perturbation is applied.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`property aggregated_score`

Implements a continuity correlation score (an addition to the original method) to evaluate the relationship between change in explanation and change in function output. It can be seen as an quantitative interpretation of visually determining how similar f(x) and R(x1) curves are.

`all_evaluation_scores: Any`

`custom_preprocess`(*x_batch: ndarray*, ***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

kwargs:
Unused.

Returns
——-
None.

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, ***kwargs*) → List[Dict[str, int]]

This method performs XAI evaluation on a single batch of explanations. For more information on the specific logic, we refer the metric's initialisation docstring.

Parameters

model:
A model that is subject to explanation.

x_batch:
A np.ndarray which contains the input data that are explained.

y_batch:
A np.ndarray which contains the output labels that are explained.

kwargs:
Unused.

Returns

scores_batch:
Evaluation results.

`evaluate_instance`(*model:* ModelInterface, *x: ndarray*, *y: ndarray*) → Dict[int, List[Any]]

Evaluate instance gets model and data for a single instance as input and returns the evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x: np.ndarray
The input to be evaluated on an instance-basis.

y: np.ndarray
The output to be evaluated on an instance-basis.

Returns

dict
The evaluation results.

`evaluation_category: ClassVar[EvaluationCategory] = 'Robustness'`

`evaluation_scores: Any`

`explain_func: Callable`

`explain_func_kwargs: Dict[str, Any]`

`model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}`

`name: ClassVar[str] = 'Continuity'`

`normalise_func: Callable[[np.ndarray], np.ndarray] | None`

`score_direction: ClassVar[ScoreDirection] = 'lower'`

quantus.metrics.robustness.local_lipschitz_estimate module

This module contains the implementation of the Local Lipschitz Estimate metric.

final class quantus.metrics.robustness.local_lipschitz_estimate.**LocalLipschitzEstimate**(*similarity_func: Callable | None = None*, *norm_numerator: Callable | None = None*, *norm_denominator: Callable | None = None*, *nr_samples: int = 200*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_mean: float = 0.0*,

Bases: *Metric*[List[float]]

Implementation of the Local Lipschitz Estimate (or Stability) test by Alvarez-Melis et al., 2018a, 2018b.

This tests asks how consistent are the explanations for similar/neighboring examples. The test denotes a (weaker) empirical notion of stability based on discrete, finite-sample neighborhoods i.e., argmax_(||f(x) - f(x')||_2 / ||x - x'||_2) where f(x) is the explanation for input x and x' is the perturbed input.

References:

1) David Alvarez-Melis and Tommi S. Jaakkola. "On the robustness of interpretability methods." arXiv preprint arXiv:1806.08049 (2018).

2) David Alvarez-Melis and Tommi S. Jaakkola. "Towards robust interpretability with self-explaining neural networks." NeurIPS (2018): 7786-7795.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(**kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	Evaluates model and attributes on a single data batch and returns the batched evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = True*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*similarity_func: Callable | None = None*, *norm_numerator: Callable | None = None*, *norm_denominator: Callable | None = None*, *nr_samples: int = 200*, *abs: bool = False*, *normalise: bool = True*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *perturb_mean: float = 0.0*, *perturb_std: float = 0.1*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, *return_nan_when_prediction_changes: bool = False*, ***kwargs*)

Parameters

similarity_func: callable
Similarity function applied to compare input and perturbed input. If None, the default value is used, default=lipschitz_constant.

norm_numerator: callable
Function for norm calculations on the numerator. If None, the default value is used, default=distance_euclidean.

norm_denominator: callable
Function for norm calculations on the denominator. If None, the default value is used, default=distance_euclidean.

nr_samples: integer
The number of samples iterated, default=200.

abs: boolean
Indicates whether absolute operation is applied on the attribution, default=False.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=gaussian_noise.

perturb_std: float
The amount of noise added, default=0.1.

perturb_mean: float
The mean of noise added, default=0.0.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

return_nan_when_prediction_changes: boolean
When set to true, the metric will be evaluated to NaN if the prediction changes after the perturbation is applied.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`custom_preprocess`(***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

kwargs:
Unused.

Returns

None

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → ndarray

Evaluates model and attributes on a single data batch and returns the batched evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

kwargs:
Unused.

Returns

scores_batch: np.ndarray
The batched evaluation results.

`evaluation_category: ClassVar[EvaluationCategory] = 'Robustness'`

`evaluation_scores: Any`

`explain_func: Callable`

`explain_func_kwargs: Dict[str, Any]`

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Local Lipschitz Estimate'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'lower'

quantus.metrics.robustness.max_sensitivity module

This module contains the implementation of the Max-Sensitivity metric.

final class quantus.metrics.robustness.max_sensitivity.**MaxSensitivity**(*similarity_func: Callable | None = None*, *norm_numerator: Callable | None = None*, *norm_denominator: Callable | None = None*, *nr_samples: int = 200*, *abs: bool = False*, *normalise: bool = False*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *lower_bound: float = 0.2*, *upper_bound: None | float = None*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, *return_nan_when_prediction_changes: bool = False*, ***kwargs*)

Bases: *Metric*[List[float]]

Implementation of Max-Sensitivity by Yeh at el., 2019.

Using Monte Carlo sampling-based approximation while measuring how explanations change under slight perturbation - the maximum sensitivity is captured.

References:

1) Chih-Kuan Yeh et al. "On the (in) fidelity and sensitivity for explanations." NeurIPS (2019): 10965-10976. 2) Umang Bhatt et al.: "Evaluating and aggregating feature-based model explanations." IJCAI (2020): 3016-3022.

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, a_batch, ...])	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(**kwargs)	Implementation of custom_preprocess_batch.
evaluate_batch(model, x_batch, y_batch, ...)	Evaluates model and attributes on a single data batch and returns the batched evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model*, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None = None*, *s_batch: ndarray | None = None*, *channel_first: bool | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict | None = None*, *model_predict_kwargs: Dict | None = None*, *softmax: bool | None = False*, *device: str | None = None*, *batch_size: int = 64*, ***kwargs*) → List[float]

This implementation represents the main logic of the metric and makes the class object callable. It completes instance-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls () on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*similarity_func: Callable | None = None*, *norm_numerator: Callable | None = None*, *norm_denominator: Callable | None = None*, *nr_samples: int = 200*, *abs: bool = False*, *normalise: bool = False*, *normalise_func: Callable[[ndarray], ndarray] | None = None*, *normalise_func_kwargs: Dict[str, Any] | None = None*, *perturb_func: Callable | None = None*, *lower_bound: float = 0.2*, *upper_bound: None | float = None*, *perturb_func_kwargs: Dict[str, Any] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable | None = None*, *default_plot_func: Callable | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, *return_nan_when_prediction_changes: bool = False*, ***kwargs*)

Parameters

similarity_func: callable
Similarity function applied to compare input and perturbed input. If None, the default value is used, default=difference.

norm_numerator: callable
Function for norm calculations on the numerator. If None, the default value is used, default=fro_norm

norm_denominator: callable
Function for norm calculations on the denominator. If None, the default value is used, default=fro_norm

nr_samples: integer
The number of samples iterated, default=200.

normalise: boolean
Indicates whether normalise operation is applied on the attribution, default=True.

normalise_func: callable
Attribution normalisation function applied in case normalise=True. If normalise_func=None, the default value is used, default=normalise_by_max.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=gaussian_noise.

lower_bound: float
The lower bound of the noise.

upper_bound: float, optional
The upper bound of the noise.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

return_nan_when_prediction_changes: boolean
When set to true, the metric will be evaluated to NaN if the prediction changes after the perturbation is applied.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`custom_preprocess`(***kwargs*) → None

Implementation of custom_preprocess_batch.

Parameters

kwargs:
Unused.

Returns

None

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → ndarray

Evaluates model and attributes on a single data batch and returns the batched evaluation result.

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on an instance-basis.

y_batch: np.ndarray
The output to be evaluated on an instance-basis.

a_batch: np.ndarray
The explanation to be evaluated on an instance-basis.

kwargs:
Unused.

Returns

scores_batch: np.ndarray
The batched evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Robustness'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Max-Sensitivity'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*] = 'lower'

quantus.metrics.robustness.relative_input_stability module

final class quantus.metrics.robustness.relative_input_stability.**RelativeInputStability**(*nr_samples: int = 200, abs: bool = False, normalise: bool = False, normalise_func: Callable[[np.ndarray, np.ndarray] | None = None, normalise_func_kwargs: Dict[str, ...] | None = None, perturb_func: Callable | None = None, perturb_func_kwargs: Dict[str, ...] | None = None, return_aggregate: bool = False, aggregate_func: Callable[[np.ndarray], np.float] | None = None, disable_warnings:*

Bases: *Metric*[List[float]]

Relative Input Stability leverages the stability of an explanation with respect to the change in the input data.

RIS(x, x', e_x, e_x') = max frac{||frac{e_x - e_{x'}}{e_x}||_p} {max (||frac{x - x'}{x}||_p, epsilon_{min})}

References:

1) Chirag Agarwal, et. al., 2022. "Rethinking stability for attribution based explanations.", https://arxiv.org/abs/2203.06877

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, ...])	For each image *x*:
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(model, x_batch, y_batch, ...)	**Parameters**
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.
relative_input_stability_objective(x, xs, ...)	Computes relative input stabilities maximization objective as defined here https://arxiv.org/pdf/2203.06877.pdf by the authors.

__call__(*model: tf.keras.Model | torch.nn.Module*, *x_batch: np.ndarray*, *y_batch: np.ndarray*, *model_predict_kwargs: Dict[str, ...] | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict[str, ...] | None = None*, *a_batch: np.ndarray | None = None*, *device: str | None = None*, *softmax: bool = False*, *channel_first: bool = True*, *batch_size: int = 64*, ***kwargs*) → List[float]

For each image *x*:

- Generate *num_perturbations* perturbed *xs* in the neighborhood of *x*.
- Compute explanations *e_x* and *e_xs*.
- Compute relative input stability objective, find max value with respect to *xs*.
- In practise we just use *max* over a finite *xs_batch*.

Parameters

model: tf.keras.Model, torch.nn.Module
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
4D tensor representing batch of input images

y_batch: np.ndarray
1D tensor, representing predicted labels for the x_batch.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

explain_func: callable, optional
Function used to generate explanations.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

a_batch: np.ndarray, optional
4D tensor with pre-computed explanations for the x_batch.

device: str, optional
Device on which torch should perform computations.

softmax: boolean, optional
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

batch_size: int
The batch size to be used.

kwargs:
not used, deprecated

Returns
———-

relative input stability: float, np.ndarray
float in case *return_aggregate=True*, otherwise np.ndarray of floats

__init__(*nr_samples: int = 200*, *abs: bool = False*, *normalise: bool = False*, *normalise_func: Callable[[np.ndarray], np.ndarray] | None = None*, *normalise_func_kwargs: Dict[str, ...] | None = None*, *perturb_func: Callable | None = None*, *perturb_func_kwargs: Dict[str, ...] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable[[np.ndarray], np.float] | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, *eps_min: float = 1e-06*, *default_plot_func: Callable | None = None*, *return_nan_when_prediction_changes: bool = True*, ***kwargs*)

Parameters

nr_samples: int
The number of samples iterated, default=200.

abs: boolean
Indicates whether absolute operation is applied on the attribution.

normalise: boolean
Flag stating if the attributions should be normalised

normalise_func: callable
Attribution normalisation function applied in case normalise=True.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=gaussian_noise.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

default_plot_func: callable
Callable that plots the metrics result.

eps_min: float
Small constant to prevent division by 0 in relative_stability_objective, default 1e-6.

return_nan_when_prediction_changes: boolean
When set to true, the metric will be evaluated to NaN if the prediction changes after the perturbation is applied, default=True.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → ndarray

Parameters

model: tf.keras.Model, torch.nn.Module
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
4D tensor representing batch of input images.

y_batch: np.ndarray
1D tensor, representing predicted labels for the x_batch.

a_batch: np.ndarray, optional
4D tensor with pre-computed explanations for the x_batch.

kwargs:
Unused.

Returns

ris: np.ndarray
The batched evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*] = 'Robustness'

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]] = {ModelType.TF, ModelType.TORCH}

name: ClassVar[str] = 'Relative Output Stability'

normalise_func: Callable[[np.ndarray], np.ndarray] | None

relative_input_stability_objective(*x: ndarray*, *xs: ndarray*, *e_x: ndarray*, *e_xs: ndarray*) → ndarray

Computes relative input stabilities maximization objective as defined here https://arxiv.org/pdf/2203.06877.pdf by the authors.

Parameters

x: np.ndarray
Batch of images.

xs: np.ndarray
Batch of perturbed images.

e_x: np.ndarray
Explanations for x.

e_xs: np.ndarray
Explanations for xs.

Returns

ris_obj: np.ndarray
RIS maximization objective.

score_direction: ClassVar[*ScoreDirection*] = 'lower'

quantus.metrics.robustness.relative_output_stability module

final class quantus.metrics.robustness.relative_output_stability.**RelativeOutputStability**(*nr_samples: int = 200, abs: bool = False, normalise: bool = False, normalise_func: Callable[[np.ndar np.ndarray] | None = None, normalise_func_kwarg Dict[str, ...] | None = None, perturb_func: Callable | None = None, perturb_func_kwargs: Dict[str, ...] | None = None, return_aggregate: bool = False, aggregate_func: Callable[[np.ndar np.float] | None = None, disable_warnings:*

Bases: *Metric*[List[float]]

Relative Output Stability leverages the stability of an explanation with respect to the change in the output logits.

ROS(x, x', ex, ex') = max frac{||frac{e_x - e_x'}{e_x}||_p} {max (||h(x) - h(x')||_p, epsilon_{min})},

where *h(x)* and *h(x')* are the output logits for *x* and *x'* respectively.

References:

1) Chirag Agarwal, et. al., 2022. "Rethinking stability for attribution based explanations.", https://arxiv.org/pdf/2203.06877.pdf

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, ...])	For each image *x*:
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(model, x_batch, y_batch, ...)	**Parameters**
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.
relative_output_stability_objective(h_x, ...)	Computes relative output stabilities maximization objective as defined here https://arxiv.org/pdf/2203.06877.pdf by the authors.

__call__(*model: tf.keras.Model | torch.nn.Module*, *x_batch: np.ndarray*, *y_batch: np.ndarray*, *model_predict_kwargs: Dict[str, ...] | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict[str, ...] | None = None*, *a_batch: np.ndarray | None = None*, *device: str | None = None*, *softmax: bool = False*, *channel_first: bool = True*, *batch_size: int = 64*, ***kwargs*) → List[float]

For each image *x*:

- Generate *num_perturbations* perturbed *xs* in the neighborhood of *x*.
- Compute explanations *e_x* and *e_xs*.
- Compute relative input output objective, find max value with respect to *xs*.
- In practise we just use *max* over a finite *xs_batch*.

Parameters

model: tf.keras.Model, torch.nn.Module
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
4D tensor representing batch of input images

y_batch: np.ndarray
1D tensor, representing predicted labels for the x_batch.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

explain_func: callable, optional
Function used to generate explanations.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

a_batch: np.ndarray, optional
4D tensor with pre-computed explanations for the x_batch.

device: str, optional
Device on which torch should perform computations.

softmax: boolean, optional
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

batch_size: int
The batch size to be used.

kwargs:
not used, deprecated

Returns
———-

relative output stability: float, np.ndarray
float in case *return_aggregate=True*, otherwise np.ndarray of floats

__init__(*nr_samples: int = 200*, *abs: bool = False*, *normalise: bool = False*, *normalise_func: Callable[[np.ndarray], np.ndarray] | None = None*, *normalise_func_kwargs: Dict[str, ...] | None = None*, *perturb_func: Callable | None = None*, *perturb_func_kwargs: Dict[str, ...] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable[[np.ndarray], np.float] | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, *eps_min: float = 1e-06*, *default_plot_func: Callable | None = None*, *return_nan_when_prediction_changes: bool = True*, ***kwargs*)

Parameters

nr_samples: int
The number of samples iterated, default=200.

abs: boolean
Indicates whether absolute operation is applied on the attribution.

normalise: boolean
Flag stating if the attributions should be normalised

normalise_func: callable
Attribution normalisation function applied in case normalise=True.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=gaussian_noise.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

default_plot_func: callable
Callable that plots the metrics result.

eps_min: float
Small constant to prevent division by 0 in relative_stability_objective, default 1e-6.

return_nan_when_prediction_changes: boolean
When set to true, the metric will be evaluated to NaN if the prediction changes after the perturbation is applied, default=True.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`data_applicability: ClassVar[Set[DataType]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}`

`evaluate_batch`(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → ndarray

Parameters

model: tf.keras.Model, torch.nn.Module
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
4D tensor representing batch of input images.

y_batch: np.ndarray
1D tensor, representing predicted labels for the x_batch.

a_batch: np.ndarray, optional
4D tensor with pre-computed explanations for the x_batch.

kwargs:
Unused.

Returns

ros: np.ndarray
A batch of explanations.

evaluation_category: `ClassVar[EvaluationCategory] = 'Robustness'`

evaluation_scores: `Any`

explain_func: `Callable`

explain_func_kwargs: `Dict[str, Any]`

model_applicability: `ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}`

name: `ClassVar[str] = 'Relative Output Stability'`

normalise_func: `Callable[[np.ndarray], np.ndarray] | None`

relative_output_stability_objective(*h_x: ndarray*, *h_xs: ndarray*, *e_x: ndarray*, *e_xs: ndarray*) → ndarray

Computes relative output stabilities maximization objective as defined here https://arxiv.org/pdf/2203.06877.pdf by the authors.

Parameters

h_x: np.ndarray
Output logits for x_batch.

h_xs: np.ndarray
Output logits for xs_batch.

e_x: np.ndarray
Explanations for x.

e_xs: np.ndarray
Explanations for xs.

Returns

ros_obj: np.ndarray
ROS maximization objective.

score_direction: `ClassVar[ScoreDirection] = 'lower'`

quantus.metrics.robustness.relative_representation_stability module

final class quantus.metrics.robustness.relative_representation_stability.**RelativeRepresentationStability**

Bases: *Metric*[List[float]]

Relative Representation Stability leverages the stability of an explanation with respect to the change in the output logits.

RRS(x, x', ex, ex') = max frac{||frac{e_x - e_{x'}}{e_x}||_p} {max (||frac{L_x - L_{x'}}{L_x}||_p, epsilon_{min})},

where *L(·)* denotes the internal model representation, e.g., output embeddings of hidden layers.

References:

1) Chirag Agarwal, et. al., 2022. "Rethinking stability for attribution based explanations.", https://arxiv.org/pdf/2203.06877.pdf

Attributes:

- _name: The name of the metric.
- _data_applicability: The data types that the metric implementation currently supports.
- _models: The model types that this metric can work with.
- score_direction: How to interpret the scores, whether higher/ lower values are considered better.
- evaluation_category: What property/ explanation quality that this metric measures.

Attributes

`disable_warnings`
A helper to avoid polluting test outputs with warnings.

`display_progressbar`
A helper to avoid polluting test outputs with tqdm progress bars.

`get_params`
List parameters of metric.

Methods

__call__(model, x_batch, y_batch[, ...])	For each image *x*:
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(model, x_batch, y_batch, ...)	**Parameters**
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.
relative_representation_stability_objecti	Computes relative representation stabilities maximization objective as defined here https://arxiv.org/pdf/2203.06877.pdf by the authors.

__call__(*model: tf.keras.Model | torch.nn.Module*, *x_batch: np.ndarray*, *y_batch: np.ndarray*, *model_predict_kwargs: Dict[str, ...] | None = None*, *explain_func: Callable | None = None*, *explain_func_kwargs: Dict[str, ...] | None = None*, *a_batch: np.ndarray | None = None*, *device: str | None = None*, *softmax: bool = False*, *channel_first: bool = True*, *batch_size: int = 64*, ***kwargs*) → List[float]

For each image *x*:

- Generate *num_perturbations* perturbed *xs* in the neighborhood of *x*.
- Compute explanations *e_x* and *e_xs*.
- Compute relative representation stability objective, find max value with respect to *xs*.
- In practise we just use *max* over a finite *xs_batch*.

Parameters

model: tf.keras.Model, torch.nn.Module
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
4D tensor representing batch of input images

y_batch: np.ndarray
1D tensor, representing predicted labels for the x_batch.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

explain_func: callable, optional
Function used to generate explanations.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

a_batch: np.ndarray, optional
4D tensor with pre-computed explanations for the x_batch.

device: str, optional
Device on which torch should perform computations.

softmax: boolean, optional
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

batch_size: int
The batch size to be used.

kwargs:
not used, deprecated

Returns
——-

relative representation stability: float, np.ndarray
float in case *return_aggregate=True*, otherwise np.ndarray of floats

__init__(*nr_samples: int = 200*, *abs: bool = False*, *normalise: bool = False*, *normalise_func: Callable[[np.ndarray], np.ndarray] | None = None*, *normalise_func_kwargs: Dict[str, ...] | None = None*, *perturb_func: Callable | None = None*, *perturb_func_kwargs: Dict[str, ...] | None = None*, *return_aggregate: bool = False*, *aggregate_func: Callable[[np.ndarray], np.ndarray] | None = None*, *disable_warnings: bool = False*, *display_progressbar: bool = False*, *eps_min: float = 1e-06*, *default_plot_func: Callable | None = None*, *layer_names: List[str] | None = None*, *layer_indices: List[int] | None = None*, *return_nan_when_prediction_changes: bool = True*, ***kwargs*)

Parameters

nr_samples: int
The number of samples iterated, default=200.

abs: boolean
Indicates whether absolute operation is applied on the attribution.

normalise: boolean
Flag stating if the attributions should be normalised

normalise_func: callable
Attribution normalisation function applied in case normalise=True.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call, default={}.

perturb_func: callable
Input perturbation function. If None, the default value is used, default=gaussian_noise.

perturb_func_kwargs: dict
Keyword arguments to be passed to perturb_func, default={}.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

disable_warnings: boolean
Indicates whether the warnings are printed, default=False.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed, default=False.

default_plot_func: callable
Callable that plots the metrics result.

eps_min: float
Small constant to prevent division by 0 in relative_stability_objective, default 1e-6.

layer_names: List[str], optional
Names of layers, representations of which should be used for RRS computation, default = all.

layer_indices: List[int], optional
Indices of layers, representations of which should be used for RRS computation, default = all.

return_nan_when_prediction_changes: boolean
When set to true, the metric will be evaluated to NaN if the prediction changes after the perturbation is applied, default=True.

a_axes: Sequence[int]

all_evaluation_scores: Any

data_applicability: ClassVar[Set[*DataType*]] = {DataType.IMAGE, DataType.TABULAR, DataType.TIMESERIES}

evaluate_batch(*model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → ndarray

Parameters

model: tf.keras.Model, torch.nn.Module
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
4D tensor representing batch of input images.

y_batch: np.ndarray
1D tensor, representing predicted labels for the x_batch.

a_batch: np.ndarray, optional
4D tensor with pre-computed explanations for the x_batch.

kwargs:
Unused.

Returns

ris: np.ndarray
The batched evaluation results.

`evaluation_category: ClassVar[EvaluationCategory] = 'Robustness'`

`evaluation_scores: Any`

`explain_func: Callable`

`explain_func_kwargs: Dict[str, Any]`

`model_applicability: ClassVar[Set[ModelType]] = {ModelType.TF, ModelType.TORCH}`

`name: ClassVar[str] = 'Relative Representation Stability'`

`normalise_func: Callable[[np.ndarray], np.ndarray] | None`

`relative_representation_stability_objective`(*l_x: ndarray*, *l_xs: ndarray*, *e_x: ndarray*, *e_xs: ndarray*) → ndarray

Computes relative representation stabilities maximization objective as defined here https://arxiv.org/pdf/2203.06877.pdf by the authors.

Parameters

l_x: np.ndarray
Internal representation for x_batch.

l_xs: np.ndarray
Internal representation for xs_batch.

e_x: np.ndarray
Explanations for x.

e_xs: np.ndarray
Explanations for xs.

Returns

rrs_obj: np.ndarray
RRS maximization objective.

`score_direction: ClassVar[ScoreDirection] = 'lower'`

Submodules

quantus.metrics.base module

This module implements the base class for creating evaluation metrics.

class quantus.metrics.base.**Metric**(*abs: bool*, *normalise: bool*, *normalise_func: Callable | None*, *normalise_func_kwargs: Dict[str, Any] | None*, *return_aggregate: bool*, *aggregate_func: Callable*, *default_plot_func: Callable | None*, *disable_warnings: bool*, *display_progressbar: bool*, ***kwargs*)

Bases: Generic[R]

Interface defining Metrics' API.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

__call__(model, x_batch, y_batch, a_batch, ...)	This implementation represents the main logic of the metric and makes the class object callable.
batch_preprocess(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
custom_batch_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
custom_postprocess(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
custom_preprocess(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
evaluate_batch(model, x_batch, y_batch, ...)	Evaluates model and attributes on a single data batch and returns the batched evaluation result.
explain_batch(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
general_preprocess(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
generate_batches(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
interpret_scores()	Get an interpretation of the scores.
plot([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

__call__(*model: keras.Model | nn.Module | None*, *x_batch: np.ndarray*, *y_batch: np.ndarray*, *a_batch: np.ndarray | None*, *s_batch: np.ndarray | None*, *channel_first: bool | None*, *explain_func: Callable | None*, *explain_func_kwargs: Dict | None*, *model_predict_kwargs: Dict | None*, *softmax: bool | None*, *device: str | None = None*, *batch_size: int = 64*, *custom_batch: Any = None*, ***kwargs*) → R

This implementation represents the main logic of the metric and makes the class object callable. It completes batch-wise evaluation of explanations (a_batch) with respect to input data (x_batch), output labels (y_batch) and a torch or tensorflow model (model).

Calls general_preprocess() with all relevant arguments, calls evaluate_instance() on each instance, and saves results to evaluation_scores. Calls custom_postprocess() afterwards. Finally returns evaluation_scores.

The content of evaluation_scores will be appended to all_evaluation_scores (list) at the end of the evaluation call.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean

Indicates whether to use softmax probabilities or logits in model prediction.
This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

custom_batch: any
Any object that can be passed to the evaluation process. Gives flexibility to the user to adapt for implementing their own metric.

kwargs: optional
Keyword arguments.

Returns

evaluation_scores: list
a list of Any with the evaluation scores of the concerned batch.

Examples:

Minimal imports. >> import quantus >> from quantus import LeNet >> import torch

Enable GPU. >> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

Load a pre-trained LeNet classification model (architecture at quantus/helpers/models). >> model = LeNet() >> model.load_state_dict(torch.load("tutorials/assets/pytests/mnist_model"))

Load MNIST datasets and make loaders. >> test_set = torchvision.datasets.MNIST(root='./sample_data', download=True) >> test_loader = torch.utils.data.DataLoader(test_set, batch_size=24)

Load a batch of inputs and outputs to use for XAI evaluation. >> x_batch, y_batch = iter(test_loader).next() >> x_batch, y_batch = x_batch.cpu().numpy(), y_batch.cpu().numpy()

Generate Saliency attributions of the test set batch of the test set. >> a_batch_saliency = Saliency(model).attribute(inputs=x_batch, target=y_batch, abs=True).sum(axis=1) >> a_batch_saliency = a_batch_saliency.cpu().numpy()

Initialise the metric and evaluate explanations by calling the metric instance. >> metric = Metric(abs=True, normalise=False) >> scores = metric(model=model, x_batch=x_batch, y_batch=y_batch, a_batch=a_batch_saliency)

__init__(*abs: bool*, *normalise: bool*, *normalise_func: Callable | None*, *normalise_func_kwargs: Dict[str, Any] | None*, *return_aggregate: bool*, *aggregate_func: Callable*, *default_plot_func: Callable | None*, *disable_warnings: bool*, *display_progressbar: bool*, ***kwargs*)

Initialise the Metric base class.

Each of the defined metrics in Quantus, inherits from Metric base class.

A child metric can benefit from the following class methods: - __call__(): Will call general_preprocess(), apply evaluate_instance() on each

instance and finally call custom_preprocess(). To use this method the child Metric needs to implement evaluate_instance().

- **general_preprocess(): Prepares all necessary data structures for evaluation.**
  Will call custom_preprocess() at the end.

The content of evaluation_scores will be appended to all_evaluation_scores (list) at the end of the evaluation call.

Parameters

abs: boolean
Indicates whether absolute operation is applied on the attribution.

normalise: boolean
Indicates whether normalise operation is applied on the attribution.

normalise_func: callable
Attribution normalisation function applied in case normalise=True.

normalise_func_kwargs: dict
Keyword arguments to be passed to normalise_func on call.

return_aggregate: boolean
Indicates if an aggregated score should be computed over all instances.

aggregate_func: callable
Callable that aggregates the scores given an evaluation call.

default_plot_func: callable
Callable that plots the metrics result.

disable_warnings: boolean
Indicates whether the warnings are printed.

display_progressbar: boolean
Indicates whether a tqdm-progress-bar is printed.

kwargs: optional
Keyword arguments.

`a_axes: Sequence[int]`

`all_evaluation_scores: Any`

`final batch_preprocess`(*data_batch: Dict[str, Any]*) → Dict[str, Any]

If *data_batch* has no *a_batch*, will compute explanations. This needs to be done on batch level to avoid OOM. Additionally will set *a_axes* property if it is None, this can be done earliest after we have first *a_batch*.

Parameters

data_batch:
A single entry yielded from the generator return by *self.generate_batches(...)*

Returns

data_batch:
Dictionary, which is ready to be passed down to *self.evaluate_batch*.

`custom_batch_preprocess`(*, *model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray*, ***kwargs*) → Dict[str, Any] | None

Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.

Parameters

model:
A model that is subject to explanation.

x_batch:
A np.ndarray which contains the input data that are explained.

y_batch:
A np.ndarray which contains the output labels that are explained.

a_batch:
A np.ndarray which contains pre-computed attributions i.e., explanations.

kwargs:
Optional, metric-specific parameters.

Returns

dict:
Optional dictionary with additional kwargs, which will be passed to *self.evaluate_batch(…)*

`custom_postprocess`(*, *model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray | None*, *a_batch: ndarray | None*, *s_batch: ndarray | None*, ***kwargs*)

Implement this method if you need custom postprocessing of results or additional attributes.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model e.g., torchvision.models that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

kwargs: any, optional
Additional data which was created in custom_preprocess().

Returns

any:
Can be implemented, optionally by the child class.

`custom_preprocess`(*, *model:* ModelInterface, *x_batch: ndarray*, *y_batch: ndarray*, *a_batch: ndarray | None*, *s_batch: ndarray | None*, *custom_batch: Any*, ***kwargs*) → Dict[str, Any] | None

Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.

If this method returns a dictionary, the keys (string) will be used as additional arguments for evaluate_instance(). If the key ends with *_batch*, this suffix will be removed from the respective argument name when passed to evaluate_instance(). If they key corresponds to the arguments *x_batch, y_batch, a_batch, s_batch*, these will be overwritten for passing *x, y, a, s* to *evaluate_instance()*. If this method returns None, no additional keyword arguments will be passed to *evaluate_instance()*.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model e.g., torchvision.models that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

custom_batch: any
Gives flexibility to the inheriting metric to use for evaluation, can hold any variable.

kwargs:
Optional, metric-specific parameters.

Returns

dict, optional

A dictionary which holds (optionally additional) preprocessed data to

be included when calling *evaluate_instance()*.

Examples

Custom Metric definition with additional keyword argument used in evaluate_instance(): >>> def custom_preprocess(>>> self, >>> model: ModelInterface, >>> x_batch: np.ndarray, >>> y_batch: Optional[np.ndarray], >>> a_batch: Optional[np.ndarray], >>> s_batch: np.ndarray, >>> custom_batch: Optional[np.ndarray], >>>) -> Dict[str, Any]: >>> return {'my_new_variable': np.mean(x_batch)} >>> >>> def evaluate_instance(>>> self, >>> model: ModelInterface, >>> x: np.ndarray, >>> y: Optional[np.ndarray], >>> a: Optional[np.ndarray], >>> s: np.ndarray, >>> my_new_variable: np.float, >>>) -> float:

Custom Metric definition with additional keyword argument that ends with *_batch* >>> def custom_preprocess(>>> self, >>> model: ModelInterface, >>> x_batch: np.ndarray, >>> y_batch: Optional[np.ndarray], >>> a_batch: Optional[np.ndarray], >>> s_batch: np.ndarray, >>> custom_batch: Optional[np.ndarray], >>>) -> Dict[str, Any]: >>> return {'my_new_variable_batch': np.arange(len(x_batch))} >>> >>> def evaluate_instance(>>> self, >>> model: ModelInterface, >>> x: np.ndarray, >>> y: Optional[np.ndarray], >>> a: Optional[np.ndarray], >>> s: np.ndarray, >>> my_new_variable: np.int, >>>) -> float:

Custom Metric definition with transformation of an existing # keyword argument from *evaluate_instance()* >>> def custom_preprocess(>>> self, >>> model: ModelInterface, >>> x_batch: np.ndarray, >>> y_batch: Optional[np.ndarray], >>> a_batch: Optional[np.ndarray], >>> s_batch: np.ndarray, >>> custom_batch: Optional[np.ndarray], >>>) -> Dict[str, Any]: >>> return {'x_batch': x_batch - np.mean(x_batch, axis=0)} >>> >>> def evaluate_instance(>>> self, >>> model: ModelInterface, >>> x: np.ndarray, >>> y: Optional[np.ndarray], >>> a: Optional[np.ndarray], >>> s: np.ndarray, >>>) -> float:

Custom Metric definition with None returned in custom_preprocess(), # but with inplace-preprocessing and additional assertion. >>> def custom_preprocess(>>> self, >>> model: ModelInterface, >>> x_batch: np.ndarray, >>> y_batch: Optional[np.ndarray], >>> a_batch: Optional[np.ndarray], >>> s_batch: np.ndarray, >>> custom_batch: Optional[np.ndarray], >>>) -> None: >>> if np.any(np.all(a_batch < 0, axis=0)): >>> raise ValueError("Attributions must not be all negative") >>> >>> x_batch -= np.mean(x_batch, axis=0) >>> >>> return None >>> >>> def evaluate_instance(>>> self, >>> model: ModelInterface, >>> x: np.ndarray, >>> y: Optional[np.ndarray], >>> a: Optional[np.ndarray], >>> s: np.ndarray, >>>) -> float:

data_applicability: ClassVar[Set[*DataType*]]

property disable_warnings: bool

A helper to avoid polluting test outputs with warnings.

property display_progressbar: bool

A helper to avoid polluting test outputs with tqdm progress bars.

abstract evaluate_batch(*model: ModelInterface*, *x_batch: np.ndarray*, *y_batch: np.ndarray*, *a_batch: np.ndarray*, *s_batch: np.ndarray | None*, ***kwargs*) → R

Evaluates model and attributes on a single data batch and returns the batched evaluation result.

This method needs to be implemented to use __call__().

Parameters

model: ModelInterface
A ModelInteface that is subject to explanation.

x_batch: np.ndarray
The input to be evaluated on a batch-basis.

y_batch: np.ndarray
The output to be evaluated on a batch-basis.

a_batch: np.ndarray
The explanation to be evaluated on a batch-basis.

s_batch: np.ndarray
The segmentation to be evaluated on a batch-basis.

Returns

np.ndarray
The batched evaluation results.

evaluation_category: ClassVar[*EvaluationCategory*]

evaluation_scores: Any

final explain_batch(*model: ModelInterface | keras.Model | nn.Module*, *x_batch: np.ndarray*, *y_batch: np.ndarray*) → np.ndarray

Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and postprocessing approach. It will do few things:

- call model.shape_input (if ModelInterface instance was provided)
- unwrap model (if ModelInterface instance was provided)
- call explain_func
- expand attribution channel
- (optionally) normalise a_batch
- (optionally) take np.abs of a_batch

Parameters

model:
A model that is subject to explanation.

x_batch:
A np.ndarray which contains the input data that are explained.

y_batch:
A np.ndarray which contains the output labels that are explained.

Returns

a_batch:
Batch of explanations ready to be evaluated.

`explain_func: Callable`

`explain_func_kwargs: Dict[str, Any]`

`final general_preprocess`(*model: keras.Model | nn.Module | None*, *x_batch: np.ndarray*, *y_batch: np.ndarray*, *a_batch: np.ndarray | None*, *s_batch: np.ndarray | None*, *channel_first: bool | None*, *explain_func: Callable*, *explain_func_kwargs: Dict[str, Any] | None*, *model_predict_kwargs: Dict[str, Any] | None*, *softmax: bool*, *device: str | None*, *custom_batch: np.ndarray | None*) → Dict[str, Any]

Prepares all necessary variables for evaluation.

- Reshapes data to channel first layout.
- Wraps model into ModelInterface.
- Creates attributions if necessary.
- Expands attributions to data shape (adds channel dimension).
- Calls custom_preprocess().
- Normalises attributions if desired.
- Takes absolute of attributions if desired.
- If no segmentation s_batch given, creates list of Nones with as many elements as there are data instances.
- If no custom_batch given, creates list of Nones with as many elements as there are data instances.

Parameters

model: torch.nn.Module, tf.keras.Model
A torch or tensorflow model e.g., torchvision.models that is subject to explanation.

x_batch: np.ndarray
A np.ndarray which contains the input data that are explained.

y_batch: np.ndarray
A np.ndarray which contains the output labels that are explained.

a_batch: np.ndarray, optional
A np.ndarray which contains pre-computed attributions i.e., explanations.

s_batch: np.ndarray, optional
A np.ndarray which contains segmentation masks that matches the input.

channel_first: boolean, optional
Indicates of the image dimensions are channel first, or channel last. Inferred from the input shape if None.

explain_func: callable
Callable generating attributions.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call.

model_predict_kwargs: dict, optional
Keyword arguments to be passed to the model's predict method.

softmax: boolean
Indicates whether to use softmax probabilities or logits in model prediction. This is used for this __call__ only and won't be saved as attribute. If None, self.softmax is used.

device: string
Indicated the device on which a torch.Tensor is or will be allocated: "cpu" or "gpu".

custom_batch: any
Gives flexibility ot the user to use for evaluation, can hold any variable.

Returns

tuple
A general preprocess.

`final generate_batches`(*data: D*, *batch_size: int*) → Generator[D, None, None]

Creates iterator to iterate over all batched instances in data dictionary. Each iterator output element is a keyword argument dictionary with string keys.

Each item key in the input data dictionary has to be of type string. - If the item value is not a sequence, the respective item key/value pair

will be written to each iterator output dictionary.

- If the item value is a sequence and the item key ends with '_batch', a check will be made to make sure length matches number of instances. The values of the batch instances in the sequence will be added to the respective iterator output dictionary with the '_batch' suffix removed.
- If the item value is a sequence but doesn't end with '_batch', it will be treated as a simple value and the respective item key/value pair will be written to each iterator output dictionary.

Parameters

data: dict[str, any]
The data input dictionary.

batch_size: int
The batch size to be used.

Returns

iterator:
Each iterator output element is a keyword argument dictionary (string keys).

`property get_params: Dict[str, Any]`

List parameters of metric.

Returns

dict:
A dictionary with attributes if not excluded from pre-determined list.

`interpret_scores`()

Get an interpretation of the scores.

`model_applicability: ClassVar[Set[ModelType]]`

`name: ClassVar[str]`

`normalise_func: Callable[[ndarray], ndarray] | None`

plot(*plot_func: Callable | None = None*, *show: bool = True*, *path_to_save: str | None = None*, **args*, ***kwargs*) → None

Basic plotting functionality for Metric class. The user provides a plot_func (Callable) that contains the actual plotting logic (but returns None).

Parameters

plot_func: callable
A Callable with the actual plotting logic. Default set to None, which implies default_plot_func is set.

show: boolean
A boolean to state if the plot shall be shown.

path_to_save: (str)
A string that specifies the path to save file.

args: optional
An optional with additional arguments.

kwargs: optional
An optional dict with additional arguments.

Returns

None

score_direction: ClassVar[*ScoreDirection*]

quantus.metrics.base_batched module

class quantus.metrics.base_batched.**BatchedMetric**(**args*, ***kwargs*)

Bases: *Metric*, ABC

Alias to quantus.Metric, will be removed in next major release.

Attributes

disable_warnings
A helper to avoid polluting test outputs with warnings.

display_progressbar
A helper to avoid polluting test outputs with tqdm progress bars.

get_params
List parameters of metric.

Methods

`__call__`(model, x_batch, y_batch, a_batch, ...)	This implementation represents the main logic of the metric and makes the class object callable.
`batch_preprocess`(data_batch)	If *data_batch* has no *a_batch*, will compute explanations.
`custom_batch_preprocess`(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data or simply for creating/initialising additional attributes or assertions before a *data_batch* can be evaluated.
`custom_postprocess`(*, model, x_batch, ...)	Implement this method if you need custom postprocessing of results or additional attributes.
`custom_preprocess`(*, model, x_batch, ...)	Implement this method if you need custom preprocessing of data, model alteration or simply for creating/initialising additional attributes or assertions.
`evaluate_batch`(model, x_batch, y_batch, ...)	Evaluates model and attributes on a single data batch and returns the batched evaluation result.
`explain_batch`(model, x_batch, y_batch)	Compute explanations, normalise and take absolute (if was configured so during metric initialization.) This method should primarily be used if you need to generate additional explanation in metrics body. It encapsulates typical for Quantus pre- and post-processing approach. It will do few things: - call model.shape_input (if ModelInterface instance was provided) - unwrap model (if ModelInterface instance was provided) - call explain_func - expand attribution channel - (optionally) normalise a_batch - (optionally) take np.abs of a_batch.
`general_preprocess`(model, x_batch, y_batch, ...)	Prepares all necessary variables for evaluation.
`generate_batches`(data, batch_size)	Creates iterator to iterate over all batched instances in data dictionary.
`interpret_scores`()	Get an interpretation of the scores.
`plot`([plot_func, show, path_to_save])	Basic plotting functionality for Metric class.

a_axes: Sequence[int]

all_evaluation_scores: Any

data_applicability: ClassVar[Set[*DataType*]]

evaluation_category: ClassVar[*EvaluationCategory*]

evaluation_scores: Any

explain_func: Callable

explain_func_kwargs: Dict[str, Any]

model_applicability: ClassVar[Set[*ModelType*]]

name: ClassVar[str]

normalise_func: Callable[[np.ndarray], np.ndarray] | None

score_direction: ClassVar[*ScoreDirection*]

1.3.2 Submodules

quantus.evaluation module

This module provides some functionality to evaluate different explanation methods on several evaluation criteria.

quantus.evaluation.**evaluate**(*metrics: ~typing.Dict, xai_methods: ~typing.Dict[str, ~typing.Callable] | ~typing.Dict[str, ~typing.Dict] | ~typing.Dict[str, ~numpy.ndarray], model: ~quantus.helpers.model.model_interface.ModelInterface, x_batch: ~numpy.ndarray, y_batch: ~numpy.ndarray, s_batch: ~numpy.ndarray | None = None, agg_func: ~typing.Callable = <function <lambda>>, explain_func_kwargs: dict | None = None, call_kwargs: ~typing.Dict | ~typing.Dict[str, ~typing.Dict] | None = None, return_as_df: bool | None = None, verbose: bool | None = None, progress: bool | None = None, *args, **kwargs*) → dict | None

Evaluate different explanation methods using specified metrics.

Parameters

metrics
[dict] A dictionary of initialized evaluation metrics. See quantus.AVAILABLE_METRICS. Example: {'Robustness': quantus.MaxSensitivity(), 'Faithfulness': quantus.PixelFlipping()}

xai_methods
[dict] A dictionary specifying the explanation methods to evaluate, which can be structured in three ways:

1) Dict[str, Dict] for built-in Quantus methods (using quantus.explain):

Example: xai_methods = {

'IntegratedGradients': {
'n_steps': 10, 'xai_lib': 'captum'

}, 'Saliency': {

'xai_lib': 'captum'

}

}

- See quantus.AVAILABLE_XAI_METHODS_CAPTUM for supported captum methods.
- See quantus.AVAILABLE_XAI_METHODS_TF for supported tensorflow methods.
- See https://github.com/chr5tphr/zennit for supported zennit methods.
- Read more about the explanation function arguments here: <https://quantus.readthedocs.io/en/latest/docs_api/quantus.functions.explanation_func.html#quantus.functions.explanation_func.explain>

2) Dict[str, Callable] for custom methods:

Example: xai_methods = {

'custom_own_xai_method': custom_explain_function

} or ai_methods = {"InputXGradient": {

"explain_func": quantus.explain, "explain_func_kwargs": {},

}}

- Here, you can provide your own callable that mirrors the input and outputs of the quantus.explain() method.

3) Dict[str, np.ndarray] for pre-calculated attributions:

Example: xai_methods = {

'LIME': precomputed_numpy_lime_attributions, 'GradientShap': precomputed_numpy_shap_attributions

}

- Note that some Quantus metrics, e.g., quantus.MaxSensitivity() within the robustness

category, includes "re-explaning" the input and output pair as a part of the evaluation metric logic. If you include such metrics in the quantus.evaluate(), this option will not be possible.

It is also possible to pass a combination of the above.

```
>>> xai_methods = {
>>>     'IntegratedGradients': {
>>>         'n_steps': 10,
>>>         'xai_lib': 'captum'
>>>     },
>>>     'Saliency': {
>>>         'xai_lib': 'captum'
>>>     },
>>>     'custom_own_xai_method': custom_explain_function,
>>>     'LIME': precomputed_numpy_lime_attributions,
>>>     'GradientShap': precomputed_numpy_shap_attributions
>>> }
```

model: Union[torch.nn.Module, tf.keras.Model]
A torch or tensorflow model that is subject to explanation.

x_batch: np.ndarray
A np.ndarray containing the input data to be explained.

y_batch: np.ndarray
A np.ndarray containing the output labels corresponding to x_batch.

s_batch: np.ndarray, optional
A np.ndarray containing segmentation masks that match the input.

agg_func: Callable
Indicates how to aggregate scores, e.g., pass np.mean.

explain_func_kwargs: dict, optional
Keyword arguments to be passed to explain_func on call. Pass None if using Dict[str, Dict] type for xai_methods.

call_kwargs: Dict[str, Dict]
Keyword arguments for the call of the metrics. Keys are names for argument sets, and values are argument dictionaries.

verbose: optional, bool
Indicates whether to print evaluation progress.

progress: optional, bool
Deprecated. Indicates whether to print evaluation progress. Use verbose instead.

return_as_df: optional, bool
Indicates whether to return the results as a pd.DataFrame. Only works if call_kwargs is not passed.

args: optional
Deprecated arguments for the call.

kwargs: optional
Deprecated keyword arguments for the call of the metrics.

Returns

results: dict
A dictionary with the evaluation results.

1.4 Contribute to Quantus

Thank you for taking interest in contributions to Quantus! We encourage you to contribute new features/metrics, optimisations, refactorings or report any bugs you may come across. In this guide, you will get an overview of the workflow and best practices for contributing to Quantus.

Questions. If you have any developer-related questions, please open an issue or write us at hedstroem.anna@gmail.com.

1.4.1 Table of Contents

1.4.2 Reporting Bugs

If you discover a bug, as a first step please check the existing Issues to see if this bug has already been reported. In case the bug has not been reported yet, please do the following:

- Open an issue.
- Add a descriptive title to the issue and write a short summary of the problem.
- Adding more context, including reference to the problematic parts of the code, would be very helpful to us.

Once the bug is reported, our development team will try to address the issue as quickly as possible.

1.4.3 General Guide to Making Changes

This is a general guide to contributing changes to Quantus. If you would like to add a new evaluation metric to Quantus, please refer to *Contributing a New Metric*. Before you start the development work, make sure to read our documentation first.

Development Installation

Make sure to install the latest version of Quantus from the main branch.

```
git clone https://github.com/understandable-machine-intelligence-lab/Quantus.git
cd quantus
# Tox will provision dev environment with editable installation for you.
python3 -m pip install tox
python3 -m tox devenv
source venv/bin/activate
```

Branching

Before you start making changes to the code, create a local branch from the latest version of `main`.

Code Style

Code is written to follow PEP-8 and for docstrings we use numpydoc. We use flake8 for quick style checks and black for code formatting with a line-width of 88 characters per line.

Unit Tests

Tests are written using pytest and executed together with codecov for coverage reports. We use tox for test automation. For complete list of CLI commands, please refer to tox - CLI interface. To perform the tests for all supported python versions execute the following CLI command:

```
python3 -m tox run
```

... alternatively, to get additionally coverage details, run:

```
python3 -m tox run -e coverage
```

It is possible to limit the scope of testing to specific sections of the codebase, for example, only test the Faithfulness metrics using python3.10:

```
python3 -m tox run -e py310 -- -m faithfulness -s
```

For a complete overview of the possible testing scopes, please refer to `pytest.ini`.

Documentation

Make sure to add docstrings to every class, method and function that you add to the codebase. The docstring should include a description of all parameters and returns. Use the existing documentation as an example. TODO: Automatic docstring generation.

Before You Create a Pull Request

Before creating a PR, double-check that the following tasks are completed:

- Make sure that the latest version of the code from the `main` branch is merged into your working branch.
- Run `black` to format source code:

```
black quantus/INSERT_YOUR_FILE_NAME.py
```

- Run `flake8` for quick style checks, e.g.:

```
flake8 quantus/INSERT_YOUR_FILE_NAME.py
```

- Create a unit test for new functionality and add under `tests/` folder, add `@pytest.mark` with fitting category.
- If newly added test cases include a new category of `@pytest.mark` then add that category with description to `pytest.ini`
- Make sure all unit tests pass for all supported python version by running:

```
python3 -m tox run
```

- Generally, every change should be covered with respective test-case, we aim at ~100% code coverage in Quantus, you can verify it by running:

```
python3 -m tox run -e coverage
```

Pull Requests

Once you are done with the changes:

- Create a pull request
- Provide a summary of the changes you are introducing.
- In case you are resolving an issue, don't forget to link it.
- Add annahedstroem as a reviewer.

1.4.4 Contributing a New Metric

We always welcome extensions to our collection of evaluation metrics. This short description provides a guideline to introducing a new metric into Quantus. We strongly encourage you to take an example from already implemented metrics.

Theoretical Foundations

Currently, we support six subgroups of evaluation metrics:

- Faithfulness
- Robustness
- Localisation
- Complexity
- Randomisation
- Axiomatic

See a more detailed description of those in README. Identify which category your metric belongs to and create a Python file for your metric class in the respective folder in `quantus/metrics`.

Add the metric to the `__init__.py` file in the respective folder.

Metric Class

Every metric class inherits from the base `Metric` class: `quantus/metrics/base.py`. Importantly, Faithfulness and Robustness inherit not from the `Metric` class directly, but rather from its child `PerturbationMetric`.

A child metric can benefit from the following class methods:

- `__call__()`: Will call general_preprocess(), apply() on each instance and finally call custom_preprocess(). To use this method the child Metric needs to implement evaluate_instance().
- `general_preprocess()`: Prepares all necessary data structures for evaluation. Will call custom_preprocess() at the end.

The following methods are expected to be implemented in the metric class:

- `__init__()`: Initialize the metric.
- `__call__()`: Typically, calls `__call__()` in the base class.
- `evaluate_instance()`: Gets model and data for a single instance as input, returns evaluation result.

The following methods are optimal for implementation:

- `custom_preprocess()`: In case `general_preprocess()` from base class is not sufficient, additional preprocessing steps can be added here. This method must return a dictionary with string keys or None. If a dictionary is returned, additional keyword arguments can be used in `evaluate_instance()`. Please make sure to read the docstring of `custom_preprocess()` for further instructions on how to appropriately name the variables that are created in the function.
- `custom_postprocess()`: Additional postprocessing steps can be added here that is added on top of the resuling evaluation scores.

For computational efficiency gains, it might be wise to consider using the `BatchedMetric` or `BatchedPerturbationMetric` when implementing your new metric. Details on the specific implementation requirements can be found in the respective class method, please see: quantus.readthedocs.io.

Using Helpers

In the `quantus/helpers` folder, you might find functions relevant to your implementation. Use search function and go through the function docstrings to explore your options.

If you find yourself developing some functionality of more general scope, consider adding this code to a respective file, or creating a new module in `quantus/helpers`.

Warnings

The `__init__()` method of a metric class typically call a warning that includes the following information:

- Metric name
- Sensitive parameters
- Proper citation of the source paper (!)

Documenting a Metric

Declaration of a method class should be followed by:

- A detailed description of the metric
- References
- Assumptions

Otherwise, please remember to add a description for all parameters and returns of each new method/function, as well as a description of the purpose of the method/function itself.

1.4.5 License

Please note that by contributing to the project you agree that it will be licensed under the License.

1.4.6 Questions

If you have any developer-related questions, please open an issue or write us at hedstroem.anna@gmail.com.

1.5 User guidelines

Just 'throwing' some metrics at your explanations and considering the job done is not a very productive approach. Before evaluating your explanations, make sure to:

- Always read the original publication to understand the context that the metric was introduced in - it may differ from your specific task and/ or data domain
- Spend time on understanding and investigating how the hyperparameters of metrics can influence the evaluation outcome. Some parameters that usually influence results significantly include:
  - the choice of perturbation function
  - whether normalisation is applied and the choice of the normalisation function
  - whether unsigned or signed attributions are considered

- Establish evidence that your chosen metric is well-behaved in your specific setting, e.g., include a random explanation (as a control variant) to verify the metric
- Reflect on the metric's underlying assumptions, e.g., most perturbation-based metrics don't account for nonlinear interactions between features
- Ensure that your model is well-trained, as a poor behaving model, e.g., a non-robust model will have useless explanations
- Each metric measures different properties of explanations, and especially the various categories (faithfulness, localisation, ...) can be viewed as different facettes of evaluation, but a single metric never suffices as a sole criterion for the quality of an explanation method

1.6 Disclaimers

1. Implementation may differ from the original author(s)

Note that the implementations of metrics in this library have not been verified by the original authors. Thus any metric implementation in this library may differ from the original authors. It is moreover likely that differences exist since

- the source code of original publication is most often not made publicly available
- sometimes the mathematical definition of the metric is missing
- the description of hyperparameter choice was left out.

This leaves room for (subjective) interpretations.

2. Discrepancy in operationalisation is likely

Metrics for XAI methods are often empirical interpretations (or translations) of qualities that researcher(s) stated were important for explanations to fulfil. Hence there may be a discrepancy between what the author claims to measure by the proposed metric and what is actually measured, e.g., using entropy as an operationalisation of explanation complexity.

3. Hyperparameters may (and should) change depending on the application/ task and dataset/ domain

Metrics are often designed with a specific use case in mind, most commonly for an image classification setting. Thus it is not always clear how to change the hyperparameters to make them suitable for another setting. Pay careful attention to how your hyperparameters should be tuned and what a proper baseline value could be in your context

4. Evaluation of explanations must be understood in its context; its application and of its kind

What evaluation metric to use can depend on the following factors:

- **The type of explanation:** e.g., an explanation by example cannot be evaluated the same way as attribution-based or feature-importance methods
- **The application/ task:** we may not require the explanations to fulfil certain criteria in some context compared to others, e.g., multi-label vs. single label classification
- **The dataset/ domain:** e.g, text vs. images, or if different dependency structures between features exist, as well as the preprocessing of the data, leading to differences on what the model may perceive, and how attribution methods can react to that
- **The user:** most evaluation metrics are founded from principles of what a user may expect from explanations, even in the seemingly objective measures. E.g., localisation asks for the explanation to be focused on objects expected to be important, and may fail independent of the explanation if the model simply does not consider those objects, while robustness asks to explain similarly over things we think looks similar, not considering how the model represents the data manifold etc. Thus it is important to define what attribution quality means for each experimental setting.

5. Evaluation (and explanations) can be unreliable if the model is not robust

Evaluation can fail (depending on the evaluation method) if you explain a poorly trained model. If the model is not robust, then explanations cannot be expected to be meaningful or interpretable [1, 2]. If the model achieves high predictive performance, but for the wrong reasons (e.g., Clever Hans effects, Backdoor issues) [3, 4], unexpected effects on localisation metrics are likely.

6. Evaluation outcomes can be true to the data or true to the model

Generally, explanations should depend on both the data and the model. However, both are difficult to measure at the same time, and the interpretation of evaluation outcomes will differ depending on whether we prioritise that attributions are faithful to data or to the model [5, 6]. As explained in [5], imagine if a model is trained to use only one of two highly correlated features. The explanation might then rightly point out that this one feature is important (and that the other correlated feature is not). But if we were to re-train the model, the model might now pick the other feature as basis for prediction, for which the explanation will consequently tell another story — that the other feature is important. Since the explanation function have returned conflicting information about what features are important — we might now believe that the explanation function in itself is unstable. But this may not necessarily be true — in this case, the explanation has remained faithful to the model but not the data. As such, in the context of evaluation, to avoid misinterpretation of results, it may therefore be important to articulate what you care most about explaining.

References

[1] P. Chalasani, J. Chen, A. R. Chowdhury, X. Wu, and S. Jha, "Concise explanations of neural networks using adversarial training," in Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, pp. 1383–1391, 2020.

[2] N. Bansal, C. Agarwal, and A. Nguyen, "SAM: the sensitivity of attribution methods to hyperparameters," in 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2020, Seattle, WA, USA, June 14-19, 2020. Computer Vision Foundation IEEE, pp. 11–21, 2020.

[3] S. Lapuschkin, S. Wäldchen, A. Binder, G. Montavon, W. Samek, and K.-R. Müller, "Unmasking clever hans predictors and assessing what machines really learn," Nature Communications, vol. 10, p. 1096, 2019.

[4] C. J. Anders, L. Weber, D. Neumann, W. Samek, K.-R. Müller, and S. Lapuschkin, "Finding and removing clever hans: Using explanation methods to debug and improve deep models," Information Fusion, vol. 77, pp. 261–295, 2022.

[5] P. Sturmfels, S. Lundberg, and S. Lee. "Visualizing the impact of feature attribution baselines." Distill 5, no. 1: e22, 2020.

[6] D. Janzing, L. Minorics, and P. Blöbaum. "Feature relevance quantification in explainable AI: A causal problem." In International Conference on Artificial Intelligence and Statistics, pp. 2907-2916. PMLR, 2020.

1.7 Citation

If you find this toolkit or its companion paper **Quantus: An Explainable AI Toolkit for Responsible Evaluation of Neural Network Explanations** interesting or useful in your research, please use the following Bibtex annotation to cite us:

```
@article{hedstrom2023quantus,
  author  = {Anna Hedstr{\"{o}}m and Leander Weber and Daniel Krakowczyk and Dilyara␣
↪Bareeva and Franz Motzkus and Wojciech Samek and Sebastian Lapuschkin and Marina␣
↪Marina M.{-}C. H{\"{o}}hne},
  title   = {Quantus: An Explainable AI Toolkit for Responsible Evaluation of Neural␣
↪Network Explanations and Beyond},
  journal = {Journal of Machine Learning Research},
  year    = {2023},
```

(continues on next page)

(continued from previous page)

```
  volume  = {24},
  number  = {34},
  pages   = {1--11},
  url     = {http://jmlr.org/papers/v24/22-0142.html}
}
```

When applying the individual metrics of Quantus, please make sure to also properly cite the work of the original authors. You can find the relevant citations in the documentation of each respective metric here.

PYTHON MODULE INDEX

q

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W