pymia Documentation

Release 0.3.1

Fabian Balsiger and Alain Jungo

Oct 12, 2021

GETTING STARTED

pymia is an open-source Python (py) package for deep learning-based medical image analysis (mia). The package addresses two main parts of deep learning pipelines: data handling and evaluation. The package itself is independent of the deep learning framework used but can easily be integrated into TensorFlow and PyTorch pipelines. Therefore, pymia is highly flexible, allows for fast prototyping, and reduces the burden of implementing data handling and evaluation.

The main features of pymia are data handling (`pymia.data` package) and evaluation (`pymia.evaluation` package). The intended use of pymia in the deep learning environment is depicted in Fig. 1. The data package is used to extract data (images, labels, demography, etc.) from a dataset in the desired format (2-D, 3-D; full- or patch-wise) for feeding to a neural network. The output of the neural network is then assembled back to the original format before extraction, if necessary. The evaluation package provides both evaluation routines as well as metrics to assess predictions against references. Evaluation can be used both for stand-alone result calculation and reporting, and for monitoring of the training progress. Further, pymia provides some basic image filtering and manipulation functionality (`pymia.filtering` package). We recommend following our *examples*.

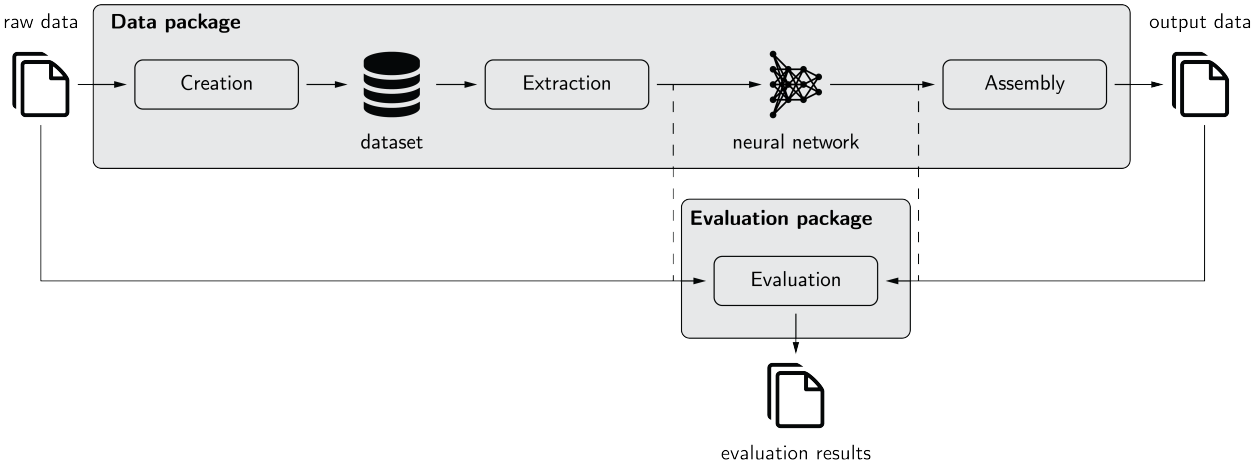


Fig. 1: The pymia package in the deep learning environment. The data package allows to create a dataset from raw data. Extraction of the data from this dataset is possible in nearly every desired format (2-D, 3-D; full- or patch-wise) for feeding to a neural network. The prediction of the neural network can, if necessary, be assembled back to the format before extraction. The evaluation package allows to evaluate predictions against references using a vast amount of metrics. It can be used stand-alone (solid) or for performance monitoring during training (dashed).

CHAPTER

ONE

GETTING STARTED

If you are new to pymia, here are a few guides to get you up to speed right away.

1.1 Installation

Install pymia using pip (e.g., within a Python virtual environment):

```
pip install pymia
```

Alternatively, you can download or clone the code from GitHub and install pymia by

```
git clone https://github.com/rundherum/pymia
cd pymia
python setup.py install
```

1.1.1 Dependencies

pymia requires Python 3.6 (or higher) and depends on the following packages:

- h5py
- NumPy
- scikit-image
- SciPy
- SimpleITK

Note: For the *pymia.data* package, not all dependencies are installed directly due to their heaviness. Meaning, you need to either manually install PyTorch by

- `pip install torch`

or TensorFlow by

- `pip install tensorflow`

depending on your preferred deep learning framework when using the *pymia.data* package. Upon loading a module from the *pymia.data* package, pymia will always check if the required dependencies are fulfilled.

1.1.2 Building the documentation

Building the documentation requires the following packages:

- Sphinx
- Read the Docs Sphinx Theme
- nbsphinx
- Sphinx-copybutton
- Jupyter

Install the required packages using pip:

```
pip install sphinx
pip install sphinx-rtd-theme
pip install nbsphinx
pip install sphinx-copybutton
pip install jupyter
```

Run Sphinx in the pymia root directory to create the documentation:

- `sphinx-build -b html ./docs ./docs/_build`
- The documentation is now available under `./docs/_build/index.html`

Note: To build the documentation, it might be required to install pandoc.

In case of the warning *WARNING: LaTeX command 'latex' cannot be run (needed for math display), check the imgmath_latex setting*, set the imgmath_latex setting in the `./docs/conf.py` file.

1.2 Examples

The following examples illustrate the intended use of pymia:

1.2.1 Creation of a dataset

This example shows how to use the `pymia.data` package to create a HDF5 (hierarchical data format version 5) dataset. All examples follow the use case of medical image segmentation of brain tissues, see *Examples* for an introduction into the data. Therefore, we create a dataset with the four subjects and their data: a T1-weighted MR image, a T2-weighted MR image, a label image (ground truth, GT), and a mask image, as well as demographic information age, grade point average (GPA), and gender.

> Tip
>
> This example is available as Jupyter notebook at ./examples/data/creation.ipynb and Python script at ./examples/data/creation.py.

Note

To be able to run this example:

- Get the example data by executing ./examples/example-data/pull_example_data.py.

Import the required modules.

[1]:
```
import enum
import glob
import os
import typing

import SimpleITK as sitk
import numpy as np

import pymia.data as data
import pymia.data.conversion as conv
import pymia.data.definition as defs
import pymia.data.creation as crt
import pymia.data.transformation as tfm
import pymia.data.creation.fileloader as file_load
```

Let us first define an enumeration with the data we will write to the dataset.

[2]:
```
class FileTypes(enum.Enum):
    T1 = 1  # The T1-weighted MR image
    T2 = 2  # The T2-weighted MR image
    GT = 3  # The label (ground truth) image
    MASK = 4  # The foreground mask
    AGE = 5  # The age
    GPA = 6  # The GPA
    GENDER = 7  # The gender
```

Next, we define a subject. Each subject will have two structural MR images (T1w, T2w), one label image (ground truth), a mask, two numericals (age and GPA), and the gender (a character "m" or "w").

[3]:
```
class Subject(data.SubjectFile):

    def __init__(self, subject: str, files: dict):
        super().__init__(subject,
                         images={FileTypes.T1.name: files[FileTypes.T1], FileTypes.T2.
↪name: files[FileTypes.T2]},
                         labels={FileTypes.GT.name: files[FileTypes.GT]},
                         mask={FileTypes.MASK.name: files[FileTypes.MASK]},
                         numerical={FileTypes.AGE.name: files[FileTypes.AGE], FileTypes.
↪GPA.name: files[FileTypes.GPA]},
                         gender={FileTypes.GENDER.name: files[FileTypes.GENDER]})
        self.subject_path = files.get(subject, '')
```

We now collect the subjects, and initialize a `Subject` holding paths to each of the data.

[4]:
```
data_dir = '../example-data'

# get subjects
subject_dirs = [subject_dir for subject_dir in glob.glob(os.path.join(data_dir, '*')) if␣
↪os.path.isdir(subject_dir) and os.path.basename(subject_dir).startswith('Subject')]
sorted(subject_dirs)
```

(continues on next page)

(continued from previous page)

```
# the keys of the data to write to the dataset
keys = [FileTypes.T1, FileTypes.T2, FileTypes.GT, FileTypes.MASK, FileTypes.AGE,␣
↪FileTypes.GPA, FileTypes.GENDER]

subjects = []
# for each subject on file system, initialize a Subject object
for subject_dir in subject_dirs:
    id_ = os.path.basename(subject_dir)

    file_dict = {id_: subject_dir}  # init dict with id_ pointing to the path of the␣
↪subject
    for file_key in keys:
        if file_key == FileTypes.T1:
            file_name = f'{id_}_T1.mha'
        elif file_key == FileTypes.T2:
            file_name = f'{id_}_T2.mha'
        elif file_key == FileTypes.GT:
            file_name = f'{id_}_GT.mha'
        elif file_key == FileTypes.MASK:
            file_name = f'{id_}_MASK.nii.gz'
        elif file_key == FileTypes.AGE or file_key == FileTypes.GPA or file_key ==␣
↪FileTypes.GENDER:
            file_name = f'{id_}_demographic.txt'
        else:
            raise ValueError('Unknown key')

        file_dict[file_key] = os.path.join(subject_dir, file_name)

    subjects.append(Subject(id_, file_dict))
```

Then, we define a `LoadData` class. We load the structural MR images (T1w and T2w) as float and the other images as int. The age, GPA, and gender are loaded from the text file.

[5]:
```
class LoadData(file_load.Load):

    def __call__(self, file_name: str, id_: str, category: str, subject_id: str) -> \
            typing.Tuple[np.ndarray, typing.Union[conv.ImageProperties, None]]:
        if id_ == FileTypes.AGE.name:
            with open(file_name, 'r') as f:
                value = np.asarray([int(f.readline().split(':')[1].strip())])
                return value, None
        if id_ == FileTypes.GPA.name:
            with open(file_name, 'r') as f:
                value = np.asarray([float(f.readlines()[1].split(':')[1].strip())])
                return value, None
        if id_ == FileTypes.GENDER.name:
            with open(file_name, 'r') as f:
                value = np.array(f.readlines()[2].split(':')[1].strip())
                return value, None

        if category == defs.KEY_IMAGES:
```

(continues on next page)

(continued from previous page)

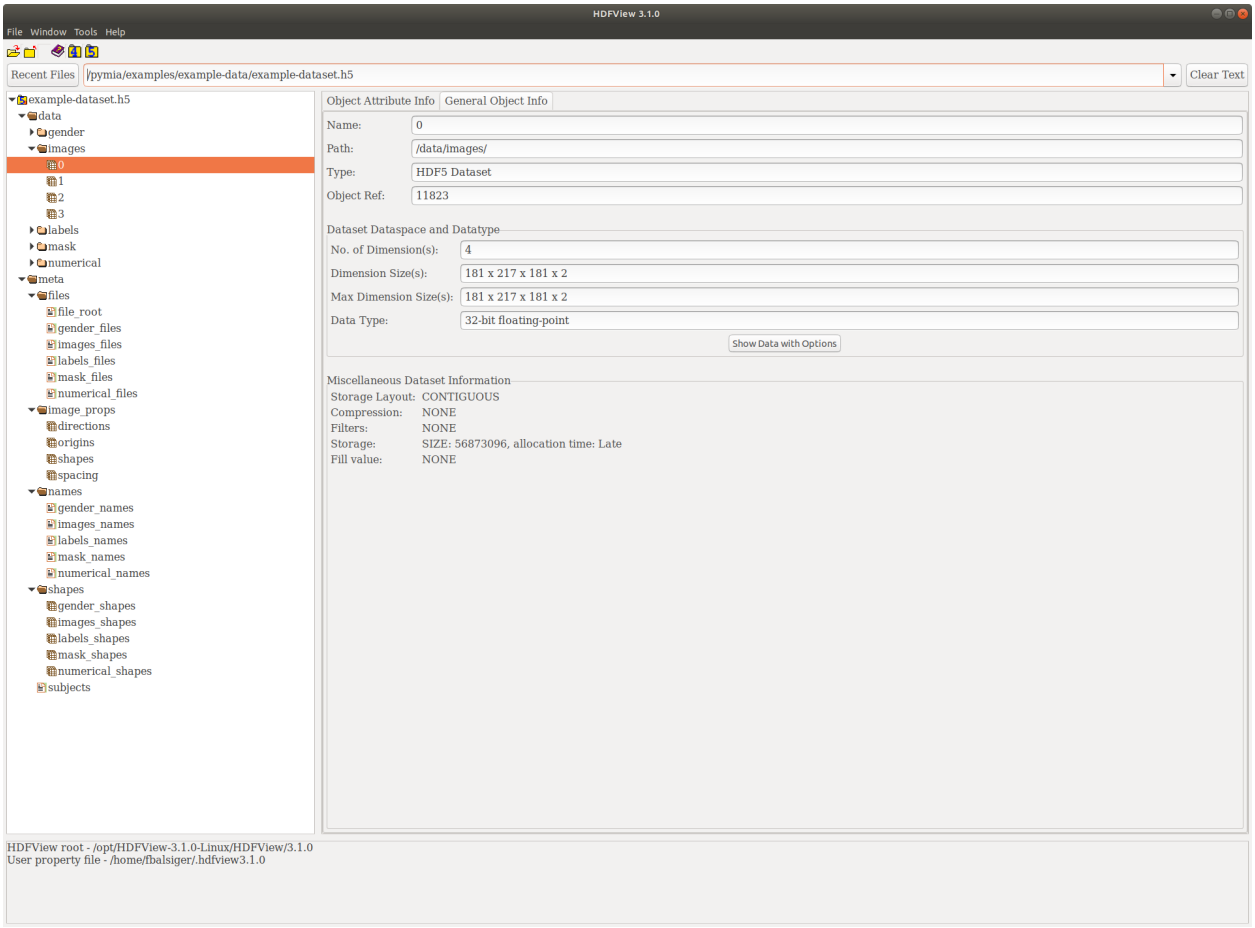
```
            img = sitk.ReadImage(file_name, sitk.sitkFloat32)
        else:
            # this is the ground truth (defs.KEY_LABELS) and mask, which will be loaded␣
↪as unsigned integer
            img = sitk.ReadImage(file_name, sitk.sitkUInt8)

        # return both the image intensities as np.ndarray and the properties of the image
        return sitk.GetArrayFromImage(img), conv.ImageProperties(img)
```

Finally, we can use a writer to create the HDF5 dataset by passing the list of `Subjects` and the `LoadData` to a `Traverser`. For the structural MR images, we also apply an intensity normalization.

[6]:
```
hdf_file = '../example-data/example-dataset.h5'

# remove the "old" dataset if it exists
if os.path.exists(hdf_file):
    os.remove(hdf_file)

with crt.get_writer(hdf_file) as writer:
    # initialize the callbacks that will actually write the data to the dataset file
    callbacks = crt.get_default_callbacks(writer)

    # add a transform to normalize the structural MR images
    transform = tfm.IntensityNormalization(loop_axis=3, entries=(defs.KEY_IMAGES, ))

    # run through the subject files (loads them, applies transformations, and calls the␣
↪callback for writing them)
    traverser = crt.Traverser()
    traverser.traverse(subjects, callback=callbacks, load=LoadData(),␣
↪transform=transform)
```

```
start dataset creation
[1/4] Subject_1
[2/4] Subject_2
[3/4] Subject_3
[4/4] Subject_4
dataset creation finished
```

This should now have created a `example-dataset.h5` in the directory `./examples/example-data`. By using a HDF5 viewer like HDF Compass or HDFView, we can inspect the dataset. It should look similar to the figure below.



1.2.2 Data extraction and assembly

This example shows how to use the *pymia.data* package to extract chunks of data from the dataset and to assemble the chunks to feed a deep neural network. It also shows how the predicted chunks are assembled back to full-images predictions.

The extraction-assemble principle is essential for large three-dimensional images that do not fit entirely in the GPU memory and thus require some kind of patch-based approach.

For simplicity reasons we use slice-wise extraction in this example, meaning that the two-dimensional slices are extracted from the three-dimensional image. Further, the example uses PyTorch as a deep learning (DL) framework.

At the end of this example you find examples for the following additional use cases:

- TensorFlow adaptions
- Extracting 3-D patches
- Extracting from a metadata dataset

Tip: This example is available as Jupyter notebook at ./examples/data/extraction_assembly.ipynb and Python scripts for PyTorch and TensorFlow at at ./examples/data/extraction_assembly.py and ./examples/data/extraction_assembly_tensorflow.py, respectively.

The extraction of 3-D patches is available as Python script at ./examples/data/extraction_assembly_3dpatch.py.

Note: To be able to run this example:

- Get the example data by executing ./examples/example-data/pull_example_data.py.

Code walkthrough

[0] Import the required modules.

```
import pymia.data.assembler as assm
import pymia.data.transformation as tfm
import pymia.data.definition as defs
import pymia.data.extraction as extr
import pymia.data.backends.pytorch as pymia_torch
```

[1] First, we create the the access to the .h5 dataset by defining: (i) the indexing strategy (*indexing_strategy*) that defines the chunks of data to be retrieved, (ii) the information to be extracted (*extractor*), and (iii) the transformation (*transform*) to be applied after extraction.

The permutation transform is required since the channels (here _T1_, _T2_) are stored in the last dimension in the .h5 dataset but PyTorch requires channel-first format.

```
hdf_file = '../example-data/example-dataset.h5'

# Data extractor for extracting the "images" entries
extractor = extr.DataExtractor(categories=(defs.KEY_IMAGES,))
# Permutation transform to go from HWC to CHW.
transform = tfm.Permute(permutation=(2, 0, 1), entries=(defs.KEY_IMAGES,))
# Indexing defining a slice-wise extraction of the data
indexing_strategy = extr.SliceIndexing()

dataset = extr.PymiaDatasource(hdf_file, indexing_strategy, extractor, transform)
```

[2] Next, we define an assembler that will puts the data/image chunks back together after prediction of the input chunks. This is required to perform a evaluation on entire subjects, and any further processing such as saving the predictions.

Also, we define extractors that we will use to extract information required after prediction. This information not need to be chunked (/indexed/sliced) and not need to interact with the DL framework. Thus, it can be extracted directly form the dataset.

```
assembler = assm.SubjectAssembler(dataset)

direct_extractor = extr.ComposeExtractor([
    extr.ImagePropertiesExtractor(),  # Extraction of image properties (origin, spacing,␣
↪etc.) for storage
    extr.DataExtractor(categories=(defs.KEY_LABELS,))  # Extraction of "labels" entries␣
↪for evaluation
])
```

[3] The batch generation and and the neural network architecture are framework dependent. Basically, all we have to do is to wrap our dataset as PyTorch dataset, to build a PyTorch data loader, and to create/load a network.

```
import torch
import torch.nn as nn
```

(continues on next page)

(continued from previous page)

```
import torch.utils.data as torch_data

# Wrap the pymia datasource
pytorch_dataset = pymia_torch.PytorchDatasetAdapter(dataset)
loader = torch_data.dataloader.DataLoader(pytorch_dataset, batch_size=2, shuffle=False)

# Dummy network representing a placeholder for a trained network
dummy_network = nn.Sequential(
    nn.Conv2d(in_channels=2, out_channels=8, kernel_size=3, padding=1),
    nn.Conv2d(in_channels=8, out_channels=1, kernel_size=3, padding=1),
    nn.Sigmoid()
).eval()
torch.set_grad_enabled(False)  # no gradients needed for testing

nb_batches = len(loader)
```

[4] We are now ready to loop over batches of data chunks. After the usual prediction of the network, the predicted data is provided to the assembler, which takes care of putting chunks back together. Once some subjects are assembled (*subjects_ready*) we extract the data required for evaluation and storing.

```
for i, batch in enumerate(loader):

    # Get data from batch and predict
    x, sample_indices = batch[defs.KEY_IMAGES], batch[defs.KEY_SAMPLE_INDEX]
    prediction = dummy_network(x)

    # translate the prediction to numpy and back to (B)HWC (channel last)
    numpy_prediction = prediction.numpy().transpose((0, 2, 3, 1))

    # add the batch prediction to the assembler
    is_last = i == nb_batches - 1
    assembler.add_batch(numpy_prediction, sample_indices.numpy(), is_last)

    # Process the subjects/images that are fully assembled
    for subject_index in assembler.subjects_ready:
        subject_prediction = assembler.get_assembled_subject(subject_index)

        # Extract the target and image properties via direct extract
        direct_sample = dataset.direct_extract(direct_extractor, subject_index)
        target, image_properties = direct_sample[defs.KEY_LABELS], direct_sample[defs.
↪KEY_PROPERTIES]

        # # Do whatever you desire...
        # do_eval()
        # do_save()
```

TensorFlow adaptions

Only the *PymiaDatasource* wrapping has to be changed to use the pymia data handling together with TensorFlow instead of PyTorch. This change, however, implies other framework related changes.

[0] Add Tensorflow specific imports.

```
import tensorflow as tf
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
import pymia.data.backends.tensorflow as pymia_tf
```

[1] Wrap the *PymiaDatasource* (*dataset*) and use Tensorflow specific data handling.

```
gen_fn = pymia_tf.get_tf_generator(dataset)
tf_dataset = tf.data.Dataset.from_generator(generator=gen_fn,
                                            output_types={defs.KEY_IMAGES: tf.float32,
                                                          defs.KEY_SAMPLE_INDEX: tf.
↪int64})
loader = tf_dataset.batch(2)

dummy_network = keras.Sequential([
    layers.Conv2D(8, kernel_size=3, padding='same'),
    layers.Conv2D(2, kernel_size=3, padding='same', activation='sigmoid')]
)
nb_batches = len(dataset) // 2
```

[2] As opposed to PyTorch, Tensorflow uses the channel-last (BWHC) configuration. Thus, the permutations are no longer required

```
# The lines following lines of the initial code ...
transform = tfm.Permute(permutation=(2, 0, 1), entries=(defs.KEY_IMAGES,))
numpy_prediction = prediction.numpy().transpose((0, 2, 3, 1))
# ... become
transform = None
numpy_prediction = prediction.numpy()
```

Extracting 3-D patches

To extract 3-D patches instead of slices requires only a few changes.

[0] Modifications on the indexing are typically due to a network change. Here, we still use a dummy network, but this time it consists of 3-D valid convolutions (instead of 2-D same convolutions).

```
dummy_network = nn.Sequential(
    nn.Conv3d(in_channels=2, out_channels=8, kernel_size=3, padding=0),
    nn.Conv3d(in_channels=8, out_channels=1, kernel_size=3, padding=0),
    nn.Sigmoid()
)
```

[1] By knowing the architecture of the new network, we can modify the pymia related extraction. Note that the network input shape is by 4 voxels larger then the output shape (valid convolutions). A input patch size of 36x36x36 extracted and the output patch size will be 32x32x32.

```
# Adapted permutation due to the additional dimension
transform = tfm.Permute(permutation=(3, 0, 1, 2), entries=(defs.KEY_IMAGES,))

# Use a pad extractor to compensate input-output shape difference of the network. Actual␣
↪image information is padded.
extractor = extr.PadDataExtractor((2, 2, 2), extr.DataExtractor(categories=(defs.KEY_
↪IMAGES,)))
```

[2] The modifications from 2-D to 3-D also affects the permutations.

```
transform = tfm.Permute(permutation=(3, 0, 1, 2), entries=(defs.KEY_IMAGES,))
numpy_prediction = prediction.numpy().transpose((0, 2, 3, 4, 1))
```

Extracting from a metadata dataset

A metadata dataset only contains metadata but not image (or other) data. Metadata datasets might be used when the amount of data is large. They avoid storing a copy of the data in the dataset and access the raw data directly via the file links.

Extracting data from a metadata dataset is very simple and only requires to employ the corresponding *Extractor*.

```
# The following line of the initial code ...
extractor = extr.DataExtractor(categories=(defs.KEY_IMAGES,))
# ... becomes
extractor = extr.FilesystemDataExtractor(categories=(defs.KEY_IMAGES,))
```

1.2.3 Evaluation of results

This example shows how to use the `pymia.evaluation` package to evaluate predicted segmentations against reference ground truths. Common metrics in medical image segmentation are the Dice coefficient, an overlap-based metric, and the Hausdorff distance, a distance-based metric. Further, we also evaluate the volume similarity, a metric that does not consider the spatial overlap. The evaluation results are logged to the console and saved to a CSV file. Further, statistics (mean and standard deviation) are calculated over all evaluated segmentations, which are again logged to the console and saved to a CSV file. The CSV files could be loaded into any statistical software for further analysis and visualization.

> Tip
>
> This example is available as Jupyter notebook at ./examples/evaluation/basic.ipynb and Python script at ./examples/evaluation/basic.py.

Note

To be able to run this example:

- Get the example data by executing ./examples/example-data/pull_example_data.py.
- Install pandas (`pip install pandas`).

Import the required modules.

[1]:
```
import glob
import os

import numpy as np
import pymia.evaluation.metric as metric
import pymia.evaluation.evaluator as eval_
import pymia.evaluation.writer as writer
import SimpleITK as sitk
```

Define the paths to the data and the result CSV files.

[2]:
```
data_dir = '../example-data'

result_file = '../example-data/results.csv'
result_summary_file = '../example-data/results_summary.csv'
```

Let us create a list with the three metrics: the Dice coefficient, the Hausdorff distance, and the volume similarity. Note that we are interested in the outlier-robust 95th Hausdorff distance, and, therefore, pass the percentile as argument and adapt the metric's name.

[3]:
```
metrics = [metric.DiceCoefficient(), metric.HausdorffDistance(percentile=95, metric=
↪'HDRFDST95'), metric.VolumeSimilarity()]
```

Now, we need to define the labels we want to evaluate. In the provided example data, we have five labels for different brain structures. Here, we are only interested in three of them: white matter, grey matter, and the thalamus.

[4]:
```
labels = {1: 'WHITEMATTER',
          2: 'GREYMATTER',
          5: 'THALAMUS'
          }
```

Finally, we can initialize an evaluator with the metrics and labels.

[5]:
```
evaluator = eval_.SegmentationEvaluator(metrics, labels)
```

We can now loop over the subjects of the example data. We will load the ground truth image as reference. An artificial segmentation (prediction) is created by eroding the ground truth. Both images, and the subject identifier are passed to the evaluator.

[6]:
```
# get subjects to evaluate
subject_dirs = [subject for subject in glob.glob(os.path.join(data_dir, '*')) if os.path.
↪isdir(subject) and os.path.basename(subject).startswith('Subject')]

for subject_dir in subject_dirs:
    subject_id = os.path.basename(subject_dir)
    print(f'Evaluating {subject_id}...')

    # load ground truth image and create artificial prediction by erosion
    ground_truth = sitk.ReadImage(os.path.join(subject_dir, f'{subject_id}_GT.mha'))
    prediction = ground_truth
    for label_val in labels.keys():
        # erode each label we are going to evaluate
        prediction = sitk.BinaryErode(prediction, 1, sitk.sitkBall, 0, label_val)
```

(continues on next page)

(continued from previous page)

```
    # evaluate the "prediction" against the ground truth
    evaluator.evaluate(prediction, ground_truth, subject_id)
```

```
Evaluating Subject_1...
Evaluating Subject_2...
Evaluating Subject_3...
Evaluating Subject_4...
```

After we evaluated all subjects, we can use a CSV writer to write the evaluation results to a CSV file.

[7]:
```
writer.CSVWriter(result_file).write(evaluator.results)
```

Further, we can use a console writer to display the results in the console.

[8]:
```
print('\nSubject-wise results...')
writer.ConsoleWriter().write(evaluator.results)
```

```

Subject-wise results...
SUBJECT    LABEL        DICE   HDRFDST95  VOLSMTY
Subject_1  GREYMATTER   0.313  9.165      0.313
Subject_1  THALAMUS     0.752  2.000      0.752
Subject_1  WHITEMATTER  0.642  6.708      0.642
Subject_2  GREYMATTER   0.298  10.863     0.298
Subject_2  THALAMUS     0.768  2.000      0.768
Subject_2  WHITEMATTER  0.654  6.000      0.654
Subject_3  GREYMATTER   0.287  8.718      0.287
Subject_3  THALAMUS     0.761  2.000      0.761
Subject_3  WHITEMATTER  0.641  6.164      0.641
Subject_4  GREYMATTER   0.259  8.660      0.259
Subject_4  THALAMUS     0.781  2.000      0.781
Subject_4  WHITEMATTER  0.649  6.000      0.649
```

We can also report statistics such as the mean and standard deviation among all subjects using dedicated statistics writers. Note that you can pass any functions that take a list of floats and return a scalar value to the writers. Again, we will write a CSV file and display the results in the console.

[9]:
```
functions = {'MEAN': np.mean, 'STD': np.std}
writer.CSVStatisticsWriter(result_summary_file, functions=functions).write(evaluator.
↪results)
print('\nAggregated statistic results...')
writer.ConsoleStatisticsWriter(functions=functions).write(evaluator.results)
```

```

Aggregated statistic results...
LABEL        METRIC     STATISTIC  VALUE
GREYMATTER   DICE       MEAN       0.289
GREYMATTER   DICE       STD        0.020
GREYMATTER   HDRFDST95  MEAN       9.351
GREYMATTER   HDRFDST95  STD        0.894
GREYMATTER   VOLSMTY    MEAN       0.289
GREYMATTER   VOLSMTY    STD        0.020
THALAMUS     DICE       MEAN       0.766
THALAMUS     DICE       STD        0.010
```

(continues on next page)

(continued from previous page)

```
THALAMUS     HDRFDST95  MEAN       2.000
THALAMUS     HDRFDST95  STD        0.000
THALAMUS     VOLSMTY    MEAN       0.766
THALAMUS     VOLSMTY    STD        0.010
WHITEMATTER  DICE       MEAN       0.647
WHITEMATTER  DICE       STD        0.005
WHITEMATTER  HDRFDST95  MEAN       6.218
WHITEMATTER  HDRFDST95  STD        0.291
WHITEMATTER  VOLSMTY    MEAN       0.647
WHITEMATTER  VOLSMTY    STD        0.005
```

Finally, we clear the results in the evaluator such that the evaluator is ready for the next evaluation.

[10]:
```
evaluator.clear()
```

Now, let us have a look at the saved result CSV file.

[11]:
```
import pandas as pd

pd.read_csv(result_file, sep=';')
```

[11]:

	SUBJECT	LABEL	DICE	HDRFDST95	VOLSMTY
0	Subject_1	GREYMATTER	0.313373	9.165151	0.313373
1	Subject_1	THALAMUS	0.752252	2.000000	0.752252
2	Subject_1	WHITEMATTER	0.642021	6.708204	0.642021
3	Subject_2	GREYMATTER	0.298358	10.862780	0.298358
4	Subject_2	THALAMUS	0.768488	2.000000	0.768488
5	Subject_2	WHITEMATTER	0.654239	6.000000	0.654239
6	Subject_3	GREYMATTER	0.287460	8.717798	0.287460
7	Subject_3	THALAMUS	0.760978	2.000000	0.760978
8	Subject_3	WHITEMATTER	0.641251	6.164414	0.641251
9	Subject_4	GREYMATTER	0.258504	8.660254	0.258504
10	Subject_4	THALAMUS	0.780754	2.000000	0.780754
11	Subject_4	WHITEMATTER	0.649203	6.000000	0.649203

And also at the saved statistics CSV file.

[12]:
```
pd.read_csv(result_summary_file, sep=';')
```

[12]:

	LABEL	METRIC	STATISTIC	VALUE
0	GREYMATTER	DICE	MEAN	0.289424
1	GREYMATTER	DICE	STD	0.020083
2	GREYMATTER	HDRFDST95	MEAN	9.351496
3	GREYMATTER	HDRFDST95	STD	0.894161
4	GREYMATTER	VOLSMTY	MEAN	0.289424
5	GREYMATTER	VOLSMTY	STD	0.020083
6	THALAMUS	DICE	MEAN	0.765618
7	THALAMUS	DICE	STD	0.010458
8	THALAMUS	HDRFDST95	MEAN	2.000000
9	THALAMUS	HDRFDST95	STD	0.000000
10	THALAMUS	VOLSMTY	MEAN	0.765618
11	THALAMUS	VOLSMTY	STD	0.010458
12	WHITEMATTER	DICE	MEAN	0.646678

(continues on next page)

(continued from previous page)

```
13  WHITEMATTER       DICE       STD  0.005355
14  WHITEMATTER  HDRFDST95      MEAN  6.218154
15  WHITEMATTER  HDRFDST95       STD  0.290783
16  WHITEMATTER    VOLSMTY      MEAN  0.646678
17  WHITEMATTER    VOLSMTY       STD  0.005355
```

1.2.4 Logging the training progress

This example shows how to use the `pymia.evaluation` package to log the performance of a neural network during training. The TensorBoard is commonly used to visualize the training in deep learning. We will log the Dice coefficient of predicted segmentations calculated against a reference ground truth to the TensorBoard to visualize the performance of a neural network during the training.

This example uses PyTorch. At the end of it, you can find the required modifications for TensorFlow.

> Tip
>
> This example is available as Jupyter notebook at ./examples/evaluation/logging.ipynb and Python scripts for PyTorch and TensorFlow at ./examples/evaluation/logging_torch.py and ./examples/evaluation/logging_tensorflow.py, respectively.

Note

To be able to run this example:

- Get the example data by executing ./examples/example-data/pull_example_data.py.
- Install torch (`pip install torch`).
- Install tensorboard (`pip install tensorboard`).

Further, it might be good to be familiar with *Data extraction and assembling* and *Evaluation of results*.

Import the required modules.

[1]:
```
import os

import numpy as np
import pymia.data.assembler as assm
import pymia.data.backends.pytorch as pymia_torch
import pymia.data.definition as defs
import pymia.data.extraction as extr
import pymia.data.transformation as tfm
import pymia.evaluation.metric as metric
import pymia.evaluation.evaluator as eval_
import pymia.evaluation.writer as writer
import torch
import torch.nn as nn
import torch.utils.data as torch_data
import torch.utils.tensorboard as tensorboard
```

Let us create a list with the metric to log, the Dice coefficient.

[2]:
```
metrics = [metric.DiceCoefficient()]
```

Now, we need to define the labels we want to log during the training. In the provided example data, we have five labels for different brain structures. Here, we are only interested in three of them: white matter, grey matter, and the thalamus.

[3]:
```
labels = {1: 'WHITEMATTER',
          2: 'GREYMATTER',
          5: 'THALAMUS'
          }
```

Using the metrics and labels, we can initialize an evaluator.

[4]:
```
evaluator = eval_.SegmentationEvaluator(metrics, labels)
```

The evaluator will return results for all subjects in the dataset. However, we would like to log only statistics like the mean and the standard deviation of the metrics among all subjects. Therefore, we initialize a statistics aggregator.

[5]:
```
functions = {'MEAN': np.mean, 'STD': np.std}
statistics_aggregator = writer.StatisticsAggregator(functions=functions)
```

PyTorch provides a module to log to the TensorBoard, which we will use.

[6]:
```
log_dir = '../example-data/log'
tb = tensorboard.SummaryWriter(os.path.join(log_dir, 'logging-example'))
```

We now initialize the data handling, please refer to the above mentioned example to understand what is going on.

[7]:
```
hdf_file = '../example-data/example-dataset.h5'
transform = tfm.Permute(permutation=(2, 0, 1), entries=(defs.KEY_IMAGES,))
dataset = extr.PymiaDatasource(hdf_file, extr.SliceIndexing(), extr.
↪DataExtractor(categories=(defs.KEY_IMAGES,)), transform)
pytorch_dataset = pymia_torch.PytorchDatasetAdapter(dataset)
loader = torch_data.dataloader.DataLoader(pytorch_dataset, batch_size=100, shuffle=False)

assembler = assm.SubjectAssembler(dataset)
direct_extractor = extr.ComposeExtractor([
    extr.SubjectExtractor(),  # extraction of the subject name for evaluation
    extr.ImagePropertiesExtractor(),  # extraction of image properties (origin, spacing,␣
↪etc.) for evaluation in physical space
    extr.DataExtractor(categories=(defs.KEY_LABELS,))  # extraction of "labels" entries␣
↪for evaluation
])
```

Let's now define a dummy network, which will actually just return a random prediction.

[8]:
```
class DummyNetwork(nn.Module):

    def forward(self, x):
        return torch.randint(0, 5, (x.size(0), 1, *x.size()[2:]))

dummy_network = DummyNetwork()
torch.manual_seed(0)  # set seed for reproducibility
```

[8]:
```
<torch._C.Generator at 0x7f09f951adb0>
```

We can now start the training loop. We will loop over the samples in our dataset, feed them to the "neural network", and assemble them to back to entire volumetric predictions. As soon as a prediction is fully assembled, it will be evaluated against its reference. We do this evaluation in the physical space, as the spacing might be important for metrics like the Hausdorff distance (distances in mm rather than voxels). At the end of each epoch, we can calculate the mean and standard deviation of the metrics among all subjects in the dataset, and log them to the TensorBoard. Note that this example is just for illustration because usually you would want to log the performance on the validation set.

[9]:
```
nb_batches = len(loader)

epochs = 10
for epoch in range(epochs):
    print(f'Epoch {epoch + 1}/{epochs}')
    for i, batch in enumerate(loader):
        # get the data from batch and predict
        x, sample_indices = batch[defs.KEY_IMAGES], batch[defs.KEY_SAMPLE_INDEX]
        prediction = dummy_network(x)

        # translate the prediction to numpy and back to (B)HWC (channel last)
        numpy_prediction = prediction.numpy().transpose((0, 2, 3, 1))

        # add the batch prediction to the assembler
        is_last = i == nb_batches - 1
        assembler.add_batch(numpy_prediction, sample_indices.numpy(), is_last)

        # process the subjects/images that are fully assembled
        for subject_index in assembler.subjects_ready:
            subject_prediction = assembler.get_assembled_subject(subject_index)

            # extract the target and image properties via direct extract
            direct_sample = dataset.direct_extract(direct_extractor, subject_index)
            reference, image_properties = direct_sample[defs.KEY_LABELS], direct_
↪sample[defs.KEY_PROPERTIES]

            # evaluate the prediction against the reference
            evaluator.evaluate(subject_prediction[..., 0], reference[..., 0], direct_
↪sample[defs.KEY_SUBJECT])

    # calculate mean and standard deviation of each metric
    results = statistics_aggregator.calculate(evaluator.results)
    # log to TensorBoard into category train
    for result in results:
        tb.add_scalar(f'train/{result.metric}-{result.id_}', result.value, epoch)

    # clear results such that the evaluator is ready for the next evaluation
    evaluator.clear()
```

```
Epoch 1/10
Epoch 2/10
Epoch 3/10
Epoch 4/10
Epoch 5/10
Epoch 6/10
Epoch 7/10
Epoch 8/10
```

(continues on next page)

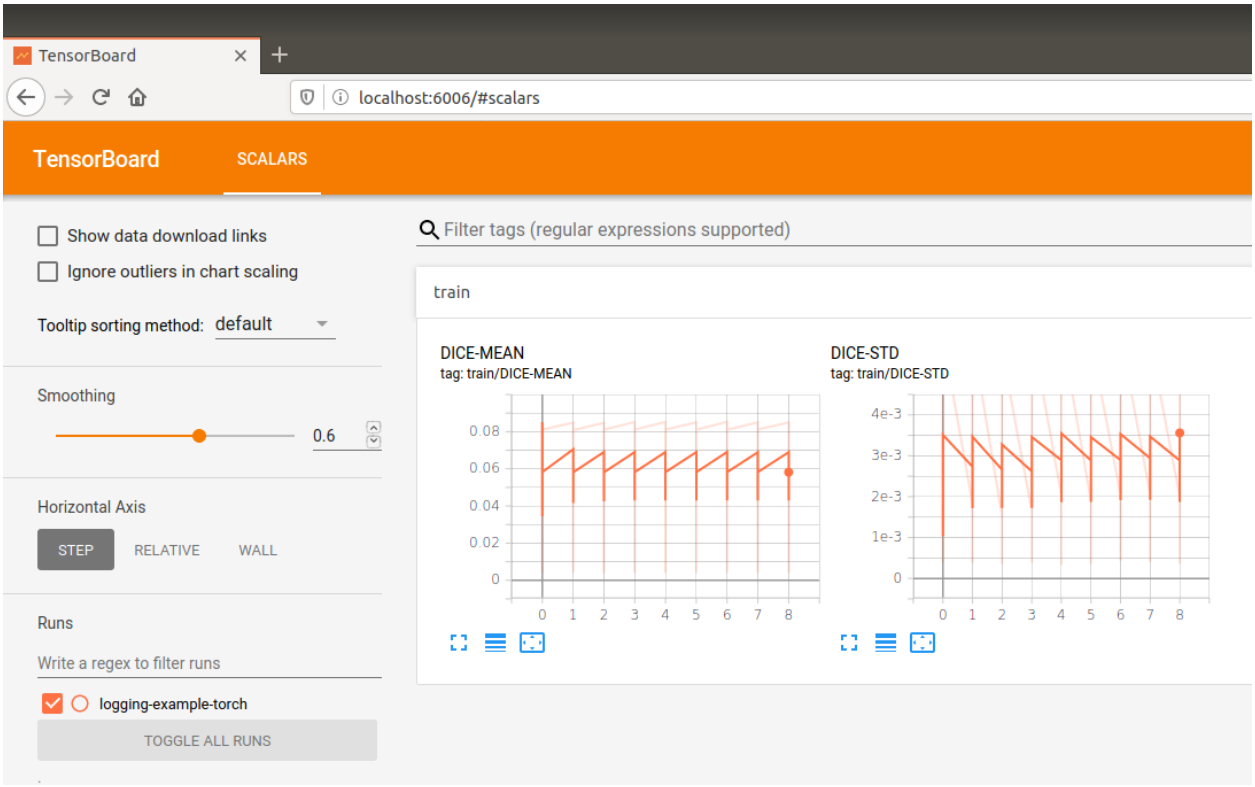
(continued from previous page)

```
Epoch 9/10
Epoch 10/10
```

You can now start the TensorBoard and point the location to the log directory:

```
tensorboard --logdir=<path_to_pymia>/examples/example-data/log
```

Open a browser and type localhost:6006 to see the logged training progress. It should look similar to the figure below (the data does not make a lot of sense as we create random predictions).



TensorFlow adaptions

For the presented logging to work with the TensorFlow framework, only minor modifications are required: (1) Modifications of the imports, (2) framework-specific TensorBoard logging, and (3) framework-specific data handling.

```
# 1)
import tensorflow as tf
import pymia.data.backends.tensorflow as pymia_tf

# 2)
tb = tf.summary.create_file_writer(os.path.join(log_dir, 'logging-example'))

for result in results:
    with tb.as_default():
        tf.summary.scalar(f'train/{result.metric}-{result.id_}', result.value, epoch)
```

(continues on next page)

(continued from previous page)

```
# 3)
gen_fn = pymia_tf.get_tf_generator(dataset)
tf_dataset = tf.data.Dataset.from_generator(generator=gen_fn,
                                            output_types={defs.KEY_IMAGES: tf.float32,
                                                          defs.KEY_SAMPLE_INDEX: tf.
↪int64})
loader = tf_dataset.batch(100)

class DummyNetwork(tf.keras.Model):

    def call(self, inputs):
        return tf.random.uniform((*inputs.shape[:-1], 1), 0, 6, dtype=tf.int32)

dummy_network = DummyNetwork()
tf.random.set_seed(0)  # set seed for reproducibility

# no permutation transform needed. Thus the lines
transform = tfm.Permute(permutation=(2, 0, 1), entries=(defs.KEY_IMAGES,))
numpy_prediction = prediction.numpy().transpose((0, 2, 3, 1))
# become
transform = None
numpy_prediction = prediction.numpy()
```

1.2.5 Filter pipelines

This example shows how to use the `pymia.filtering` package to set up image filter pipeline and apply it to an image. The pipeline consists of a gradient anisotropic diffusion filter followed by a histogram matching. This pipeline will be applied to a T1-weighted MR image and a T2-weighted MR image will be used as a reference for the histogram matching.

> Tip
>
> This example is available as Jupyter notebook at ./examples/filtering/basic.ipynb and Python script at ./examples/filtering/basic.py.

Note

To be able to run this example:

- Get the example data by executing ./examples/example-data/pull_example_data.py.
- Install matplotlib (`pip install matplotlib`).

Import the required modules.

[1]:
```
import glob
import os

import matplotlib.pyplot as plt
import pymia.filtering.filter as flt
```

(continues on next page)

(continued from previous page)

```
import pymia.filtering.preprocessing as prep
import SimpleITK as sitk
```

Define the path to the data.

[2]:
```
data_dir = '../example-data'
```

Let us create a list with the two filters, a gradient anisotropic diffusion filter followed by a histogram matching.

[3]:
```
filters = [
    prep.GradientAnisotropicDiffusion(time_step=0.0625),
    prep.HistogramMatcher()
]

histogram_matching_filter_idx = 1  # we need the index later to update the␣
↪HistogramMatcher's parameters
```

Now, we can initialize the filter pipeline.

[4]:
```
pipeline = flt.FilterPipeline(filters)
```

We can now loop over the subjects of the example data. We will both load the T1-weighted and T2-weighted MR images and execute the pipeline on the T1-weighted MR image. Note that for each subject, we update the parameters for the histogram matching filter to be the corresponding T2-weighted image.

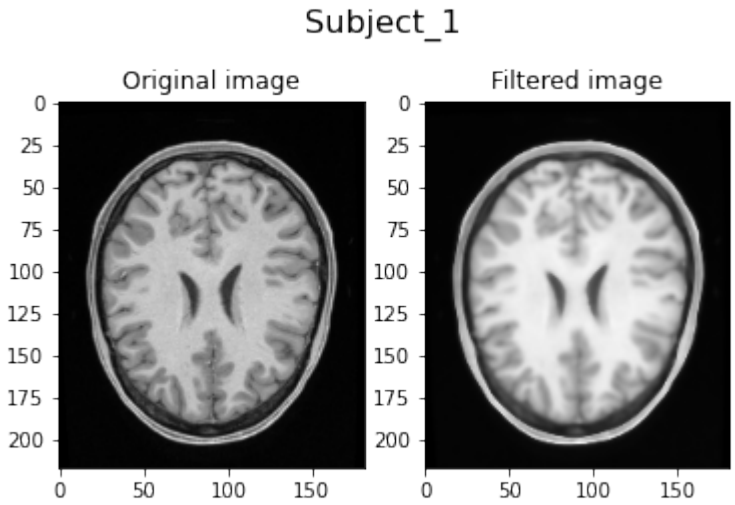
[5]:
```
# get subjects to evaluate
subject_dirs = [subject for subject in glob.glob(os.path.join(data_dir, '*')) if os.path.
↪isdir(subject) and os.path.basename(subject).startswith('Subject')]

for subject_dir in subject_dirs:
    subject_id = os.path.basename(subject_dir)
    print(f'Filtering {subject_id}...')

    # load the T1- and T2-weighted MR images
    t1_image = sitk.ReadImage(os.path.join(subject_dir, f'{subject_id}_T1.mha'))
    t2_image = sitk.ReadImage(os.path.join(subject_dir, f'{subject_id}_T2.mha'))

    # set the T2-weighted MR image as reference for the histogram matching
    pipeline.set_param(prep.HistogramMatcherParams(t2_image), histogram_matching_filter_
↪idx)

    # execute filtering pipeline on the T1-weighted image
    filtered_t1_image = pipeline.execute(t1_image)

    # plot filtering result
    slice_no_for_plot = t1_image.GetSize()[2] // 2
    fig, axs = plt.subplots(1, 2)
    axs[0].imshow(sitk.GetArrayFromImage(t1_image[:, :, slice_no_for_plot]), cmap='gray')
    axs[0].set_title('Original image')
    axs[1].imshow(sitk.GetArrayFromImage(filtered_t1_image[:, :, slice_no_for_plot]),␣
↪cmap='gray')
    axs[1].set_title('Filtered image')
```

(continues on next page)

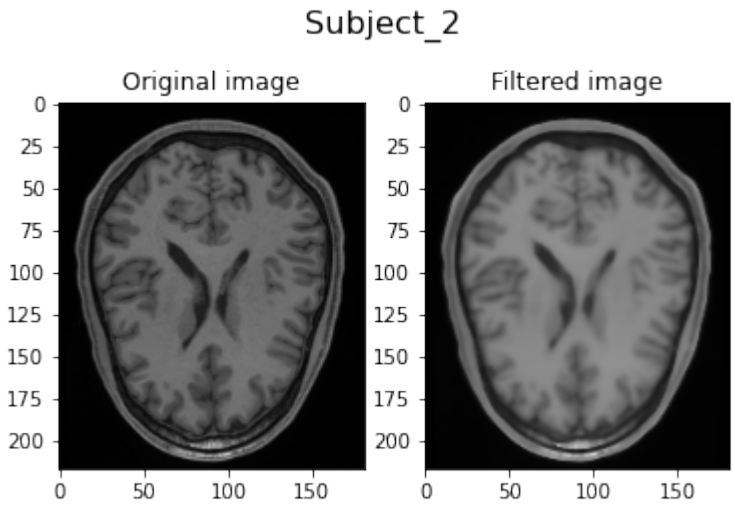
(continued from previous page)

```
    fig.suptitle(f'{subject_id}', fontsize=16)
    plt.show()
```

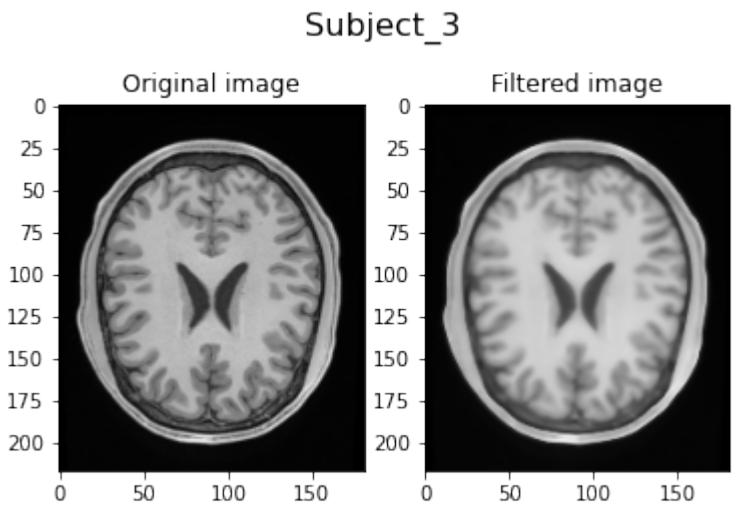
```
Filtering Subject_1...
```



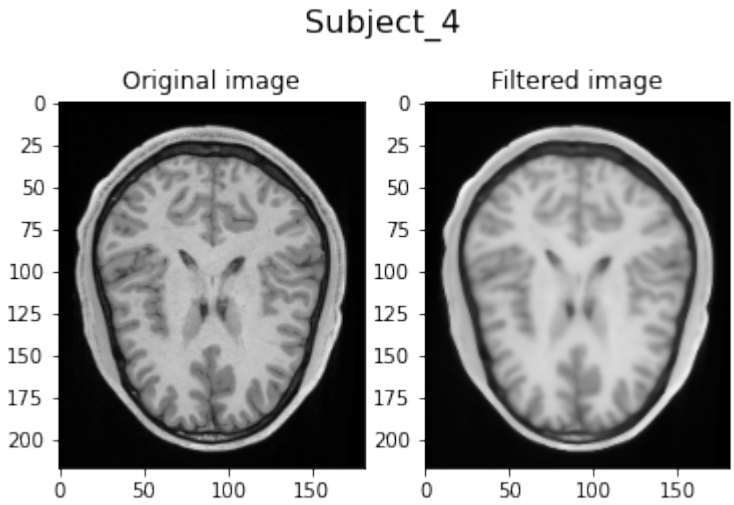

```
Filtering Subject_2...
```




```
Filtering Subject_3...
```




```
Filtering Subject_4...
```



Visually, we can clearly see the smoothing of the filtered image due to the anisotrophic filtering. Also, the image intensities are brighter due to the histogram matching.

1.2.6 Augmentation

This example shows how to apply data augmentation in conjunction with the `pymia.data` package. Besides transformations from the `pymia.data.augmentation` module, transformations from the Python packages batchgenerators and TorchIO are integrated.

> Tip
>
> This example is available as Jupyter notebook at ./examples/augmentation/basic.ipynb and Python script at ./examples/augmentation/basic.py.

Note

To be able to run this example:

- Get the example data by executing ./examples/example-data/pull_example_data.py.
- Install matplotlib (`pip install matplotlib`).
- Install matplotlib (`pip install batchgenerators`).
- Install matplotlib (`pip install torchio`).

Import the required modules.

[1]:
```
import batchgenerators.transforms as bg_tfm
import matplotlib.pyplot as plt
import numpy as np
import torchio as tio

import pymia.data.transformation as tfm
import pymia.data.augmentation as augm
import pymia.data.definition as defs
import pymia.data.extraction as extr
```

```
If you use TorchIO for your research, please cite the following paper:
Pérez-García et al., TorchIO: a Python library for efficient loading,
preprocessing, augmentation and patch-based sampling of medical images
in deep learning. Credits instructions: https://torchio.readthedocs.io/#credits
```

We create the the access to the .h5 dataset by defining: (i) the indexing strategy (`indexing_strategy`) that defines the chunks of data to be retrieved, and (ii) the information to be extracted (`extractor`).

[2]:
```
hdf_file = '../example-data/example-dataset.h5'
indexing_strategy = extr.SliceIndexing()
extractor = extr.DataExtractor(categories=(defs.KEY_IMAGES, defs.KEY_LABELS))
dataset = extr.PymiaDatasource(hdf_file, indexing_strategy, extractor)
```

For reproducibility, set the seed and define a sample index for plotting.

[3]:
```
seed = 1
np.random.seed(seed)
sample_idx = 55
```

We can now define the transformations to apply. For reference, we first do not apply any data augmentation.

[4]:
```
transforms_augmentation = []
transforms_before_augmentation = [tfm.Permute(permutation=(2, 0, 1)), ]  # to have the␣
↪channel-dimension first
transforms_after_augmentation = [tfm.Squeeze(entries=(defs.KEY_LABELS,)), ]  # get rid␣
↪of the channel-dimension for the labels
train_transforms = tfm.ComposeTransform(transforms_before_augmentation + transforms_
↪augmentation + transforms_after_augmentation)
dataset.set_transform(train_transforms)
sample = dataset[sample_idx]
```

pymia augmentation

Let's us now use pymia to apply a random 90° rotation and a random mirroring.

[5]:
```
transforms_augmentation = [augm.RandomRotation90(axes=(-2, -1)), augm.RandomMirror()]
train_transforms = tfm.ComposeTransform(
    transforms_before_augmentation + transforms_augmentation + transforms_after_
↪augmentation)
dataset.set_transform(train_transforms)
sample_pymia = dataset[sample_idx]
```

```
/home/fbalsiger/PycharmProjects/pymia/pymia/data/augmentation.py:231: RuntimeWarning:␣
↪entry "images" has unequal in-plane dimensions (217, 181). Random 90 degree rotation␣
↪might produce undesired results. Verify the output!
  warnings.warn(f'entry "{entry}" has unequal in-plane dimensions ({sample[entry].
↪shape[self.axes[0]]}, '
/home/fbalsiger/PycharmProjects/pymia/pymia/data/augmentation.py:231: RuntimeWarning:␣
↪entry "labels" has unequal in-plane dimensions (217, 181). Random 90 degree rotation␣
↪might produce undesired results. Verify the output!
  warnings.warn(f'entry "{entry}" has unequal in-plane dimensions ({sample[entry].
↪shape[self.axes[0]]}, '
```

batchgenerators augmentation

Let's us now use batchgenerators to apply a random 90° rotation and a random mirroring. To use batchgenerators, we create wrapper classes for simple integration into pymia.

[6]:
```
class BatchgeneratorsTransform(tfm.Transform):
    """Example wrapper for `batchgenerators <https://github.com/MIC-DKFZ/batchgenerators>
↪`_ transformations."""

    def __init__(self, transforms, entries=(defs.KEY_IMAGES, defs.KEY_LABELS)) -> None:
        super().__init__()
        self.transforms = transforms
        self.entries = entries

    def __call__(self, sample: dict) -> dict:
        # unsqueeze samples to add a batch dimensions, as required by batchgenerators
        for entry in self.entries:
            if entry not in sample:
                if tfm.raise_error_if_entry_not_extracted:
                    raise ValueError(tfm.ENTRY_NOT_EXTRACTED_ERR_MSG.format(entry))
                continue

            np_entry = tfm.check_and_return(sample[entry], np.ndarray)
            sample[entry] = np.expand_dims(np_entry, 0)

        # apply batchgenerators transforms
        for t in self.transforms:
            sample = t(**sample)

        # squeeze samples back to original format
        for entry in self.entries:
```

(continues on next page)

(continued from previous page)

```
            np_entry = tfm.check_and_return(sample[entry], np.ndarray)
            sample[entry] = np_entry.squeeze(0)

        return sample

transforms_augmentation = [BatchgeneratorsTransform([
    bg_tfm.spatial_transforms.MirrorTransform(axes=(0, 1), data_key=defs.KEY_IMAGES,␣
↪label_key=defs.KEY_LABELS),
    bg_tfm.noise_transforms.GaussianBlurTransform(blur_sigma=(0.2, 1.0), data_key=defs.
↪KEY_IMAGES, label_key=defs.KEY_LABELS),
])]
train_transforms = tfm.ComposeTransform(
    transforms_before_augmentation + transforms_augmentation + transforms_after_
↪augmentation)
dataset.set_transform(train_transforms)
sample_batchgenerators = dataset[sample_idx]
```

TorchIO augmentation

Let's us now use TorchIO to apply a random flip and a random affine transformation. To use TorchIO, we create wrapper classes for simple integration into pymia.

[7]:
```
class TorchIOTransform(tfm.Transform):
    """Example wrapper for `TorchIO <https://github.com/fepegar/torchio>`_␣
↪transformations."""

    def __init__(self, transforms: list, entries=(defs.KEY_IMAGES, defs.KEY_LABELS)) ->␣
↪None:
        super().__init__()
        self.transforms = transforms
        self.entries = entries

    def __call__(self, sample: dict) -> dict:
        # unsqueeze samples to be 4-D tensors, as required by TorchIO
        for entry in self.entries:
            if entry not in sample:
                if tfm.raise_error_if_entry_not_extracted:
                    raise ValueError(tfm.ENTRY_NOT_EXTRACTED_ERR_MSG.format(entry))
                continue

            np_entry = tfm.check_and_return(sample[entry], np.ndarray)
            sample[entry] = np.expand_dims(np_entry, -1)

        # apply TorchIO transforms
        for t in self.transforms:
            sample = t(sample)

        # squeeze samples back to original format
        for entry in self.entries:
            np_entry = tfm.check_and_return(sample[entry].numpy(), np.ndarray)
            sample[entry] = np_entry.squeeze(-1)
```

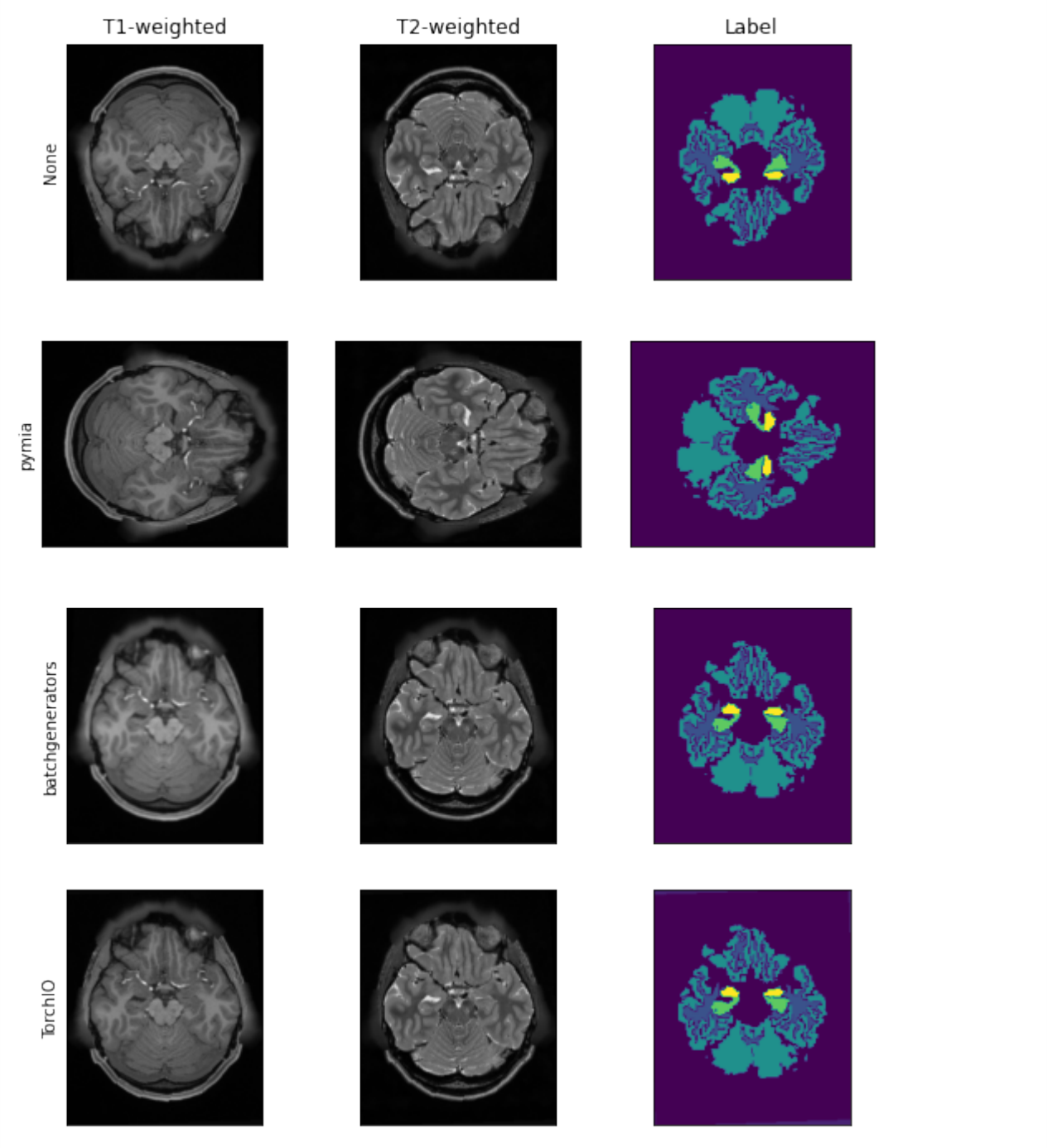
(continues on next page)

(continued from previous page)

```
        return sample

transforms_augmentation = [TorchIOTransform(
    [tio.RandomFlip(axes=('LR'), flip_probability=1.0, keys=(defs.KEY_IMAGES, defs.KEY_
↪LABELS), seed=seed),
     tio.RandomAffine(scales=(0.9, 1.2), degrees=(10), isotropic=False, default_pad_
↪value='otsu',
                      image_interpolation='NEAREST', keys=(defs.KEY_IMAGES, defs.KEY_
↪LABELS), seed=seed),
     ])]
train_transforms = tfm.ComposeTransform(
    transforms_before_augmentation + transforms_augmentation + transforms_after_
↪augmentation)
dataset.set_transform(train_transforms)
sample_torchio = dataset[sample_idx]
```

[8]:
```
# prepare and format the plot
fig, axs = plt.subplots(4, 3, figsize=(9, 12))
axs[0, 0].set_title('T1-weighted')
axs[0, 1].set_title('T2-weighted')
axs[0, 2].set_title('Label')
axs[0, 0].set_ylabel('None')
axs[1, 0].set_ylabel('pymia')
axs[2, 0].set_ylabel('batchgenerators')
axs[3, 0].set_ylabel('TorchIO')
plt.setp(axs, xticks=[], yticks=[])

axs[0, 0].imshow(sample[defs.KEY_IMAGES][0], cmap='gray')
axs[0, 1].imshow(sample[defs.KEY_IMAGES][1], cmap='gray')
axs[0, 2].imshow(sample[defs.KEY_LABELS], cmap='viridis')
axs[1, 0].imshow(sample_pymia[defs.KEY_IMAGES][0], cmap='gray')
axs[1, 1].imshow(sample_pymia[defs.KEY_IMAGES][1], cmap='gray')
axs[1, 2].imshow(sample_pymia[defs.KEY_LABELS], cmap='viridis')
axs[2, 0].imshow(sample_batchgenerators[defs.KEY_IMAGES][0], cmap='gray')
axs[2, 1].imshow(sample_batchgenerators[defs.KEY_IMAGES][1], cmap='gray')
axs[2, 2].imshow(sample_batchgenerators[defs.KEY_LABELS], cmap='viridis')
axs[3, 0].imshow(sample_torchio[defs.KEY_IMAGES][0], cmap='gray')
axs[3, 1].imshow(sample_torchio[defs.KEY_IMAGES][1], cmap='gray')
axs[3, 2].imshow(sample_torchio[defs.KEY_LABELS], cmap='viridis')
```

[8]:
```
<matplotlib.image.AxesImage at 0x7f7d44430f40>
```



Visually, we can clearly see the difference between the non-transformed and transformed images using different transformations and Python packages.

The examples are available as Jupyter notebooks and Python scripts on GitHub or directly rendered in the documentation by following the links above. Furthermore, there exist complete training scripts in TensorFlow and PyTorch at ./examples/training-examples on GitHub. For all examples, 3 tesla MR images of the head of four healthy subjects from the Human Connectome Project (HCP) [VanEssen2013] are used. Each subject has four 3-D images (in the MetaImage and Nifty format) and demographic information provided as a text file. The images are a T1-weighted MR image, a T2-weighted MR image, a label image (ground truth), and a brain mask image. The demographic information is artificially created age, gender, and grade point average (GPA). The label images contain annotations of five brain structures

(1: white matter, 2: grey matter, 3: hippocampus, 4: amygdala, and 5: thalamus [0 is background]), automatically segmented by FreeSurfer 5.3 [Fischl2012] [Fischl2002]. Therefore, the examples mimic the problem of medical image segmentation of brain tissues.

1.2.7 Projects using pymia

pymia was used for several projects, which have public code available and can serve as an additional point of reference complementing the documentation. Projects using version >= 0.3.0 are:

- Spatially Regularized Parametric Map Reconstruction for Fast Magnetic Resonance Fingerprinting: Code for the Medical Image Analysis paper by Balsiger et al. with data handling and evaluation.
- Learning Bloch Simulations for MR Fingerprinting by Invertible Neural Networks: Code for the MLMIR 2020 paper by Balsiger and Jungo et al. with evaluation.
- Medical Image Analysis Laboratory: Code for a MSc-level lecture at the University of Bern with image filtering and evaluation.

1.2.8 References

1.3 Contribution

Contributors are highly welcome on all levels such as new features, improvements, bug fixes, and documentation. Please read this guide carefully to hold a certain standard in code quality.

1.3.1 Code style

We follow the PEP 8 – Style Guide for Python Code.

1.3.2 Code documentation

Please document your code. Each package, module, class, and function should have a comment. We use Google style docstrings, and you can find a great example here. For major changes, it might also be good to update the documentation you are currently reading. It is generated with Sphinx, and you can find the source files in the `./docs` directory.

1.3.3 Code tests

You do write tests, don't you? They are located in the `./test` directory.

1.3.4 Commit messages

The commit messages follow the AngularJS Git Commit Message Conventions format:

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

Usually the first line is enough, i.e. `<type>(<scope>): <subject>` . It contains a succinct description of the change. Allowed `<type>` s are:

- `feat`: feature
- `fix`: bug fix
- `docs`: documentation
- `style`: formatting, missing semi colons, ...
- `refactor`
- `test`: when adding tests
- `chore`: maintain

An example would be: `feat(metric): add Dice coefficient metric`

1.3.5 TODOs

Mark todos like this:

```
# TODO(<name>): improve performance by vectorization
```

Where `<name>` should be replaced by your GitHub name.

1.4 Change history

The change history lists the most important changes and is not an exhaustive list.

1.4.1 Upcoming

- *SegmentationEvaluator* now verifies the input (reference and prediction) to be integer or boolean
- Extended the *examples* with augmentation and training (U-Net) scripts

1.4.2 0.3.1 (2020-08-02)

- Fixed missing dependency in `setup.py`

1.4.3 0.3.0 (2020-07-14)

- *pymia.data* package now supports PyTorch and TensorFlow. A few classes have been renamed and refactored.
- *pymia.evaluation* package with new evaluator and writer classes. Metrics are now categorized into *pymia.evaluation.metric.categorical* and *pymia.evaluation.metric.continuous* modules
- New metrics *PeakSignalToNoiseRatio* and *StructuralSimilarityIndexMeasure*
- Removed `config`, `deeplearning`, and `plotting` packages
- Improved readability of code
- Revised examples
- Revised documentation

Migration guide

Heavy changes have been made to move pymia towards a lightweight data handling and evaluation package for medical image analysis with deep learning. Therefore, this release is, unfortunately, not backward compatible. To facilitate transition to this and coming versions, we thoroughly revised the documentation and the *examples*.

1.4.4 0.2.4 (2020-05-22)

- Bug fixes in the *pymia.evaluation* package

1.4.5 0.2.3 (2019-12-13)

- Refactored: *pymia.data.transformation*
- Bug fixes and code maintenance

1.4.6 0.2.2 (2019-11-11)

- Removed the `tensorflow`, `tensorboardX`, and `torch` dependencies during installation
- Bug fixes and code maintenance

1.4.7 0.2.1 (2019-09-04)

- New statistics plotting module `pymia.plotting.statistics` (subject to heavy changes and possibly removal!)
- Bug fixes and code maintenance
- Several improvements to the documentation

1.4.8 0.2.0 (2019-04-12)

- New `pymia.deeplearning` package
- New extractor *PadDataExtractor*, which replaces the `PadPatchDataExtractor` (see migration guide below)
- New metrics *NormalizedRootMeanSquaredError*, *SurfaceDiceOverlap*, and *SurfaceOverlap*
- Faster and more generic implementation of *HausdorffDistance*
- New data augmentation module *pymia.data.augmentation*
- New filter *BinaryThreshold*
- Replaced the transformation in *SubjectAssembler* by a more flexible function (see migration guide below)
- Minor bug fixes and maintenance
- Several improvements to the documentation

We kindly appreciate the help of our contributors:

- Jan Riedo
- Yannick Soom

Migration guide

The extractor `PadPatchDataExtractor` has been replaced by the *PadDataExtractor* to facilitate the extraction flexibility. The *PadDataExtractor* works now with any kind of the three data extractors (*DataExtractor*, *RandomDataExtractor*, and *SelectiveDataExtractor*), which are passed as argument. Further, it is now possible to pass a function for the padding as argument to replace the default zero padding. Suppose you used the `PadPatchDataExtractor` like this:

```
import pymia.data.extraction as pymia_extr
pymia_extr.PadPatchDataExtractor(padding=(10, 10, 10), categories=('images',))
```

To have the same behaviour, replace it by:

```
import pymia.data.extraction as pymia_extr
pymia_extr.PadDataExtractor(padding=(10, 10, 10),
                            extractor=pymia_extr.DataExtractor(categories=('images',)))
```

The transformation in *SubjectAssembler.add_batch()* has been removed and replaced by the `on_sample_fn` parameter in the constructor. Replacing the transformation by this function should be straight forward by rewriting your transformation as function:

```
def on_sample_fn(params: dict):
  key = '__prediction'
  batch = params['batch']
  idx = params['batch_idx']

  data = params[key]
  index_expr = batch['index_expr'][idx]

  # manipulate data and index_expr according to your needs

  return data, index_expr
```

1.4.9 0.1.1 (2018-08-04)

- Improves the documentation
- Mocks the torch dependency to build the docs

1.4.10 0.1.0 (2018-08-03)

- Initial release on PyPI

1.5 Acknowledgments

pymia would not be possible without the help of contributors and also open source code bases.

1.5.1 Contributors

Following people, who are not part of the core development team, contributed to pymia (in alphabetical order by last name):

- Jan Riedo (jriedo)
- Yannick Soom (soomy)

Thank you very much, guys!

1.5.2 Open source code

Parts of pymia base on open source code, which we acknowledge hereby:

- Some distance metrics in the `pymia.evaluation.metric` package are taken from https://github.com/deepmind/surface-distance.
- The `pymia.evaluation.metric` package is largely inspired by https://github.com/Visceral-Project/EvaluateSegmentation.
- *Installation* helps you installing pymia.
- *Examples* give you an overview of pymia's intended use. Jupyter notebooks and Python scripts are available at GitHub.
- Do you want to contribute? See *Contribution*.
- *Change history*.
- *Acknowledgments*.

CHAPTER

TWO

CITATION

If you use pymia for your research, please acknowledge it accordingly by citing our paper:

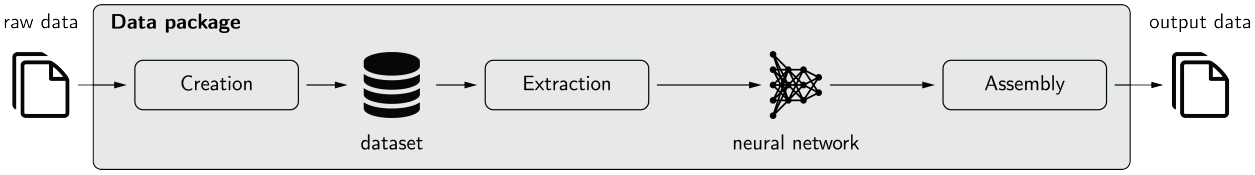
Jungo, A., Scheidegger, O., Reyes, M., & Balsiger, F. (2021). pymia: A Python package for data handling and evaluation in deep learning-based medical image analysis. Computer Methods and Programs in Biomedicine, 198, 105796

BibTeX entry:

```
@article{Jungo2021a,
author = {Jungo, Alain and Scheidegger, Olivier and Reyes, Mauricio and Balsiger, Fabian}
↪,
doi = {10.1016/j.cmpb.2020.105796},
issn = {01692607},
journal = {Computer Methods and Programs in Biomedicine},
pages = {105796},
title = {{pymia: A Python package for data handling and evaluation in deep learning-
↪based medical image analysis}},
volume = {198},
year = {2021},
}
```

2.1 Data (`pymia.data` package)

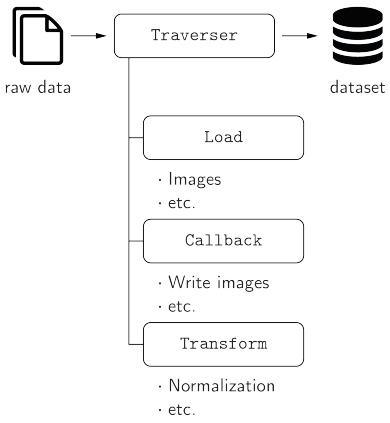
This data package provides data handling functionality for machine learning (especially deep learning) projects. The concept of the data package is illustrated in the figure below.



The three main components of the data package are creation, extraction, and assembly.

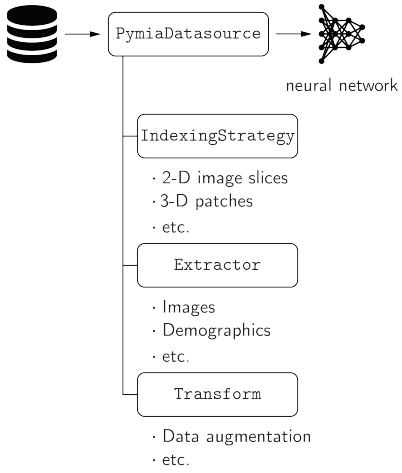
Creation

The creation of a dataset is managed by the *`Traverser`* class, which processes the data of every subject (case) iteratively. It employs *`Load`* and *`Callback`* classes to load the raw data and write it to the dataset. *`Transform`* classes can be used to apply modifications to the data, e.g., an intensity normalization. For the ease of usage, the defaults *`get_default_callbacks()`* and *`LoadDefault`* are implemented, which cover the most fundamental cases. The code example *Creation of a dataset* illustrates how to create a dataset.



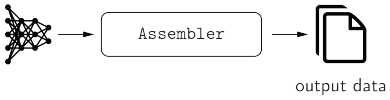
Extraction

Data extraction from the dataset is managed by the *PymiaDatasource* class, which provides a flexible interface for retrieving data, or chunks of data, to form training samples. An *IndexingStrategy* is used to define how the data is indexed, meaning accessing, for instance, an image slice or a 3-D patch of an 3-D image. *Extractor* classes extract the data from the dataset, and *Transform* classes can be used to alter the extracted data. The code example *Data extraction and assembly* illustrates how to extract data.



Assembly

The *Assembler* class manages the assembly of the predicted neural network outputs by using the identical indexing that was employed to extract the data by the *PymiaDatasource* class. The code example *Data extraction and assembly* illustrates how to assemble data.



2.1.1 Subpackages

Backends (pymia.data.backends package)

PyTorch

class pymia.data.backends.pytorch.**PytorchDatasetAdapter**(**args: Any*, ***kwargs: Any*)
A wrapper class for *PymiaDatasource* to fit the torch.utils.data.Dataset interface.

Parameters **datasource** (PymiaDatasource) – The pymia datasource instance.

class pymia.data.backends.pytorch.**SubsetSequentialSampler**(**args: Any*, ***kwargs: Any*)
Samples elements sequential from a given list of indices, without replacement.

The class adopts the torch.utils.data.Sampler interface.

Parameters **list** (*indices*) – list of indices that define the subset to be used for the sampling.

TensorFlow

pymia.data.backends.tensorflow.**get_tf_generator**(*data_source:* pymia.data.extraction.datasource.PymiaDatasource)
Returns a generator that wraps *PymiaDatasource* for the TensorFlow data handling.

The returned generator can be used with tf.data.Dataset.from_generator in order to build a TensorFlow dataset.

Parameters **data_source** (PymiaDatasource) – the datasource to be wrapped.

Returns Function that loops over the entire datasource and yields all entries.

Return type generator

Creation (pymia.data.creation package)

Callback (pymia.data.creation.callback module)

class pymia.data.creation.callback.**Callback**
Bases: object

Base class for the interaction with the dataset creation.

Implementations of the *Callback* class can be provided to *Traverser.traverse()* in order to write/process specific information of the original data.

on_end(*params: dict*)
Called at the end of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser*. The provided parameters will differ from *Callback.on_subject()*.

on_start(*params: dict*)
Called at the beginning of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser*. The provided parameters will differ from *Callback.on_subject()*.

on_subject(*params: dict*)
Called for each subject of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser* containing subject specific information and data.

class pymia.data.creation.callback.**ComposeCallback**(*callbacks: List[*pymia.data.creation.callback.Callback*]*)

Bases: *pymia.data.creation.callback.Callback*

Composes many *Callback* instances and behaves like an single *Callback* instance.

This class allows passing multiple *Callback* to *Traverser.traverse()*.

Parameters **callbacks** (*list*) – A list of *Callback* instances.

on_end(*params: dict*)
Called at the end of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser*. The provided parameters will differ from *Callback.on_subject()*.

on_start(*params: dict*)
Called at the beginning of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser*. The provided parameters will differ from *Callback.on_subject()*.

on_subject(*params: dict*)
Called for each subject of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser* containing subject specific information and data.

class pymia.data.creation.callback.**MonitoringCallback**

Bases: *pymia.data.creation.callback.Callback*

Callback that monitors the dataset creation process by logging the progress to the console.

on_end(*params: dict*)
Called at the end of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser*. The provided parameters will differ from *Callback.on_subject()*.

on_start(*params: dict*)
Called at the beginning of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser*. The provided parameters will differ from *Callback.on_subject()*.

on_subject(*params: dict*)
Called for each subject of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser* containing subject specific information and data.

class pymia.data.creation.callback.**WriteDataCallback**(*writer:* pymia.data.creation.writer.Writer)

Bases: *pymia.data.creation.callback.Callback*

Callback that writes the raw data to the dataset.

Parameters **writer** (creation.writer.Writer) – The writer used to write the data.

on_subject(*params: dict*)
Called for each subject of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser* containing subject specific information and data.

class pymia.data.creation.callback.**WriteEssentialCallback**(*writer:* pymia.data.creation.writer.Writer)

Bases: *pymia.data.creation.callback.Callback*

Callback that writes the essential information to the dataset.

Parameters **writer** (creation.writer.Writer) – The writer used to write the data.

on_start(*params: dict*)
Called at the beginning of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser*. The provided parameters will differ from *Callback.on_subject()*.

on_subject(*params: dict*)
Called for each subject of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser* containing subject specific information and data.

class pymia.data.creation.callback.**WriteFilesCallback**(*writer:* pymia.data.creation.writer.Writer)

Bases: *pymia.data.creation.callback.Callback*

Callback that writes the file names to the dataset.

Parameters **writer** (creation.writer.Writer) – The writer used to write the data.

on_start(*params: dict*)
Called at the beginning of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser*. The provided parameters will differ from *Callback.on_subject()*.

on_subject(*params: dict*)
Called for each subject of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser* containing subject specific information and data.

class pymia.data.creation.callback.**WriteImageInformationCallback**(*writer:* pymia.data.creation.writer.Writer, *category='images'*)

Bases: *pymia.data.creation.callback.Callback*

Callback that writes the image information (shape, origin, direction, spacing) to the dataset.

Parameters

- **writer** (creation.writer.Writer) – The writer used to write the data.
- **category** (*str*) – The category from which to extract the information from.

on_start(*params: dict*)
Called at the beginning of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser*. The provided parameters will differ from *Callback.on_subject()*.

on_subject(*params: dict*)
Called for each subject of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser* containing subject specific information and data.

class pymia.data.creation.callback.**WriteNamesCallback**(*writer:* pymia.data.creation.writer.Writer)
Bases: *pymia.data.creation.callback.Callback*

Callback that writes the names of the category entries to the dataset.

Parameters **writer** (creation.writer.Writer) – The writer used to write the data.

on_start(*params: dict*)
Called at the beginning of *Traverser.traverse()*.

Parameters **params** (*dict*) – Parameters provided by the *Traverser*. The provided parameters will differ from *Callback.on_subject()*.

pymia.data.creation.callback.**get_default_callbacks**(*writer:* pymia.data.creation.writer.Writer, *meta_only=False*) → *pymia.data.creation.callback.ComposeCallback*
Provides a selection of commonly used callbacks to write the most important information to the dataset.

Parameters

- **writer** (creation.writer.Writer) – The writer used to write the data.
- **meta_only** (*bool*) – Whether only callbacks for a metadata dataset creation should be returned.

Returns The composed selection of common callbacks.

Return type *Callback*

File loader (pymia.data.creation.fileloader module)

class pymia.data.creation.fileloader.**Load**
Bases: abc.ABC

Interface for loading the data during the dataset creation in *Traverser.traverse()*

abstract **__call__**(*file_name: str*, *id_: str*, *category: str*, *subject_id: str*) → Tuple[numpy.ndarray, Optional[pymia.data.conversion.ImageProperties]]
Loads the data from the file system according to the implementation.

Parameters

- **file_name** (*str*) – Path to the corresponding data.
- **id** (*str*) – Identifier for the entry of the category, e.g., "Flair".
- **category** (*str*) – Name of the category, e.g., 'images'.
- **subject_id** (*str*) – Identifier of the current subject.

Returns A numpy array containing the loaded data and ImageProperties describing the data. ImageProperties is None if the loaded data does not contain further properties.

Return type tuple

class pymia.data.creation.fileloader.**LoadDefault**
Bases: *pymia.data.creation.fileloader.Load*

The default loader.

It loads every data item (id/entry, category) for each subject as `sitk.Image` and the corresponding `ImageProperties`.

Traverser (`pymia.data.creation.traverser` module)

class pymia.data.creation.traverser.**Traverser**(*categories: Optional[Union[str, Tuple[str, ...]]] = None*)
Bases: `object`

Class managing the dataset creation process.

Parameters **categories** (*str or tuple of str*) – The categories to traverse. If None, then all categories of a *SubjectFile* will be traversed.

traverse(*subject_files: List[pymia.data.subjectfile.SubjectFile], load=<pymia.data.creation.fileloader.LoadDefault object>, callback: Optional[pymia.data.creation.callback.Callback] = None, transform: Optional[pymia.data.transformation.Transform] = None, concat_fn=<function default_concat>*)
Controls the actual dataset creation. It goes through the file list, loads the files, applies transformation to the data, and calls the callbacks to do the storing (or other stuff).

Parameters

- **subject_files** (*list*) – list of `SubjectFile` to be processes.
- **load** (*callable*) – A load function or *Load* instance that performs the data loading
- **callback** (`Callback`) – A callback or composed (*ComposeCallback*) callback performing the storage of the loaded data (and other things such as logging).
- **transform** (`Transform`) – Transformation to be applied to the data after loading and before `Callback.on_subject()` is called
- **concat_fn** (*callable*) – Function that concatenates all the entries of a category (e.g. T1, T2 data from "images" category). Default is *default_concat()*.

pymia.data.creation.traverser.**default_concat**(*data: List[numpy.ndarray]*) → numpy.ndarray
Default concatenation function used to combine all entries from a category (e.g. T1, T2 data from "images" category) in *Traverser.traverse()*

Parameters **data** (*list*) – List of numpy.ndarray entries to be concatenated.

Returns Concatenated entry.

Return type numpy.ndarray

Writer (`pymia.data.creation.writer` module)

class pymia.data.creation.writer.**Hdf5Writer**(*file_path: str*)
Bases: *pymia.data.creation.writer.Writer*

Writer class for HDF5 file type.

Parameters **file_path** (*str*) – The path to the dataset file to write.

close()
see *Writer.close()*

fill(*entry: str, data, index: Optional[pymia.data.indexexpression.IndexExpression] = None*)
see *Writer.fill()*

open()
see *Writer.open()*

reserve(*entry: str*, *shape: tuple*, *dtype=None*)
see *Writer.reserve()*

write(*entry: str*, *data*, *dtype=None*)
see *Writer.write()*

class pymia.data.creation.writer.**Writer**
Bases: abc.ABC

Represents the abstract dataset writer defining an interface for the writing process.

abstract close()
Close the writer.

abstract fill(*entry: str*, *data*, *index: Optional[*pymia.data.indexexpression.IndexExpression*] = None*)
Fill parts of a reserved dataset entry.

Parameters

- **entry** (*str*) – The dataset entry to be filled.
- **data** (*object*) – The data to write.
- **index** (IndexExpression) – The slicing expression.

abstract open()
Open the writer.

abstract reserve(*entry: str*, *shape: tuple*, *dtype=None*)
Reserve space in the dataset for later writing.

Parameters

- **entry** (*str*) – The dataset entry to be created.
- **shape** (*tuple*) – The shape to be reserved.
- **dtype** – The dtype.

abstract write(*entry: str*, *data*, *dtype=None*)
Create and write entry.

Parameters

- **entry** (*str*) – The dataset entry to be written.
- **data** (*object*) – The data to write.
- **dtype** – The dtype.

pymia.data.creation.writer.**get_writer**(*file_path: str*) → *pymia.data.creation.writer.Writer*
Get the dataset writer corresponding to the file extension.

Parameters **file_path** (*str*) – The path of the dataset file to be written.

Returns Writer corresponding to dataset file extension.

Return type *creation.writer.Writer*

pymia.data.creation.writer.**writer_registry = {'.h5': <class 'pymia.data.creation.writer.Hdf5Writer'>, '.hdf5': <class 'pymia.data.creation.writer.Hdf5Writer'>}**
Registry defining the mapping between file extension and *Writer* class. Alternative writers need to be added to this registry in order to use *get_writer()*.

Extraction (pymia.data.extraction package)

Datasource (pymia.data.extraction.datasource module)

class pymia.data.extraction.datasource.**PymiaDatasource**(*dataset_path: str*, *indexing_strategy: Optional[pymia.data.extraction.indexing.IndexingStrategy] = None*, *extractor: Optional[pymia.data.extraction.extractor.Extractor] = None*, *transform: Optional[pymia.data.transformation.Transform] = None*, *subject_subset: Optional[list] = None*, *init_reader_once: bool = True*)

Bases: object

Provides convenient and adaptable reading of the data from a created dataset.

Parameters

- **dataset_path** (*str*) – The path to the dataset to be read from.
- **indexing_strategy** (IndexingStrategy) – Strategy defining how the data is indexed for reading.
- **extractor** (Extractor) – Extractor or multiple extractors (*ComposeExtractor*) extracting the desired data from the dataset.
- **transform** (Transform) – Transformation(s) to be applied to the extracted data.
- **subject_subset** (*list*) – A list of subject identifiers defining a subset of subject to be processed.
- **init_reader_once** (*bool*) – Whether the reader is initialized once or for every retrieval (default: True)

Examples

The class mainly allows to modes of operation. The first mode is by extracting the data by index.

```
>>> ds = PymiaDatasource(...)
>>> for i in range(len(ds)):
>>>     sample = ds[i]
```

The second mode of operation is by directly extracting data.

```
>>> ds = PymiaDatasource(...)
>>> # Different from ds[index] since the extractor and transform override the ones␣
↪in ds
>>> sample = ds.direct_extract(extractor, index, transform=transform)
```

Typically, the first mode is use to loop over the entire dataset as fast as possible, extracting just the necessary information, such as data chunks (e.g., slice, patch, sub-volume). Less critical information (e.g. image shape, orientation) not required with every chunk of data can independently be extracted with the second mode of operation.

close_reader()
Close the reader.

direct_extract(*extractor:* pymia.data.extraction.extractor.Extractor, *subject_index: int*, *index_expr: Optional[*pymia.data.indexexpression.IndexExpression*] = None*, *transform: Optional[*pymia.data.transformation.Transform*] = None*)

Extract data directly, bypassing the extractors and transforms of the instance.

The purpose of this method is to enable extraction of data that is not required for every data chunk (e.g., slice, patch, sub-volume) but only from time to time e.g., image shape, origin.

Parameters

- **extractor** (Extractor) – Extractor or multiple extractors (*ComposeExtractor*) extracting the desired data from the dataset.
- **subject_index** (*int*) – Index of the subject to be extracted.
- **index_expr** (IndexExpression) – The indexing to extract a chunk of data only. Not required if only image related information (e.g., image shape, origin) should be extracted. Needed when desiring a chunk of data (e.g., slice, patch, sub-volume).
- **transform** (Transform) – Transformation(s) to be applied to the extracted data.

Returns Extracted data in a dictionary. Keys are defined by the used *Extractor*.

Return type dict

get_subjects()

"Get all the subjects in the dataset.

Returns All subject identifiers in the dataset.

Return type list

indices

A list containing all sample indices. This is a mapping from item *i* to tuple *(subject_index, index_expression)*.

Type list

set_extractor(*extractor:* pymia.data.extraction.extractor.Extractor)

Set the extractor(s).

Parameters **extractor** (Extractor) – Extractor or multiple extractors (*ComposeExtractor*) extracting the desired data from the dataset.

set_indexing_strategy(*indexing_strategy:* pymia.data.extraction.indexing.IndexingStrategy, *subject_subset: Optional[list] = None*)

Set (or modify) the indexing strategy.

Parameters

- **indexing_strategy** (IndexingStrategy) – Strategy defining how the data is indexed for reading.
- **subject_subset** (*list*) – A list of subject identifiers defining a subset of subject to be processed.

set_transform(*transform:* pymia.data.transformation.Transform)

Set the transform.

Parameters **transform** (Transform) – Transformation(s) to be applied to the extracted data.

Extractor (pymia.data.extraction.extractor module)

class pymia.data.extraction.extractor.**ComposeExtractor**(*extractors: list*)
Bases: *pymia.data.extraction.extractor.Extractor*

Composes many *Extractor* instances and behaves like an single *Extractor* instance.

Parameters **extractors** (*list*) – A list of *Extractor* instances.

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**DataExtractor**(*categories=('images',)*, *ignore_indexing: bool = False*)
Bases: *pymia.data.extraction.extractor.Extractor*

Extracts data of a given category.

Adds `category` as key to `extracted`.

Parameters

- **categories** (*tuple*) – Categories for which to extract the names.
- **ignore_indexing** (*bool*) – Whether to ignore the indexing in `params`. This is useful when extracting entire images.

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**Extractor**
Bases: `abc.ABC`

Interface unifying the extraction of data from a dataset.

abstract extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
Extract data from the dataset.

Parameters

- **reader** (Reader) – Reader instance that can read from dataset.
- **params** (*dict*) – Extraction parameters containing information such as subject index and index expression.
- **extracted** (*dict*) – The dictionary to put the extracted data in.

class pymia.data.extraction.extractor.**FilesExtractor**(*cache: bool = True*, *categories=('images', 'labels')*)
Bases: *pymia.data.extraction.extractor.Extractor*

Extracts the file paths.

Added key to `extracted`:

- *pymia.data.definition.KEY_FILE_ROOT* with `str` content
- *pymia.data.definition.KEY_PLACEHOLDER_FILES* with `str` content

Parameters

- **cache** (*bool*) – Whether to cache the results. If `True`, the dataset is only accessed once. `True` is often preferred since the file name entries are typically unique in the dataset (i.e. independent of data chunks).
- **categories** (*tuple*) – Categories for which to extract the file names.

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**FilesystemDataExtractor**(*categories=('images',)*, *load_fn=None*, *ignore_indexing: bool = False*, *override_file_root=None*)

Bases: *pymia.data.extraction.extractor.Extractor*

Extracts data of a given category.

Adds `category` as key to `extracted`.

Parameters

- **categories** (*tuple*) – Categories for which to extract the names.
- **load_fn** (*callable*) – Callable that loads a file given the file path and the category, and returns a numpy.ndarray.
- **ignore_indexing** (*bool*) – Whether to ignore the indexing in `params`. This is useful when extracting entire images.

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**ImagePropertiesExtractor**(*do_pickle: bool = False*)

Bases: *pymia.data.extraction.extractor.Extractor*

Extracts the image properties.

Added key to `extracted`:

- *pymia.data.definition.KEY_PROPERTIES* with `ImageProperties` content (or byte if `do_pickle`)

Parameters **do_pickle** (*bool*) – whether to pickle the extracted `ImageProperties` instance. This allows usage in multiprocessing environment.

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**ImagePropertyShapeExtractor**(*numpy_format: bool = True*)

Bases: *pymia.data.extraction.extractor.Extractor*

Extracts the shape image property of an image.

Added key to `extracted`:

- *pymia.data.definition.KEY_SHAPE* with `tuple` content

Parameters **numpy_format** (*bool*) – Whether the shape is numpy or ITK format (first and last dimension are swapped).

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**IndexingExtractor**(*do_pickle: bool = False*)

Bases: *pymia.data.extraction.extractor.Extractor*

Extracts the index expression.

Added key to `extracted`:

- *pymia.data.definition.KEY_SUBJECT_INDEX* with `int` content

- *pymia.data.definition.KEY_INDEX_EXPR* with *IndexExpression* content

Parameters **do_pickle** (*bool*) – whether to pickle the extracted `ImageProperties` instance. This is useful when applied with PyTorch DataLoader since it prevents from automatic translation to torch.Tensor.

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**NamesExtractor**(*cache: bool = True*, *categories=('images', 'labels')*)

Bases: *pymia.data.extraction.extractor.Extractor*

Extracts the names of the entries within a category (e.g. "Flair", "T1" for the category "images").

Added key to `extracted`:

- *pymia.data.definition.KEY_PLACEHOLDER_NAMES* with `str` content

Parameters

- **cache** (*bool*) – Whether to cache the results. If `True`, the dataset is only accessed once. `True` is often preferred since the name entries are typically unique in the dataset.
- **categories** (*tuple*) – Categories for which to extract the names.

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**PadDataExtractor**(*padding: Union[tuple, List[tuple]]*, *extractor:* pymia.data.extraction.extractor.Extractor, *pad_fn=None*)

Bases: *pymia.data.extraction.extractor.Extractor*

Pads the data extracted by `extractor`

Parameters

- **padding** (*tuple, list*) – Lengths of the tuple or the list must be equal to the number of dimensions of the extracted data. If tuple, values are considered as symmetric padding in each dimension. If list, the each entry must consist of a tuple indicating (left, right) padding for one dimension.
- **extractor** (`Extractor`) – The extractor performing the extraction of the data to be padded.
- **pad_fn** (*callable, optional*) – Optional function performing the padding. Default is `PadDataExtractor.zero_pad()`.

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**RandomDataExtractor**(*selection=None*, *category: str = 'labels'*)

Bases: *pymia.data.extraction.extractor.Extractor*

Extracts data of a given category randomly.

Adds `category` as key to `extracted`.

Parameters

- **selection** (*str, tuple*) – Entries (e.g., "T1", "T2") within the category to select an entry randomly from. If selection is None, an entry from all entries is randomly selected.

- **category** (*str*) – The category (e.g. "images") to extract data from.

Note: Requires results of *NamesExtractor* in `extracted`.

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**SelectiveDataExtractor**(*selection=None*, *category: str = 'labels'*)

Bases: *pymia.data.extraction.extractor.Extractor*

Extracts data of a given category selectively.

Adds `category` as key to `extracted`.

Parameters

- **selection** (*str, tuple*) – Entries (e.g., "T1", "T2") within the category to select. If selection is None, the class has the same behaviour as the DataExtractor and selects all entries.
- **category** (*str*) – The category (e.g. "images") to extract data from.

Note: Requires results of *NamesExtractor* in `extracted`.

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

class pymia.data.extraction.extractor.**SubjectExtractor**

Bases: *pymia.data.extraction.extractor.Extractor*

Extracts the subject's identification.

Added key to `extracted`:

- *pymia.data.definition.KEY_SUBJECT_INDEX* with `int` content
- *pymia.data.definition.KEY_SUBJECT* with `str` content

extract(*reader:* pymia.data.extraction.reader.Reader, *params: dict*, *extracted: dict*) → None
see *Extractor.extract()*

Indexing (pymia.data.extraction.indexing module)

class pymia.data.extraction.indexing.**EmptyIndexing**

Bases: *pymia.data.extraction.indexing.IndexingStrategy*

An empty indexing strategy. This is useful when a strategy is required but entire images should be extracted.

class pymia.data.extraction.indexing.**IndexingStrategy**

Bases: `abc.ABC`

Interface for indexing strategies that can be applied to images.

abstract __call__(*shape: tuple*) → List[*pymia.data.indexexpression.IndexExpression*]
Calculate the indexes for a given shape

Parameters **shape** (*tuple*) – The shape to determine the indexes for.

Returns The list of *IndexExpression* instances defining the indexes for an image shape.

Return type list

__repr__() → str

Returns Representation of the strategy. Should include attributes such that it uniquely defines the strategy.

Return type str

class pymia.data.extraction.indexing.**PatchWiseIndexing**(*patch_shape: tuple*, *ignore_incomplete=True*)

Bases: *pymia.data.extraction.indexing.IndexingStrategy*

Strategy to generate indices for patches (sub-volumes) of an image.

Parameters

- **patch_shape** (*tuple*) – The patch shape.
- **ignore_incomplete** (*bool*) – If even division of image by patch shape ignore incomplete patch on True. Boundary condition.

class pymia.data.extraction.indexing.**SliceIndexing**(*slice_axis: Union[int, tuple] = 0*)

Bases: *pymia.data.extraction.indexing.IndexingStrategy*

Strategy to generate a slice-wise indexing.

Parameters **slice_axis** (*int, tuple*) – The axis to be sliced. Multi-axis slicing can be achieved by providing a tuple of axes.

class pymia.data.extraction.indexing.**VoxelWiseIndexing**(*image_dimension: int = 3*)

Bases: *pymia.data.extraction.indexing.IndexingStrategy*

Strategy to generate indices for every voxel of an image.

Parameters **image_dimension** (*int*) – The image dimension without the dimension of the voxels itself.

Reader (pymia.data.extraction.reader module)

class pymia.data.extraction.reader.**Hdf5Reader**(*file_path: str*, *category='images'*)

Bases: *pymia.data.extraction.reader.Reader*

Represents the dataset reader for HDF5 files.

Initializes a new instance.

Parameters

- **file_path** (*str*) – The path to the dataset file.
- **category** (*str*) – The category of an entry that defines the shape request

close()

see *Reader.close()*

get_shape(*subject_index: int*) → list

see *Reader.get_shape()*

get_subject_entries() → list

see *Reader.get_subject_entries()*

get_subjects() → list
see *Reader.get_subjects()*

has(*entry: str*) → bool
see *Reader.has()*

open()
see *Reader.open()*

read(*entry: str*, *index: Optional[*pymia.data.indexexpression.IndexExpression*] = None*)
see *Reader.read()*

class pymia.data.extraction.reader.**Reader**(*file_path: str*)
Bases: abc.ABC

Abstract dataset reader.

Parameters **file_path** (*str*) – The path to the dataset file.

abstract close()
Close the reader.

abstract get_shape(*subject_index: int*) → list
Get the shape from an entry.

Parameters **subject_index** (*int*) – The index of the subject.

Returns The shape of each dimension.

Return type list

abstract get_subject_entries() → list
Get the dataset entries holding the subject's data.

Returns The list of subject entry strings.

Return type list

abstract get_subjects() → list
Get the subject names in the dataset.

Returns The list of subject names.

Return type list

abstract has(*entry: str*) → bool
Check whether a dataset entry exists.

Parameters **entry** (*str*) – The dataset entry.

Returns Whether the entry exists.

Return type bool

abstract open()
Open the reader.

abstract read(*entry: str*, *index: Optional[*pymia.data.indexexpression.IndexExpression*] = None*)
Read a dataset entry.

Parameters

- **entry** (*str*) – The dataset entry.
- **index** (*expr.IndexExpression*) – The slicing expression.

Returns The read data.

pymia.data.extraction.reader.**get_reader**(*file_path: str*, *direct_open: bool = False*) → *pymia.data.extraction.reader.Reader*

Get the dataset reader corresponding to the file extension.

Parameters

- **file_path** (*str*) – The path to the dataset file.
- **direct_open** (*bool*) – Whether the file should directly be opened.

Returns Reader corresponding to dataset file extension.

Return type *Reader*

pymia.data.extraction.reader.**reader_registry = {'.h5': <class 'pymia.data.extraction.reader.Hdf5Reader'>, '.hdf5': <class 'pymia.data.extraction.reader.Hdf5Reader'>}**

Registry defining the mapping between file extension and *Reader* class. Alternative writers need to be added to this registry in order to use *get_reader()*.

Selection (pymia.data.extraction.selection module)

class pymia.data.extraction.selection.**ComposeSelection**(*strategies*)

Bases: *pymia.data.extraction.selection.SelectionStrategy*

class pymia.data.extraction.selection.**NonBlackSelection**(*black_value: float = 0.0*)

Bases: *pymia.data.extraction.selection.SelectionStrategy*

class pymia.data.extraction.selection.**NonConstantSelection**(*loop_axis=None*)

Bases: *pymia.data.extraction.selection.SelectionStrategy*

class pymia.data.extraction.selection.**PercentileSelection**(*percentile: float*)

Bases: *pymia.data.extraction.selection.SelectionStrategy*

class pymia.data.extraction.selection.**SelectionStrategy**

Bases: abc.ABC

Interface for selecting indices according some rule.

abstract __call__(*sample: dict*) → bool

Parameters **sample** (*dict*) – An extracted from *PymiaDatasource*.

Returns Whether or not the sample should be considered.

Return type bool

__repr__() → str

Returns Representation of the strategy. Should include attributes such that it uniquely defines the strategy.

Return type str

class pymia.data.extraction.selection.**SubjectSelection**(*subjects*)

Bases: *pymia.data.extraction.selection.SelectionStrategy*

Select subjects by their name or index.

class pymia.data.extraction.selection.**WithForegroundSelection**

Bases: *pymia.data.extraction.selection.SelectionStrategy*

2.1.2 Assembler (pymia.data.assembler module)

class pymia.data.assembler.**ApplyTransformInteractionFn**(*transform:* pymia.data.transformation.Transform)

Bases: *pymia.data.assembler.AssembleInteractionFn*

class pymia.data.assembler.**AssembleInteractionFn**

Bases: object

Function interface enabling interaction with the *index_expression* and the *data* before it gets added to the assembled *prediction* in *SubjectAssembler*.

__call__(*key*, *data*, *index_expr*, ***kwargs*)

Parameters

- **key** (*str*) – The identifier or key of the data.
- **data** (*numpy.ndarray*) – The data.
- **index_expr** (IndexExpression) – The current index_expression that might be modified.
- ****kwargs** (*dict*) – Any other arguments

Returns Modified *data* and modified *index_expression*

Return type tuple

class pymia.data.assembler.**Assembler**

Bases: abc.ABC

Interface for assembling images from batch, which contain parts (chunks) of the images only.

abstract add_batch(*to_assemble*, *sample_indices*, *last_batch=False*, ***kwargs*)

Add the batch results to be assembled.

Parameters

- **to_assemble** (*object, dict*) – object or dictionary of objects to be assembled to an image.
- **sample_indices** (*iterable*) – iterable of all the sample indices in the processed batch
- **last_batch** (*bool*) – Whether the current batch is the last.

abstract get_assembled_subject(*subject_index: int*)

Parameters **subject_index** (*int*) – Index of the assembled subject to be retrieved.

Returns The assembled data of the subject (might be multiple arrays).

Return type object

abstract property subjects_ready

The indices of the subjects that are finished assembling.

Type list, set

class pymia.data.assembler.**PlaneSubjectAssembler**(*datasource: pymia.data.extraction.datasource.PymiaDatasource*, *merge_fn=<function mean_merge_fn>*, *zero_fn=<function numpy_zeros>*)

Bases: *pymia.data.assembler.Assembler*

Assembles predictions of one or multiple subjects where predictions are made in all three planes.

This class assembles the prediction from all planes (axial, coronal, sagittal) and merges the prediction according to `merge_fn`.

Assumes that the network output, i.e. to_assemble, is of shape (B, ..., C) where B is the batch size and C is the numbers of channels (must be at least 1) and ... refers to an arbitrary image dimension.

Parameters

- **datasource** (PymiaDatasource) – The datasource
- **merge_fn** – A function that processes a sample. Args: planes: list with the assembled prediction for all planes. Returns: Merged numpy.ndarray
- **zero_fn** – A function that initializes the numpy array to hold the predictions. Args: shape: tuple with the shape of the subject's labels, id: str identifying the subject. Returns: A np.ndarray

add_batch(*to_assemble: Union[numpy.ndarray, Dict[str, numpy.ndarray]]*, *sample_indices: numpy.ndarray*, *last_batch=False*, ***kwargs*)
see *Assembler.add_batch()*

get_assembled_subject(*subject_index: int*)
see *Assembler.get_assembled_subject()*

property subjects_ready
see *Assembler.subjects_ready()*

class pymia.data.assembler.**Subject2dAssembler**(*datasource: pymia.data.extraction.datasource.PymiaDatasource*)

Bases: *pymia.data.assembler.Assembler*

Assembles predictions of two-dimensional images.

Two-dimensional images do not specifically require assembling. For pipeline compatibility reasons this class provides , nevertheless, a implementation for the two-dimensional case.

Parameters **datasource** (PymiaDatasource) – The datasource

add_batch(*to_assemble: Union[numpy.ndarray, Dict[str, numpy.ndarray]]*, *sample_indices: numpy.ndarray*, *last_batch=False*, ***kwargs*)
see *Assembler.add_batch()*

get_assembled_subject(*subject_index*)
see *Assembler.get_assembled_subject()*

property subjects_ready
see *Assembler.subjects_ready()*

class pymia.data.assembler.**SubjectAssembler**(*datasource: pymia.data.extraction.datasource.PymiaDatasource*, *zero_fn=<function numpy_zeros>*, *assemble_interaction_fn=None*)

Bases: *pymia.data.assembler.Assembler*

Assembles predictions of one or multiple subjects.

Assumes that the network output, i.e. to_assemble, is of shape (B, ..., C) where B is the batch size and C is the numbers of channels (must be at least 1) and ... refers to an arbitrary image dimension.

Parameters

- **datasource** (PymiaDatasource) – The datasource.

- **zero_fn** – A function that initializes the numpy array to hold the predictions. Args: shape: tuple with the shape of the subject's labels. Returns: A np.ndarray
- **assemble_interaction_fn** (*callable, optional*) – A *callable* that may modify the sample and indexing before adding the data to the assembled array. This enables handling special cases. Must follow the `.AssembleInteractionFn.__call__` interface. By default neither data nor indexing is modified.

add_batch(*to_assemble: Union[numpy.ndarray, Dict[str, numpy.ndarray]]*, *sample_indices: numpy.ndarray*, *last_batch=False*, ***kwargs*)
Add the batch results to be assembled.

Parameters

- **to_assemble** (*object, dict*) – object or dictionary of objects to be assembled to an image.
- **sample_indices** (*iterable*) – iterable of all the sample indices in the processed batch
- **last_batch** (*bool*) – Whether the current batch is the last.

get_assembled_subject(*subject_index: int*)
see *Assembler.get_assembled_subject()*

property subjects_ready
see *Assembler.subjects_ready()*

2.1.3 Augmentation (pymia.data.augmentation module)

This module holds classes for data augmentation.

The data augmentation bases on the transformation concept (see *pymia.data.transformation.Transform*) and can easily be incorporated into the data loading process.

See also:

The pymia documentation features an code example for Augmentation, which shows how to apply data augmentation in conjunction with the *pymia.data* package. Besides transformations from the *pymia.data.augmentation* module, transformations from the Python packages batchgenerators and TorchIO are integrated.

Warning: The augmentation relies on the random number generator of `numpy`. If you want to obtain reproducible result, set numpy's seed prior to executing any augmentation:

```
>>> import numpy as np
>>> your_seed = 0
>>> np.random.seed(your_seed)
```

class pymia.data.augmentation.**RandomCrop**(*shape: Union[int, tuple]*, *axis: Optional[Union[int, tuple]] = None*, *p: float = 1.0*, *entries=('images', 'labels')*)
Bases: *pymia.data.transformation.Transform*

Randomly crops the sample to the specified shape.

The sample shape must be bigger than the crop shape.

Notes

A probability lower than 1.0 might make not much sense because it results in inconsistent output dimensions.

Parameters

- **shape** (*int, tuple*) – The shape of the sample after the cropping. If axis is not defined, the cropping will be applied from the first dimension onwards of the sample. Use None to exclude an axis or define axis to specify the axis/axes to crop. E.g.:
  - shape=256 with the default axis parameter results in a shape of 256 x …
  - shape=(256, 128) with the default axis parameter results in a shape of 256 x 128 x …
  - shape=(None, 256) with the default axis parameter results in a shape of <as before> x 256 x …
  - shape=(256, 128) with axis=(1, 0) results in a shape of 128 x 256 x …
  - shape=(None, 128, 256) with axis=(1, 2, 0) results in a shape of 256 x <as before> x 256 x …
- **axis** (*int, tuple*) – Axis or axes to which the shape int or tuple correspond(s) to. If defined, must have the same length as shape.
- **p** (*float*) – The probability of the cropping to be applied.
- **entries** (*tuple*) – The sample's entries to apply the cropping to.

class pymia.data.augmentation.**RandomElasticDeformation**(*num_control_points: int = 4*, *deformation_sigma: float = 5.0*, *interpolators: tuple = (3, 1)*, *spatial_rank: int = 2*, *fill_value: float = 0.0*, *p: float = 0.5*, *entries=('images', 'labels')*)

Bases: *pymia.data.transformation.Transform*

Randomly transforms the sample elastically.

Notes

The code bases on NiftyNet's RandomElasticDeformationLayer class (version 0.3.0).

Warning: Always inspect the results of this transform on some samples (especially for 3-D data).

Parameters

- **num_control_points** (*int*) – The number of control points for the b-spline mesh.
- **deformation_sigma** (*float*) – The maximum deformation along the deformation mesh.
- **interpolators** (*tuple*) – The SimpleITK interpolators to use for each entry in entries.
- **spatial_rank** (*int*) – The spatial rank (dimension) of the sample.
- **fill_value** (*float*) – The fill value for the resampling.
- **p** (*float*) – The probability of the elastic transformation to be applied.
- **entries** (*tuple*) – The sample's entries to apply the elastic transformation to.

class pymia.data.augmentation.**RandomMirror**(*axis: int = - 2*, *p: float = 1.0*, *entries=('images', 'labels')*)
Bases: *pymia.data.transformation.Transform*

Randomly mirrors the sample along a given axis.

Parameters

- **p** (*float*) – The probability of the mirroring to be applied.
- **axis** (*int*) – The axis to apply the mirroring.
- **entries** (*tuple*) – The sample's entries to apply the mirroring to.

class pymia.data.augmentation.**RandomRotation90**(*axes: Tuple[int] = (- 3, - 2)*, *p: float = 1.0*, *entries=('images', 'labels')*)
Bases: *pymia.data.transformation.Transform*

Randomly rotates the sample 90, 180, or 270 degrees in the plane specified by axes.

Raises **UserWarning** – If the plane to rotate is not rectangular.

Parameters

- **axes** (*tuple*) – The sample is rotated in the plane defined by the axes. Axes must be of length two and different.
- **p** (*float*) – The probability of the rotation to be applied.
- **entries** (*tuple*) – The sample's entries to apply the rotation to.

class pymia.data.augmentation.**RandomShift**(*shift: Union[int, tuple]*, *axis: Optional[Union[int, tuple]] = None*, *p: float = 1.0*, *entries=('images', 'labels')*)
Bases: *pymia.data.transformation.Transform*

Randomly shifts the sample along axes by a value from the interval [-p * size(axis), +p * size(axis)], where p is the percentage of shifting and size(axis) is the size along an axis.

Parameters

- **shift** (*int, tuple*) – The percentage of shifting of the axis' size. If axis is not defined, the shifting will be applied from the first dimension onwards of the sample. Use None to exclude an axis or define axis to specify the axis/axes to crop. E.g.:
  - shift=0.2 with the default axis parameter shifts the sample along the 1st axis.
  - shift=(0.2, 0.1) with the default axis parameter shifts the sample along the 1st and 2nd axes.
  - shift=(None, 0.2) with the default axis parameter shifts the sample along the 2st axis.
  - shift=(0.2, 0.1) with axis=(1, 0) shifts the sample along the 1st and 2nd axes.
  - shift=(None, 0.1, 0.2) with axis=(1, 2, 0) shifts the sample along the 1st and 3rd axes.
- **axis** (*int, tuple*) – Axis or axes to which the shift int or tuple correspond(s) to. If defined, must have the same length as shape.
- **p** (*float*) – The probability of the shift to be applied.
- **entries** (*tuple*) – The sample's entries to apply the shifting to.

2.1.4 Conversion (pymia.data.conversion module)

This module holds classes related to image conversion.

The main purpose of this module is the conversion between SimpleITK images and numpy arrays.

class pymia.data.conversion.**NumpySimpleITKImageBridge**

Bases: object

A numpy to SimpleITK bridge, which provides static methods to convert between numpy array and SimpleITK image.

static convert(*array: numpy.ndarray*, *properties: pymia.data.conversion.ImageProperties*) → SimpleITK.SimpleITK.Image

Converts a numpy array to a SimpleITK image.

Parameters

- **array** (*np.ndarray*) – The image as numpy array. The shape can be either:
  - shape=(n,), where n = total number of voxels
  - shape=(n,v), where n = total number of voxels and v = number of components per pixel (vector image)
  - shape=(<reversed image size>), what you get from sitk.GetArrayFromImage()
  - **shape=(<reversed image size>,v), what you get from sitk.GetArrayFromImage()** and v = number of components per pixel (vector image)
- **properties** (*ImageProperties*) – The image properties.

Returns The SimpleITK image.

Return type sitk.Image

class pymia.data.conversion.**SimpleITKNumpyImageBridge**

Bases: object

A SimpleITK to numpy bridge.

Converts SimpleITK images to numpy arrays. Use the NumpySimpleITKImageBridge to convert back.

static convert(*image: SimpleITK.SimpleITK.Image*) → Tuple[numpy.ndarray, pymia.data.conversion.ImageProperties]

Converts an image to a numpy array and an ImageProperties class.

Parameters **image** (*SimpleITK.Image*) – The image.

Returns The image as numpy array and the image properties.

Return type A Tuple[np.ndarray, ImageProperties]

Raises **ValueError** – If *image* is *None*.

2.1.5 Definition (pymia.data.definition module)

This module contains global definitions for the *pymia.data* package.

pymia.data.definition.**KEY_CATEGORIES = 'categories'**

pymia.data.definition.**KEY_FILE_ROOT = 'file_root'**

pymia.data.definition.**KEY_IMAGES = 'images'**

pymia.data.definition.**KEY_INDEX_EXPR = 'index_expr'**

pymia.data.definition.**KEY_LABELS = 'labels'**

pymia.data.definition.**KEY_PLACEHOLDER_FILES = '{}_files'**

pymia.data.definition.**KEY_PLACEHOLDER_NAMES = '{}_names'**

pymia.data.definition.**KEY_PLACEHOLDER_PROPERTIES = '{}_properties'**

pymia.data.definition.**KEY_PROPERTIES = 'properties'**

pymia.data.definition.**KEY_SAMPLE_INDEX = 'sample_index'**

pymia.data.definition.**KEY_SHAPE = 'shape'**

pymia.data.definition.**KEY_SUBJECT = 'subject'**

pymia.data.definition.**KEY_SUBJECT_FILES = 'subject_files'**

pymia.data.definition.**KEY_SUBJECT_INDEX = 'subject_index'**

2.1.6 Index expression (pymia.data.indexexpression module)

class pymia.data.indexexpression.**IndexExpression**(*indexing: Optional[Union[int, tuple, List[int], List[tuple], List[list]]] = None*, *axis: Optional[Union[int, tuple]] = None*)

Bases: `object`

Defines the indexing of a chunk of raw data in the dataset.

Parameters

- **indexing** (*int, tuple, list*) – The indexing. If `int` or list of `int`, individual entries of and axis are indexed. If `tuple` or list of `tuple`, the axis should be sliced.
- **axis** (*int, tuple*) – The axis/axes to the corresponding indexing. If `tuple`, the length has to be equal to the list length of `indexing`

expression
list of `slice` objects defining the slicing each axis

get_indexing()

Returns a list tuples defining the indexing (i.e., None, index, (start, stop)) at each axis. Can be used to generate a new index expression.

Return type list

set_indexing(*indexing: Union[int, tuple, slice, List[int], List[tuple], List[list]]*, *axis: Optional[Union[int, tuple]] = None*)

2.1.7 Subject file (pymia.data.subjectfile module)

class pymia.data.subjectfile.**FileCategory**(*entries=None*)
Bases: object

class pymia.data.subjectfile.**SubjectFile**(*subject: str*, ***file_groups*)
Bases: object

Holds the file information of a subject.

Parameters

- **subject** (*str*) – The subject identifier.
- ****file_groups** (*dict*) – The groups of file types containing the file path entries.

get_all_files()

Returns All file path entries of a subject *flattened* (without groups/category).

Return type dict

2.1.8 Transformation (pymia.data.transformation module)

class pymia.data.transformation.**ClipPercentile**(*upper_percentile: float*, *lower_percentile: Optional[float] = None*, *loop_axis=None*, *entries=('images',)*)
Bases: *pymia.data.transformation.LoopEntryTransform*

transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

class pymia.data.transformation.**ComposeTransform**(*transforms: Iterable[*pymia.data.transformation.Transform*]*)
Bases: *pymia.data.transformation.Transform*

class pymia.data.transformation.**IntensityNormalization**(*loop_axis=None*, *entries=('images',)*)
Bases: *pymia.data.transformation.LoopEntryTransform*

transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

class pymia.data.transformation.**IntensityRescale**(*lower*, *upper*, *loop_axis=None*, *entries=('images',)*)
Bases: *pymia.data.transformation.LoopEntryTransform*

transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

class pymia.data.transformation.**LambdaTransform**(*lambda_fn*, *loop_axis=None*, *entries=('images',)*)
Bases: *pymia.data.transformation.LoopEntryTransform*

transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

class pymia.data.transformation.**LoopEntryTransform**(*loop_axis=None*, *entries=()*)
Bases: *pymia.data.transformation.Transform*, abc.ABC

static loop_entries(*sample: dict*, *fn*, *entries*, *loop_axis=None*)

abstract transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

class pymia.data.transformation.**Mask**(*mask_key: str*, *mask_value: int = 0*, *masking_value: float = 0.0*, *loop_axis=None*, *entries=('images', 'labels')*)
Bases: *pymia.data.transformation.Transform*

class pymia.data.transformation.**Permute**(*permutation: tuple*, *entries=('images', 'labels')*)
Bases: *pymia.data.transformation.LoopEntryTransform*

transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

class pymia.data.transformation.**RandomCrop**(*size: tuple*, *loop_axis=None*, *entries=('images', 'labels')*)
Bases: *pymia.data.transformation.LoopEntryTransform*

transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

class pymia.data.transformation.**Relabel**(*label_changes: Dict[int, int]*, *entries=('labels',)*)
Bases: *pymia.data.transformation.LoopEntryTransform*

transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

class pymia.data.transformation.**Reshape**(*shapes: dict*)
Bases: *pymia.data.transformation.LoopEntryTransform*

Initializes a new instance of the Reshape class.

Parameters **shapes** (*dict*) – A dict with keys being the entries and the values the new shapes of the entries. E.g. shapes = {defs.KEY_IMAGES: (-1, 4), defs.KEY_LABELS : (-1, 1)}

transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

class pymia.data.transformation.**SizeCorrection**(*shape: Tuple[Union[None, int], ...]*, *pad_value: int = 0*, *entries=('images', 'labels')*)
Bases: *pymia.data.transformation.Transform*

Size correction transformation.

Corrects the size, i.e. shape, of an array to a given reference shape.

Initializes a new instance of the SizeCorrection class.

Parameters

- **shape** (*tuple of ints*) – The reference shape in NumPy format, i.e. z-, y-, x-order. To not correct an axis dimension, set the axis value to None.
- **pad_value** (*int*) – The value to set the padded values of the array.
- **()** (*entries*) –

class pymia.data.transformation.**Squeeze**(*entries=('images', 'labels')*, *squeeze_axis=None*)
Bases: *pymia.data.transformation.LoopEntryTransform*

transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

class pymia.data.transformation.**Transform**
Bases: abc.ABC

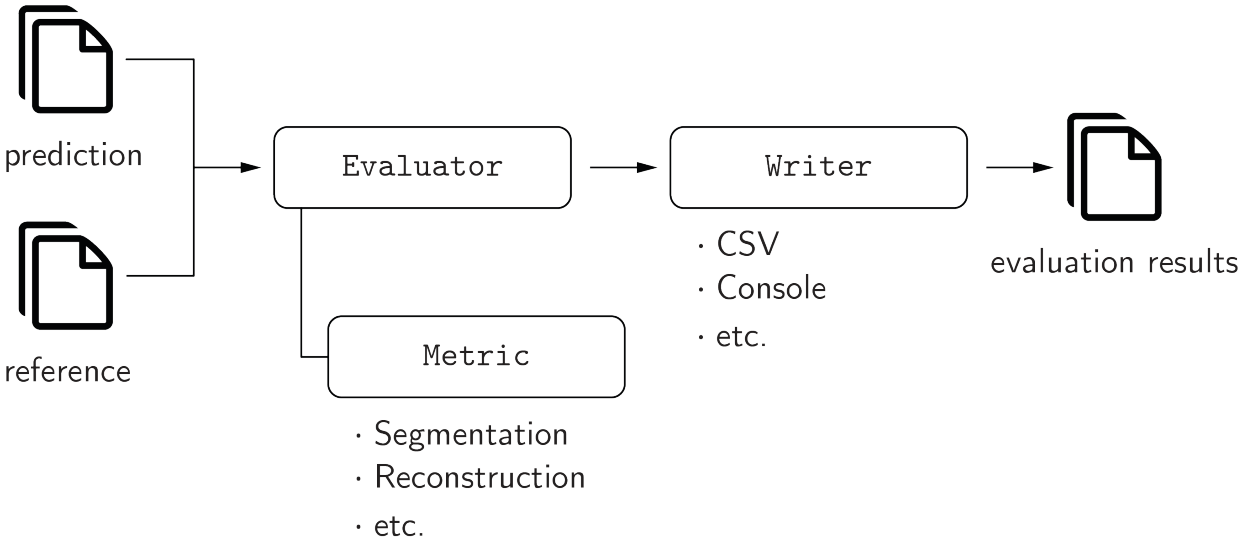
class pymia.data.transformation.**UnSqueeze**(*axis=- 1*, *entries=('images', 'labels')*)
Bases: *pymia.data.transformation.LoopEntryTransform*

transform_entry(*np_entry*, *entry*, *loop_i=None*) → numpy.ndarray

pymia.data.transformation.**check_and_return**(*obj*, *type_*)

2.2 Evaluation (pymia.evaluation package)

The evaluation package provides metrics and evaluation functionalities for image segmentation, image reconstruction, and regression. The concept of the evaluation package is illustrated in the figure below.



All metrics (*pymia.evaluation.metric.metric* package) implement the *pymia.evaluation.metric.base.Metric* interface, and can be used with the *pymia.evaluation.evaluator* package to evaluate results (e.g., with the *pymia.evaluation.evaluator.SegmentationEvaluator*). The *pymia.evaluation.writer* package provides several writers to report the results, and statistics of the results, to CSV files (e.g., the *pymia.evaluation.writer.CSVWriter* and *pymia.evaluation.writer.CSVStatisticsWriter*) and the console (e.g., the *pymia.evaluation.writer.ConsoleWriter* and *pymia.evaluation.writer.ConsoleStatisticsWriter*).

Refer to *Evaluation of results* for a code example on how to evaluate segmentation results. The code example *Logging the training progress* illustrates how to use the evaluation package to log results during the training of deep learning methods.

2.2.1 Subpackages

Metric (pymia.evaluation.metric package)

The metric package provides metrics for evaluation of image segmentation, image reconstruction, and regression.

All metrics implement the *pymia.evaluation.metric.base.Metric* interface, and can be used with the *pymia.evaluation.evaluator* package to evaluate results (e.g., with the *pymia.evaluation.evaluator.SegmentationEvaluator*). To implement your own metric and use it with the *pymia.evaluation.evaluator.Evaluator*, you need to inherit from *pymia.evaluation.metric.base.Metric*, *pymia.evaluation.metric.base.ConfusionMatrixMetric*, *pymia.evaluation.metric.base.DistanceMetric*, *pymia.evaluation.metric.base.NumpyArrayMetric*, or *pymia.evaluation.metric.base.SpacingMetric* and implement *pymia.evaluation.metric.base.Metric.calculate()*.

Note: The segmentation metrics are selected based on the paper by Taha and Hanbury. We recommend to refer to the paper for guidelines on how to select appropriate metrics, descriptions, and the math.

> Taha, A. A., & Hanbury, A. (2015). Metrics for evaluating 3D medical image segmentation: analysis, selection, and tool. BMC Medical Imaging, 15. https://doi.org/10.1186/s12880-015-0068-x

Base (pymia.evaluation.metric.base) module

The base module provides metric base classes.

class pymia.evaluation.metric.base.**ConfusionMatrix**(*prediction: numpy.ndarray*, *reference: numpy.ndarray*)

Bases: object

Represents a confusion matrix (or error matrix).

Parameters

- **prediction** (*np.ndarray*) – The prediction binary array.
- **reference** (*np.ndarray*) – The reference binary array.

class pymia.evaluation.metric.base.**ConfusionMatrixMetric**(*metric: str = 'ConfusionMatrixMetric'*)

Bases: *pymia.evaluation.metric.base.Metric*, abc.ABC

Represents a metric based on the confusion matrix.

Parameters **metric** (*str*) – The identification string of the metric.

class pymia.evaluation.metric.base.**DistanceMetric**(*metric: str = 'DistanceMetric'*)

Bases: *pymia.evaluation.metric.base.Metric*, abc.ABC

Represents a metric based on distances.

Parameters **metric** (*str*) – The identification string of the metric.

class pymia.evaluation.metric.base.**Distances**(*prediction: numpy.ndarray*, *reference: numpy.ndarray*, *spacing: tuple*)

Bases: object

Represents distances for distance metrics.

Parameters

- **prediction** (*np.ndarray*) – The prediction binary array.
- **reference** (*np.ndarray*) – The reference binary array.
- **spacing** (*tuple*) – The spacing in mm of each dimension.

See also:

- Nikolov, S., Blackwell, S., Mendes, R., De Fauw, J., Meyer, C., Hughes, C., … Ronneberger, O. (2018). Deep learning to achieve clinically applicable segmentation of head and neck anatomy for radiotherapy. http://arxiv.org/abs/1809.04430
- Original implementation

class pymia.evaluation.metric.base.**Information**(*column_name: str*, *value: str*)

Bases: *pymia.evaluation.metric.base.Metric*

Represents an information "metric".

Can be used to add an additional column of information to an evaluator.

Parameters

- **column_name** (*str*) – The identification string of the information.

- **value** (*str*) – The information.

calculate()
Outputs the value of the information.

class pymia.evaluation.metric.base.**Metric**(*metric: str = 'Metric'*)
Bases: abc.ABC

Metric base class.

Parameters **metric** (*str*) – The identification string of the metric.

abstract calculate()
Calculates the metric.

exception pymia.evaluation.metric.base.**NotComputableMetricWarning**
Bases: RuntimeWarning

Warning class to raise if a metric cannot be computed.

class pymia.evaluation.metric.base.**NumpyArrayMetric**(*metric: str = 'NumpyArrayMetric'*)
Bases: *pymia.evaluation.metric.base.Metric*, abc.ABC

Represents a metric based on numpy arrays.

Parameters **metric** (*str*) – The identification string of the metric.

class pymia.evaluation.metric.base.**SpacingMetric**(*metric: str = 'SpacingMetric'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*, abc.ABC

Represents a metric based on images with a physical spacing.

Parameters **metric** (*str*) – The identification string of the metric.

Metric (pymia.evaluation.metric.metric) module

The metric module provides a set of metrics.

pymia.evaluation.metric.metric.**get_classical_metrics**()
Gets a list of classical metrics.

Returns A list of metrics.

Return type list[*Metric*]

pymia.evaluation.metric.metric.**get_distance_metrics**()
Gets a list of distance-based metrics.

Returns A list of metrics.

Return type list[*Metric*]

pymia.evaluation.metric.metric.**get_overlap_metrics**()
Gets a list of overlap-based metrics.

Returns A list of metrics.

Return type list[*Metric*]

pymia.evaluation.metric.metric.**get_reconstruction_metrics**()
Gets a list with reconstruction metrics.

Returns A list of metrics.

Return type list[*Metric*]

pymia.evaluation.metric.metric.**get_regression_metrics**()
Gets a list with regression metrics.

Returns A list of metrics.

Return type list[*Metric*]

pymia.evaluation.metric.metric.**get_segmentation_metrics**()
Gets a list with segmentation metrics.

Returns A list of metrics.

Return type list[*Metric*]

Categorical metrics (pymia.evaluation.metric.categorical) module

The categorical module provides metrics to measure image segmentation performance.

class pymia.evaluation.metric.categorical.**Accuracy**(*metric: str = 'ACURCY'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents an accuracy metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the accuracy.

class pymia.evaluation.metric.categorical.**AdjustedRandIndex**(*metric: str = 'ADJRIND'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents an adjusted rand index metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the adjusted rand index.

class pymia.evaluation.metric.categorical.**AreaMetric**(*metric: str = 'AREA'*)
Bases: *pymia.evaluation.metric.base.SpacingMetric*, abc.ABC

Represents an area metric base class.

Parameters **metric** (*str*) – The identification string of the metric.

class pymia.evaluation.metric.categorical.**AreaUnderCurve**(*metric: str = 'AUC'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents an area under the curve metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the area under the curve.

class pymia.evaluation.metric.categorical.**AverageDistance**(*metric: str = 'AVGDIST'*)
Bases: *pymia.evaluation.metric.base.SpacingMetric*

Represents an average (Hausdorff) distance metric.

Calculates the distance between the set of non-zero pixels of two images using the following equation:

$$AVD(A, B) = max(d(A, B), d(B, A)),$$

where

$$d(A, B) = \frac{1}{N} \sum_{a \in A} \min_{b \in B} \|a - b\|$$

is the directed Hausdorff distance and A and B are the set of non-zero pixels in the images.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the average (Hausdorff) distance.

class pymia.evaluation.metric.categorical.**CohenKappaCoefficient**(*metric: str = 'KAPPA'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a Cohen's kappa coefficient metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the Cohen's kappa coefficient.

class pymia.evaluation.metric.categorical.**DiceCoefficient**(*metric: str = 'DICE'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a Dice coefficient metric with empty target handling, defined as:

$$\begin{cases} 1 & |y| = |\hat{y}| = 0 \\ Dice(y, \hat{y}) & |y| > 0 \end{cases}$$

where $\hat{y}$ is the prediction and y the target.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the Dice coefficient.

class pymia.evaluation.metric.categorical.**FMeasure**(*beta: float = 1.0, metric: str = 'FMEASR'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a F-measure metric.

Parameters

- **beta** (*float*) – The beta to trade-off precision and recall. Use 0.5 or 2 to calculate the F0.5 and F2 measure, respectively.
- **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the F1 measure.

class pymia.evaluation.metric.categorical.**Fallout**(*metric: str = 'FALLOUT'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a fallout (false positive rate) metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the fallout (false positive rate).

class pymia.evaluation.metric.categorical.**FalseNegative**(*metric: str = 'FN'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a false negative metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the false negatives.

class pymia.evaluation.metric.categorical.**FalseNegativeRate**(*metric: str = 'FNR'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a false negative rate metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the false negative rate.

class pymia.evaluation.metric.categorical.**FalsePositive**(*metric: str = 'FP'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a false positive metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the false positives.

class pymia.evaluation.metric.categorical.**GlobalConsistencyError**(*metric: str = 'GCOERR'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a global consistency error metric.

Implementation based on Martin 2001. todo(fabianbalsiger): add entire reference

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the global consistency error.

class pymia.evaluation.metric.categorical.**HausdorffDistance**(*percentile: float = 100.0*, *metric: str = 'HDRFDST'*)
Bases: *pymia.evaluation.metric.base.DistanceMetric*

Represents a Hausdorff distance metric.

Calculates the distance between the set of non-zero pixels of two images using the following equation:

$$H(A, B) = max(h(A, B), h(B, A)),$$

where

$$h(A, B) = \max_{a \in A} \min_{b \in B} \|a - b\|$$

is the directed Hausdorff distance and A and B are the set of non-zero pixels in the images.

Parameters

- **percentile** (*float*) – The percentile (0, 100] to compute, i.e. 100 computes the Hausdorff distance and 95 computes the 95th Hausdorff distance.
- **metric** (*str*) – The identification string of the metric.

See also:

- Nikolov, S., Blackwell, S., Mendes, R., De Fauw, J., Meyer, C., Hughes, C., … Ronneberger, O. (2018). Deep learning to achieve clinically applicable segmentation of head and neck anatomy for radiotherapy. http://arxiv.org/abs/1809.04430

- Original implementation

calculate()
Calculates the Hausdorff distance.

class pymia.evaluation.metric.categorical.**InterclassCorrelation**(*metric: str = 'ICCORR'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*

Represents an interclass correlation metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the interclass correlation.

class pymia.evaluation.metric.categorical.**JaccardCoefficient**(*metric: str = 'JACRD'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a Jaccard coefficient metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the Jaccard coefficient.

class pymia.evaluation.metric.categorical.**MahalanobisDistance**(*metric: str = 'MAHLNBS'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*

Represents a Mahalanobis distance metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the Mahalanobis distance.

class pymia.evaluation.metric.categorical.**MutualInformation**(*metric: str = 'MUTINF'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a mutual information metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the mutual information.

class pymia.evaluation.metric.categorical.**Precision**(*metric: str = 'PRCISON'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a precision metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the precision.

class pymia.evaluation.metric.categorical.**PredictionArea**(*slice_number: int = - 1*, *metric: str = 'PREDAREA'*)
Bases: *pymia.evaluation.metric.categorical.AreaMetric*

Represents a prediction area metric.

Parameters

- **slice_number** (*int*) – The slice number to calculate the area. Defaults to -1, which will calculate the area on the intermediate slice.

- **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the predicted area on a specified slice in mm2.

class pymia.evaluation.metric.categorical.**PredictionVolume**(*metric: str = 'PREDVOL'*)
Bases: *pymia.evaluation.metric.categorical.VolumeMetric*

Represents a prediction volume metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the predicted volume in mm3.

class pymia.evaluation.metric.categorical.**ProbabilisticDistance**(*metric: str = 'PROBDST'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*

Represents a probabilistic distance metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the probabilistic distance.

class pymia.evaluation.metric.categorical.**RandIndex**(*metric: str = 'RNDIND'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a rand index metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the rand index.

class pymia.evaluation.metric.categorical.**ReferenceArea**(*slice_number: int = - 1*, *metric: str = 'REFAREA'*)
Bases: *pymia.evaluation.metric.categorical.AreaMetric*

Represents a reference area metric.

Parameters

- **slice_number** (*int*) – The slice number to calculate the area. Defaults to -1, which will calculate the area on the intermediate slice.
- **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the reference area on a specified slice in mm2.

class pymia.evaluation.metric.categorical.**ReferenceVolume**(*metric: str = 'REFVOL'*)
Bases: *pymia.evaluation.metric.categorical.VolumeMetric*

Represents a reference volume metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the reference volume in mm3.

class pymia.evaluation.metric.categorical.**Sensitivity**(*metric: str = 'SNSVTY'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a sensitivity (true positive rate or recall) metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the sensitivity (true positive rate).

class pymia.evaluation.metric.categorical.**Specificity**(*metric: str = 'SPCFTY'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a specificity metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the specificity.

class pymia.evaluation.metric.categorical.**SurfaceDiceOverlap**(*tolerance: float = 1*, *metric: str = 'SURFDICE'*)
Bases: *pymia.evaluation.metric.base.DistanceMetric*

Represents a surface Dice coefficient overlap metric.

Parameters

- **tolerance** (*float*) – The tolerance of the surface distance in mm.
- **metric** (*str*) – The identification string of the metric.

See also:

- Nikolov, S., Blackwell, S., Mendes, R., De Fauw, J., Meyer, C., Hughes, C., … Ronneberger, O. (2018). Deep learning to achieve clinically applicable segmentation of head and neck anatomy for radiotherapy. http://arxiv.org/abs/1809.04430
- Original implementation

calculate()
Calculates the surface Dice coefficient overlap.

class pymia.evaluation.metric.categorical.**SurfaceOverlap**(*tolerance: float = 1.0*, *prediction_to_reference: bool = True*, *metric: str = 'SURFOVLP'*)
Bases: *pymia.evaluation.metric.base.DistanceMetric*

Represents a surface overlap metric.

Computes the overlap of the reference surface with the predicted surface and vice versa allowing a specified tolerance (maximum surface-to-surface distance that is regarded as overlapping). The overlapping fraction is computed by correctly taking the area of each surface element into account.

Parameters

- **tolerance** (*float*) – The tolerance of the surface distance in mm.
- **prediction_to_reference** (*bool*) – Computes the prediction to reference if *True*, otherwise the reference to prediction.
- **metric** (*str*) – The identification string of the metric.

See also:

- Nikolov, S., Blackwell, S., Mendes, R., De Fauw, J., Meyer, C., Hughes, C., … Ronneberger, O. (2018). Deep learning to achieve clinically applicable segmentation of head and neck anatomy for radiotherapy. http://arxiv.org/abs/1809.04430
- Original implementation

calculate()
Calculates the surface overlap.

class pymia.evaluation.metric.categorical.**TrueNegative**(*metric: str = 'TN'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a true negative metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the true negatives.

class pymia.evaluation.metric.categorical.**TruePositive**(*metric: str = 'TP'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a true positive metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the true positives.

class pymia.evaluation.metric.categorical.**VariationOfInformation**(*metric: str = 'VARINFO'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a variation of information metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the variation of information.

class pymia.evaluation.metric.categorical.**VolumeMetric**(*metric: str = 'VOL'*)
Bases: *pymia.evaluation.metric.base.SpacingMetric*, abc.ABC

Represents a volume metric base class.

Parameters **metric** (*str*) – The identification string of the metric.

class pymia.evaluation.metric.categorical.**VolumeSimilarity**(*metric: str = 'VOLSMTY'*)
Bases: *pymia.evaluation.metric.base.ConfusionMatrixMetric*

Represents a volume similarity metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the volume similarity.

Continuous metrics (pymia.evaluation.metric.continuous) module

The continuous module provides metrics to measure image reconstruction and regression performance.

class pymia.evaluation.metric.continuous.**CoefficientOfDetermination**(*metric: str = 'R2'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*

Represents a coefficient of determination (R^2) error metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the coefficient of determination (R^2) error.

See also:

https://stackoverflow.com/a/45538060

class pymia.evaluation.metric.continuous.**MeanAbsoluteError**(*metric: str = 'MAE'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*

Represents a mean absolute error metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the mean absolute error.

class pymia.evaluation.metric.continuous.**MeanSquaredError**(*metric: str = 'MSE'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*

Represents a mean squared error metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the mean squared error.

class pymia.evaluation.metric.continuous.**NormalizedRootMeanSquaredError**(*metric: str = 'NRMSE'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*

Represents a normalized root mean squared error metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the normalized root mean squared error.

class pymia.evaluation.metric.continuous.**PeakSignalToNoiseRatio**(*metric: str = 'PSNR'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*

Represents a peak signal to noise ratio metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the peak signal to noise ratio.

class pymia.evaluation.metric.continuous.**RootMeanSquaredError**(*metric: str = 'RMSE'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*

Represents a root mean squared error metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the root mean squared error.

class pymia.evaluation.metric.continuous.**StructuralSimilarityIndexMeasure**(*metric: str = 'SSIM'*)
Bases: *pymia.evaluation.metric.base.NumpyArrayMetric*

Represents a structural similarity index measure metric.

Parameters **metric** (*str*) – The identification string of the metric.

calculate()
Calculates the structural similarity index measure.

2.2.2 The evaluator module (pymia.evaluation.evaluator)

The evaluator module provides classes to evaluate the metrics on predictions.

All evaluators inherit the *pymia.evaluation.evaluator.Evaluator*, which contains a list of results after calling *pymia.evaluation.evaluator.Evaluator.evaluate()*. The results can be passed to a writer of the *pymia.evaluation.writer* module.

class pymia.evaluation.evaluator.**Evaluator**(*metrics: List[*pymia.evaluation.metric.base.Metric*]*)

Bases: abc.ABC

Evaluator base class.

Parameters **metrics** (*list of pymia_metric.Metric*) – A list of metrics.

clear()

Clears the results.

abstract evaluate(*prediction: Union[SimpleITK.SimpleITK.Image, numpy.ndarray]*, *reference: Union[SimpleITK.SimpleITK.Image, numpy.ndarray]*, *id_: str*, ***kwargs*)

Evaluates the metrics on the provided prediction and reference.

Parameters

- **prediction** (*typing.Union[sitk.Image, np.ndarray]*) – The prediction.
- **reference** (*typing.Union[sitk.Image, np.ndarray]*) – The reference.
- **id** (*str*) – The identification of the case to evaluate.

class pymia.evaluation.evaluator.**Result**(*id_: str*, *label: str*, *metric: str*, *value*)

Bases: object

Represents a result.

Parameters

- **id** (*str*) – The identification of the result (e.g., the subject's name).
- **label** (*str*) – The label of the result (e.g., the foreground).
- **metric** (*str*) – The metric.
- **value** (*int, float*) – The value of the metric.

class pymia.evaluation.evaluator.**SegmentationEvaluator**(*metrics: List[*pymia.evaluation.metric.base.Metric*]*, *labels: dict*)

Bases: *pymia.evaluation.evaluator.Evaluator*

Represents a segmentation evaluator, evaluating metrics on predictions against references.

Parameters

- **metrics** (*list of pymia_metric.Metric*) – A list of metrics.
- **labels** (*dict*) – A dictionary with labels (key of type int) and label descriptions (value of type string).

add_label(*label: Union[tuple, int]*, *description: str*)

Adds a label with its description to the evaluation.

Parameters

- **label** (*Union[tuple, int]*) – The label or a tuple of labels that should be merged.
- **description** (*str*) – The label's description.

evaluate(*prediction: Union[SimpleITK.SimpleITK.Image, numpy.ndarray]*, *reference: Union[SimpleITK.SimpleITK.Image, numpy.ndarray]*, *id_: str*, ***kwargs*)
Evaluates the metrics on the provided prediction and reference image.

Parameters

- **prediction** (*typing.Union[sitk.Image, np.ndarray]*) – The predicted image.
- **reference** (*typing.Union[sitk.Image, np.ndarray]*) – The reference image.
- **id** (*str*) – The identification of the case to evaluate.

Raises **ValueError** – If no labels are defined (see add_label).

2.2.3 The writer module (pymia.evaluation.writer)

The writer module provides classes to write evaluation results.

All writers inherit the *pymia.evaluation.writer.Writer*, which writes the results when calling *pymia.evaluation.writer.Writer.write()*. Currently, pymia has CSV file (*pymia.evaluation.writer.CSVWriter* and *pymia.evaluation.writer.CSVStatisticsWriter*) and console writers (*pymia.evaluation.writer.ConsoleWriter* and *pymia.evaluation.writer.ConsoleStatisticsWriter*).

class pymia.evaluation.writer.**CSVStatisticsWriter**(*path: str*, *delimiter: str = ';'*, *functions: Optional[dict] = None*)
Bases: *pymia.evaluation.writer.Writer*

Represents a CSV file evaluation results statistics writer.

Parameters

- **path** (*str*) – The CSV file path.
- **delimiter** (*str*) – The CSV column delimiter.
- **functions** (*dict*) – The functions to calculate the statistics.

write(*results: List[*pymia.evaluation.evaluator.Result*]*, ***kwargs*)
Writes the evaluation statistic results (e.g., mean and standard deviation of a metric over all cases).

Parameters **results** (*typing.List[evaluator.Result]*) – The evaluation results.

class pymia.evaluation.writer.**CSVWriter**(*path: str*, *delimiter: str = ';'*)
Bases: *pymia.evaluation.writer.Writer*

Represents a CSV file evaluation results writer.

Parameters

- **path** (*str*) – The CSV file path.
- **delimiter** (*str*) – The CSV column delimiter.

write(*results: List[*pymia.evaluation.evaluator.Result*]*, ***kwargs*)
Writes the evaluation results to a CSV file.

Parameters **results** (*typing.List[evaluator.Result]*) – The evaluation results.

class pymia.evaluation.writer.**ConsoleStatisticsWriter**(*precision: int = 3*, *use_logging: bool = False*, *functions: Optional[dict] = None*)
Bases: *pymia.evaluation.writer.Writer*

Represents a console evaluation results statistics writer.

Parameters

- **precision** (*int*) – The float precision.
- **use_logging** (*bool*) – Indicates whether to use the Python logging module or not.
- **functions** (*dict*) – The function handles to calculate the statistics.

write(*results: List[*pymia.evaluation.evaluator.Result*]*, ***kwargs*)
Writes the evaluation statistic results (e.g., mean and standard deviation of a metric over all cases).

Parameters **results** (*typing.List[*evaluator.Result*]*) – The evaluation results.

class pymia.evaluation.writer.**ConsoleWriter**(*precision: int = 3*, *use_logging: bool = False*)
Bases: *pymia.evaluation.writer.Writer*

Represents a console evaluation results writer.

Parameters

- **precision** (*int*) – The decimal precision.
- **use_logging** (*bool*) – Indicates whether to use the Python logging module or not.

write(*results: List[*pymia.evaluation.evaluator.Result*]*, ***kwargs*)
Writes the evaluation results.

Parameters **results** (*typing.List[*evaluator.Result*]*) – The evaluation results.

class pymia.evaluation.writer.**ConsoleWriterHelper**(*use_logging: bool = False*)
Bases: object

Represents a console writer helper.

Parameters **use_logging** (*bool*) – Indicates whether to use the Python logging module or not.

format_and_write(*lines: list*)
Formats and writes the results.

Parameters **lines** (*list of lists*) – The lines to write. Each line is a list of columns.

class pymia.evaluation.writer.**StatisticsAggregator**(*functions: Optional[dict] = None*)
Bases: object

Represents a statistics evaluation results aggregator.

Parameters **functions** (*dict*) – The numpy function handles to calculate the statistics.

calculate(*results: List[*pymia.evaluation.evaluator.Result*]*) → List[*pymia.evaluation.evaluator.Result*]
Calculates aggregated results (e.g., mean and standard deviation of a metric over all cases).

Parameters **results** (*typing.List[*evaluator.Result*]*) – The results to aggregate.

Returns The aggregated results.

Return type typing.List[*evaluator.Result*]

class pymia.evaluation.writer.**Writer**
Bases: abc.ABC

Represents an evaluation results writer base class.

abstract write(*results: List[*pymia.evaluation.evaluator.Result*]*, ***kwargs*)
Writes the evaluation results.

Parameters **results** (*list of evaluator.Result*) – The evaluation results.

2.3 Filtering (pymia.filtering package)

The filtering package provides basic image filter and manipulation functions.

All filters in the *pymia.filtering* package implement the *pymia.filtering.filter.Filter* interface, and can be used to set up a pipeline with the *pymia.filtering.filter.FilterPipeline*. Refer to *Filter pipelines* for a code example.

2.3.1 Filter pipeline (pymia.filtering.filter module)

This module provides classes to set up a filtering pipeline.

class pymia.filtering.filter.**Filter**
Bases: abc.ABC

Filter base class.

abstract execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[*pymia.filtering.filter.FilterParams*] = None*) → SimpleITK.SimpleITK.Image
Executes a filter on an image.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (FilterParams) – The filter parameters.

Returns The filtered image.

Return type sitk.Image

class pymia.filtering.filter.**FilterParams**
Bases: abc.ABC

Represents a filter parameters interface.

class pymia.filtering.filter.**FilterPipeline**(*filters: Optional[List[*pymia.filtering.filter.Filter*]] = None*)
Bases: object

Represents a filter pipeline, which sequentially executes filters (*Filter*) on an image.

Parameters **filters** (*list of Filter*) – The filters of the pipeline.

add_filter(*filter_:* pymia.filtering.filter.Filter, *params: Optional[*pymia.filtering.filter.FilterParams*] = None*)
Adds a filter to the pipeline.

Parameters

- **filter** (Filter) – A filter.
- **params** (FilterParams) – The filter parameters.

execute(*image: SimpleITK.SimpleITK.Image*) → SimpleITK.SimpleITK.Image
Executes the filter pipeline on an image.

Parameters **image** (*sitk.Image*) – The image to filter.

Returns The filtered image.

Return type sitk.Image

set_param(*params:* pymia.filtering.filter.FilterParams, *filter_index: int*)
Sets an image-specific parameter for a filter.

Use this function to update the parameters of a filter to be specific to the image to be filtered.

Parameters

- **params** (FilterParams) – The parameter(s).
- **filter_index** (*int*) – The filter's index the parameters belong to.

2.3.2 Miscellaneous (pymia.filtering.misc module)

The misc (miscellaneous) module provides filters, which don't have a classical purpose.

class pymia.filtering.misc.**CmdlineExecutor**(*executable_path: str*)
Bases: *pymia.filtering.filter.Filter*

Represents a command line executable.

Use this filter to execute for instance a C++ command line program, which loads and image, processes, and saves it.

Parameters **executable_path** (*str*) – The path to the executable to run.

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[*pymia.filtering.misc.CmdlineExecutorParams*] = None*) → SimpleITK.SimpleITK.Image
Executes a command line program.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (CmdlineExecutorParams) – The execution specific command line parameters.

Returns The filtered image.

Return type sitk.Image

class pymia.filtering.misc.**CmdlineExecutorParams**(*arguments: List[str]*)
Bases: *pymia.filtering.filter.FilterParams*

Command line executor filter parameters used by the *CmdlineExecutor* filter.

Parameters **arguments** (*typing.List[str]*) – Additional arguments for the command line execution.

class pymia.filtering.misc.**Relabel**(*label_changes: Dict[int, Union[int, tuple]]*)
Bases: *pymia.filtering.filter.Filter*

Represents a relabel filter.

Parameters **label_changes** (*typing.Dict[int, typing.Union[int, tuple]]*) – Label change rule where the key is the new label and the value the existing (can be multiple) label.

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[*pymia.filtering.filter.FilterParams*] = None*) → SimpleITK.SimpleITK.Image
Executes the relabeling of the label image.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (FilterParams) – The filter parameters (unused).

Returns The filtered image.

Return type sitk.Image

class pymia.filtering.misc.**SizeCorrection**(*two_sided: bool = True*, *pad_constant: float = 0.0*)
Bases: *pymia.filtering.filter.Filter*

Represents a filter to correct the shape/size by padding or cropping.

Parameters

- **two_sided** (*bool*) – Indicates whether the cropping and padding should be applied on one or both side(s) of the dimension.
- **pad_constant** (*float*) – The constant value used for padding.

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[*pymia.filtering.misc.SizeCorrectionParams*] = None*) → SimpleITK.SimpleITK.Image
Executes the shape/size correction by padding or cropping.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (SizeCorrectionParams) – The filter parameters containing the reference (target) shape.

Returns The filtered image.

Return type sitk.Image

class pymia.filtering.misc.**SizeCorrectionParams**(*reference_shape: tuple*)
Bases: *pymia.filtering.filter.FilterParams*

Represents size (shape) correction filter parameters used by the *SizeCorrection* filter.

Parameters **reference_shape** (*tuple*) – The reference or target shape.

2.3.3 Post-processing (pymia.filtering.postprocessing module)

The post-processing module provides filters for image post-processing.

class pymia.filtering.postprocessing.**BinaryThreshold**(*threshold: float*)
Bases: *pymia.filtering.filter.Filter*

Represents a binary threshold image filter.

Parameters **threshold** (*float*) – The threshold value.

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[*pymia.filtering.filter.FilterParams*] = None*) → SimpleITK.SimpleITK.Image
Executes the binary threshold filter on an image.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (FilterParams) – The filter parameters (unused).

Returns The filtered image.

Return type sitk.Image

class pymia.filtering.postprocessing.**LargestNConnectedComponents**(*number_of_components: int = 1*, *consecutive_component_labels: bool = False*)

Bases: *pymia.filtering.filter.Filter*

Represents a largest N connected components filter.

Extracts the largest N connected components from a label image. By default the N components will all have the value 1 in the output image. Use the *consecutive_component_labels* option such that the largest has value 1, the second largest has value 2, etc. Background is always assumed to be 0.

Parameters

- **number_of_components** (*int*) – The number of largest components to extract.
- **consecutive_component_labels** (*bool*) – The largest component has value 1, the second largest has value 2, ect. if set to True; otherwise, all components will have value 1.

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[*pymia.filtering.filter.FilterParams*] = None*) → SimpleITK.SimpleITK.Image

Executes the largest N connected components filter on an image.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (FilterParams) – The filter parameters (unused).

Returns The filtered image.

Return type sitk.Image

2.3.4 Pre-processing (pymia.filtering.preprocessing module)

The pre-processing module provides filters for image pre-processing.

class pymia.filtering.preprocessing.**BiasFieldCorrector**(*convergence_threshold: float = 0.001*, *max_iterations: List[int] = (50, 50, 50, 50)*, *fullwidth_at_halfmax: float = 0.15*, *filter_noise: float = 0.01*, *histogram_bins: int = 200*, *control_points: List[int] = (4, 4, 4)*, *spline_order: int = 3*)

Bases: *pymia.filtering.filter.Filter*

Represents a bias field correction filter.

Parameters

- **convergence_threshold** (*float*) – The threshold to stop the optimizer.
- **max_iterations** (*typing.List[int]*) – The maximum number of optimizer iterations at each level.
- **fullwidth_at_halfmax** (*float*) – The full width at half maximum.
- **filter_noise** (*float*) – Wiener filter noise.
- **histogram_bins** (*int*) – Number of histogram bins.
- **control_points** (*typing.List[int]*) – The number of spline control points.
- **spline_order** (*int*) – The spline order.

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[pymia.filtering.preprocessing.BiasFieldCorrectorParams] = None*) → SimpleITK.SimpleITK.Image

Executes a bias field correction on an image.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (BiasFieldCorrectorParams) – The bias field correction filter parameters.

Returns The bias field corrected image.

Return type sitk.Image

class pymia.filtering.preprocessing.**BiasFieldCorrectorParams**(*mask: SimpleITK.SimpleITK.Image*)

Bases: *pymia.filtering.filter.FilterParams*

Bias field correction filter parameters used by the *BiasFieldCorrector* filter.

Parameters **mask** (*sitk.Image*) – A mask image (0=background; 1=mask).

Examples

To generate a default mask use Otsu's thresholding:

```
>>> sitk.OtsuThreshold(image, 0, 1, 200)
```

class pymia.filtering.preprocessing.**GradientAnisotropicDiffusion**(*time_step: float = 0.125*, *conductance: int = 3*, *conductance_scaling_update_interval: int = 1*, *no_iterations: int = 5*)

Bases: *pymia.filtering.filter.Filter*

Represents a gradient anisotropic diffusion filter.

Parameters

- **time_step** (*float*) – The time step.
- **conductance** (*int*) – The conductance (the higher the smoother the edges).
- **conductance_scaling_update_interval** – TODO
- **no_iterations** (*int*) – Number of iterations.

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[pymia.filtering.filter.FilterParams] = None*) → SimpleITK.SimpleITK.Image

Executes a gradient anisotropic diffusion on an image.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (FilterParams) – The parameters (unused).

Returns The smoothed image.

Return type sitk.Image

class pymia.filtering.preprocessing.**HistogramMatcher**(*histogram_levels: int = 256*, *match_points: int = 1*, *threshold_mean_intensity: bool = True*)

Bases: *pymia.filtering.filter.Filter*

Represents a histogram matching filter.

Parameters

- **histogram_levels** (*int*) – Number of histogram levels.
- **match_points** (*int*) – Number of match points.
- **threshold_mean_intensity** (*bool*) – Threshold at mean intensity.

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[pymia.filtering.preprocessing.HistogramMatcherParams] = None*) → SimpleITK.SimpleITK.Image

Matches the image intensity histogram to a reference.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (HistogramMatcherParams) – The filter parameters.

Returns The filtered image.

Return type sitk.Image

class pymia.filtering.preprocessing.**HistogramMatcherParams**(*reference_image: SimpleITK.SimpleITK.Image*)

Bases: *pymia.filtering.filter.FilterParams*

Histogram matching filter parameters used by the *HistogramMatcher* filter.

Parameters **reference_image** (*sitk.Image*) – Reference image for the matching.

class pymia.filtering.preprocessing.**NormalizeZScore**

Bases: *pymia.filtering.filter.Filter*

Represents a z-score normalization filter.

Filter base class.

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[pymia.filtering.filter.FilterParams] = None*) → SimpleITK.SimpleITK.Image

Executes a z-score normalization on an image.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (FilterParams) – The parameters (unused).

Returns The normalized image.

Return type sitk.Image

class pymia.filtering.preprocessing.**RescaleIntensity**(*min_intensity: float*, *max_intensity: float*)

Bases: *pymia.filtering.filter.Filter*

Represents a rescale intensity filter.

Parameters

- **min_intensity** (*float*) – The min intensity value.
- **max_intensity** (*float*) – The max intensity value.

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[pymia.filtering.filter.FilterParams] = None*) → SimpleITK.SimpleITK.Image

Executes an intensity rescaling on an image.

Parameters

- **image** (*sitk.Image*) – The image to filter.
- **params** (FilterParams) – The parameters (unused).

Returns The intensity rescaled image.

Return type sitk.Image

2.3.5 Registration (pymia.filtering.registration module)

The registration module provides classes for image registration.

See also:

- ITK Registration
- ITK Software Guide Registration

class pymia.filtering.registration.**MultiModalRegistration**(*registration_type: pymia.filtering.registration.RegistrationType = RegistrationType.RIGID*, *number_of_histogram_bins: int = 200*, *learning_rate: float = 1.0*, *step_size: float = 0.001*, *number_of_iterations: int = 200*, *relaxation_factor: float = 0.5*, *shrink_factors: List[int] = (2, 1, 1)*, *smoothing_sigmas: List[float] = (2, 1, 0)*, *sampling_percentage: float = 0.2*, *sampling_seed: int = 0*, *resampling_interpolator=3*)

Bases: *pymia.filtering.filter.Filter*

Represents a multi-modal image registration filter.

The filter estimates a 3-dimensional rigid or affine transformation between images of different modalities using - Mutual information similarity metric - Linear interpolation - Gradient descent optimization

Parameters

- **registration_type** (RegistrationType) – The type of the registration ('rigid' or 'affine').
- **number_of_histogram_bins** (*int*) – The number of histogram bins.
- **learning_rate** (*float*) – The optimizer's learning rate.
- **step_size** (*float*) – The optimizer's step size. Each step in the optimizer is at least this large.
- **number_of_iterations** (*int*) – The maximum number of optimization iterations.
- **relaxation_factor** (*float*) – The relaxation factor to penalize abrupt changes during optimization.
- **shrink_factors** (*typing.List[int]*) – The shrink factors at each shrinking level (from high to low).
- **smoothing_sigmas** (*typing.List[int]*) – The Gaussian sigmas for smoothing at each shrinking level (in physical units).

- **sampling_percentage** (*float*) – Fraction of voxel of the fixed image that will be used for registration (0, 1]. Typical values range from 0.01 (1 %) for low detail images to 0.2 (20 %) for high detail images. The higher the fraction, the higher the computational time.
- **sampling_seed** – The seed for reproducible behavior.
- **resampling_interpolator** – Interpolation to be applied while resampling the image by the determined transformation.

Examples

The following example shows the usage of the MultiModalRegistration class.

```
>>> fixed_image = sitk.ReadImage('/path/to/image/fixed.mha')
>>> moving_image = sitk.ReadImage('/path/to/image/moving.mha')
>>> registration = MultiModalRegistration()  # specify parameters to your needs
>>> parameters = MultiModalRegistrationParams(fixed_image)
>>> registered_image = registration.execute(moving_image, parameters)
```

execute(*image: SimpleITK.SimpleITK.Image*, *params: Optional[*pymia.filtering.registration.MultiModalRegistrationParams*] = None*) → SimpleITK.SimpleITK.Image

Executes a multi-modal rigid registration.

Parameters

- **image** (*sitk.Image*) – The moving image to register.
- **params** (MultiModalRegistrationParams) – The parameters, which contain the fixed image.

Returns The registered image.

Return type sitk.Image

class pymia.filtering.registration.**MultiModalRegistrationParams**(*fixed_image: SimpleITK.SimpleITK.Image*, *fixed_image_mask: Optional[SimpleITK.SimpleITK.Image] = None*, *callbacks: Optional[List[*pymia.filtering.registration.RegistrationC *= None*)

Bases: *pymia.filtering.filter.FilterParams*

Represents parameters for the multi-modal rigid registration used by the *MultiModalRegistration* filter.

Parameters

- **fixed_image** (*sitk.Image*) – The fixed image for the registration.
- **fixed_image_mask** (*sitk.Image*) – A mask for the fixed image to limit the registration.
- **callbacks** (*t.List[*RegistrationCallback*]*) – Path to the directory where to plot the registration progress if any. Note that this increases the computational time.

class pymia.filtering.registration.**PlotOnResolutionChangeCallback**(*plot_dir: str*, *file_name_prefix: str = ''*)

Bases: *pymia.filtering.registration.RegistrationCallback*

Represents a plotter for registrations.

Saves the moving image on each resolution change and the registration end.

Parameters

- **plot_dir** (*str*) – Path to the directory where to save the plots.
- **file_name_prefix** (*str*) – The file name prefix for the plots.

registration_ended()
Callback for the EndEvent.

registration_iteration_ended()
Callback for the IterationEvent.

registration_resolution_changed()
Callback for the MultiResolutionIterationEvent.

registration_started()
Callback for the StartEvent.

class pymia.filtering.registration.**RegistrationCallback**
Bases: abc.ABC

Represents the abstract handler for the registration callbacks.

registration_ended()
Callback for the EndEvent.

registration_iteration_ended()
Callback for the IterationEvent.

registration_resolution_changed()
Callback for the MultiResolutionIterationEvent.

registration_started()
Callback for the StartEvent.

set_params(*registration_method: SimpleITK.SimpleITK.ImageRegistrationMethod*, *fixed_image: SimpleITK.SimpleITK.Image*, *moving_image: SimpleITK.SimpleITK.Image*, *transform: SimpleITK.SimpleITK.Transform*)
Sets the parameters that might be used during the callbacks

Parameters

- **registration_method** (*sitk.ImageRegistrationMethod*) – The registration method.
- **fixed_image** (*sitk.Image*) – The fixed image.
- **moving_image** (*sitk.Image*) – The moving image.
- **transform** (*sitk.Transform*) – The transformation.

class pymia.filtering.registration.**RegistrationType**(*value*)
Bases: enum.Enum

Represents the registration transformation type.

AFFINE = 1

BSPLINE = 4

RIGID = 3

SIMILARITY = 2

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

[VanEssen2013] Van Essen, D. C., Smith, S. M., Barch, D. M., Behrens, T. E. J., Yacoub, E., Ugurbil, K., & WU-Minn HCP Consortium. (2013). The WU-Minn Human Connectome Project: An overview. NeuroImage, 80, 62–79. https://doi.org/10.1016/j.neuroimage.2013.05.041

[Fischl2012] Fischl, B. (2012). FreeSurfer. NeuroImage, 62(2), 774–781. https://doi.org/10.1016/j.neuroimage.2012.01.021

[Fischl2002] Fischl, B., Salat, D. H., Busa, E., Albert, M., Dieterich, M., Haselgrove, C., … Dale, A. M. (2002). Whole brain segmentation: Automated labeling of neuroanatomical structures in the human brain. Neuron, 33(3), 341–355. https://doi.org/10.1016/S0896-6273(02)00569-X

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W