bmmltools

Release 0.2

Curcuraci Luca

Nov 11, 2022

CONTENTS

1 How to install **1**

2 Data and Trace **3**
2.1 Data . . . 3
2.2 Trace . . . 5

3 Operations in general **9**
3.1 Example of operation usage . . . 11

4 Reference API **13**

5 Io **15**
5.1 Input . . . 15
5.2 InputFromTrace . . . 16
5.3 OutputRawLabels . . . 16
5.4 OutputValidLabels . . . 17
5.5 OutputSegmentation . . . 19

6 Feature **21**
6.1 Binarizer . . . 21
6.2 Patch Transform 3D . . . 22
6.3 Patch Discrete Fourier Transform 3D . . . 23
6.4 Dimensional reduction . . . 25
6.5 Data standardization . . . 26

7 Clustering **29**
7.1 Clusterer . . . 29
7.2 Clusterer HDBSCAN . . . 30
7.3 ClusterValidator . . . 32
7.4 ArchetypeIdentifier . . . 34
7.5 RotationalSimilarityIdentifier . . . 35

8 Segmentation **39**
8.1 RandomForestSegmenter . . . 39

9 Explanation **43**
9.1 MultiCollinearityReducer . . . 43
9.2 ExplainWithClassifier . . . 44
9.3 InterpretPIandPD . . . 46

10 bmmlboard **49**

10.1 Label Visualizer 49
10.2 Segmentation Visualizer 49
10.3 Explainer Visualizer 50

11 Case of study: segmentation pipeline **51**
11.1 Goals 51
11.2 General structure of the pipeline 51
11.3 The segmentation pipeline in detail 53
11.4 Example 56
11.5 Reference 63

12 Case of study: identification for similarity under rotation **65**
12.1 Archetypes 65
12.2 Rotational Similarity measured with a statistical test 66
12.3 Example 69

13 Case of study: explanation pipeline **75**
13.1 General strategy 75
13.2 Dataset and its preparation 76
13.3 Tools for the explanation 77
13.4 Interpreting F1 scores, PI and PD 80
13.5 Example 83
13.6 References 88

14 Indices and tables **89**

Bibliography **91**

CHAPTER

ONE

HOW TO INSTALL

To install bmmltools use the Anaconda propt. In the propt, copy the lines below

```
> (base) conda create -n new_env python=3.8
> (base) conda activate new_env
> (new_env) conda install pytables=3.6.1
> (new_env) conda install hdbscan
> (new_env) pip install bmmltools
```

CHAPTER

TWO

DATA AND TRACE

bmmltools is organized around two key objects:

- *Data*: which is used to load external data;
- *Trace*: which is used to store all the partials results, and more generally execute various task during the application of a series of operations to some input data.

2.1 Data

`Data` is the python module used to load data in bmmltools. Input data are loaded and stored in an hdf5 file, saved in a folder selected by the user. `Data` is able to load may formats in different ways, which are listed below. Each of these input type has its own specific input method in `Data` which is indicated between parenthesis.

- *stacks* save in a multitiff (see `load_stack`);
- *stacks* saved as tiff slice-by-slice in a folder (see `load_stack_from_folder`);
- *numpy arrays* which are homogenous in data-type and contains only numeric or boolean data (see `from_array`);
- *file .npy* containing numpy array homogenous in data-type and contains only numeric or boolean data (see `load_npy`);
- *pandas dataframe* (see `from_pandas_df`);
- *json files* containing pandas dataframe (see `load_pandas_df_from_json`);
- *csv files* containing pandas dataframe (see `load_pandas_df_from_csv`).

In addition to these, there are two main methods:

- `new`, to create a new hdf5 file to store that data loaded. The hdf5 file will be create in a folder, specified in the `working_folder` field of this function. The code below show how to create a new `Data` object from numpy array

```
import numpy as np
from bmmltools.core.data import Data


# initialize a Data object
d = Data()
d.create(working_folder = r'[PATH TO SOME FOLDER]')

# load data from a numpy array
arr = np.random.uniform(0,1,size=10)
d.from_array(arr,'x')
```

With the code above, the content of the numpy array `arr` is stored in the hdf5 file in a dataset called `x`.

There is also the possibility to specify the working folder directly in the data initialization. Keeping this in mind the initialization lines in the code above can be reduced to

```
d = Data(working_folder = r'[PATH TO SOME FOLDER]')
```

Note: The name of the hdf5 file created by `Data` has a standard structure, which is `data_XXXX.hdf5`. `XXXX` is a 4 digits code which is randomly generated once the file is created in order to uniquely identify this file: these 4 digits are called *trace code*.

- `link`, to link the data object initialized to an already existing hdf5 file (typically already containing some dataset previously loaded). To link an initialized `Data` object to an existing hdf5 file, one need to specify the folder where the file is in the field `working_folder`, and the data code (as string) in the field `data_code`.

  The code lines below show how this can be done.

```
from bmmltools.core.data import Data


# initialize a Data object
d = Data()

# link an existing hdf5 file.
d.link(working_folder=r'[PATH TO FOLDER WITH data_XXXX.hdf5 FILE]',data_code='XXXX')
```

Note: `infodata` is method of a `Data` object which can be used to print the current datasets present in the hdf5 file linked to it.

```
d.infodata()
```

This command is particularly useful to inspect a `Data` objects after linking to check the content of the hdf5 file linked.

Once `Data` object are created and filled with some input method or linked to some hdf5, the dataset can be used by specifying the its name within square parenthesis, as showed in the example before. After these square parenthesis one can use the slicing notation of h5py, which mimic the numpy slicing notation.

```
print(d['x'][0])
```

The line above print the 0-th element of the dataset called `x` present in the hdf5 file linked to the `Data` object. Alternatively one can use `use_dataset` , which can be particularly useful if the dataset have to be used many times. Consider the example below

```
# ...
# [creation and filling of a Data object or linking to an hdf5 file]
# ...

# select a dataset
d.use_dataset('x')
print(d[0])
print(d[1])
```

(continues on next page)

(continued from previous page)

```
# unselect a dataset
d.use_dataset(None)
print(d['x'][0])      # <- This should work.
print(d[1])           # <- This should give rise to error.
```

In the code above, the dataset `x` is first specified, and then every time the data object is called the use of this particular dataset is assumed: the first two prints will print the elements 0 and 1 of the dataset without the need of specifying the dataset name two times. To "unselect" a dataset `None` should be given as argument of `use_dataset`. As showed in the example above, in this case one have to proceed in the standard way, as the two last line of code above should show.

2.2 Trace

`Trace` is the core class of bmmltoools. It is used to track all the intermediate results during the application of a series of operation, in automatic manner and *without keeping these results in the computer RAM*. `Trace` produces a series of file in a folder called *trace folder*, which is a folder created at a path specified by the user (see below). The trace folder has a standard name: `trace_XXXX`, where `XXXX` is a random 4 digits number (called *trace code*) uniquely identifying the trace. The files generated by `Trace` are listed and explained below.

- *trace hdf5*: here the intermediate results are stored. This file is produced once the trace is created (see below) and is the file which can be linked to a `Trace` object.
- *trace json*: here the trace graph, i.e. all the information to reconstruct the sequence of operations applied on a given trace, and the parameters of the various operations applied on the trace are stored in a dictionary-like format.
- *trace dill*: here the initialized operation applied on a trace are saved as dill object once the application of them terminate (i.e. they are saved in the state they have at the end of the application on a dataset contained in the trace). This file is produced only if `enable_trace_graph = True` when the `Trace` object is initialized (this is the default setting).

When operations act on a trace they can produce a series of folders where various files are saved during the application on the trace. The `Trace` object is also responsible for the creation and organization in a standard way of these folder. These folders are organized as follows.

- *trace file folder*, to save the intermediate quantities produced during the application of an operation. The path to this folder is standard (it is a folder called `trace_files` inside the trace folder) and can be obtained calling the method `trace_file_path`.
- *trace readings folder*, to save the final result one has at the end of the application of an operation (i.e. possibly an intermediate result of the application of a series of operations). The path to this folder is standard (it is a folder called `trace_readings` inside the trace folder)and can be obtained calling the method `trace_readings_path`.
- *trace outputs folder*, the *output operations* store the files produced in this folder when they are applied. The path to this folder is standard (it is a folder called `trace_outputs` inside the trace folder) and can be obtained calling the method `trace_outputs_path`.

Form a practical point of view, `Trace` works similarly to Data. More precisely, a `Trace` object once initialized needs to create or to be linked to an hdf5 file. Two methods are used for that:

- `create` is used to create an hdf5 file (and a trace json too). To create an hdf5 file one needs to specify a folder where the trace folder is created. This is done by specifying the path in the `working_folder` field. It is also possible to specify the group where all the intermediate results are stored in the field `group_name`. By default the group is the root of the hdf5 file, i.e. the intermediate results are stored in the dataset `/[variable_name]`. When the group is specified, the intermediate results are saved at `/[group_name]/[variable_name]` . The code below show how to initialize a new trace.

```
from bmmltools.core.trace import Trace


# initialization with creation of necessary file of a trace
t = trace()
t.create(working_folder=r'[SOME FOLDER PATH]', group_name='[GROUP NAME]')
```

It is not mandatory to used groups inside a trace but they can be useful: groups can used to give some internal organization to the hdf5 trace file, keeping separated intermediate results coming from different pipelines of operations, for example.

- `link` is used to link an initialized `Trace` object to an already existing hdf5 file (and json file) containing the trace. To do that one needs to specify the path to the trace folder in the `trace_folder` field, and the name of the group (if any) in the `group_name` field

```
from bmmltools.core.trace import Trace


# initialization of a trace object with link to an existing trace folder
t = trace()
t.link(trace_folder=r'[TRACE FOLDER PATH]', group_name='[GROUP NAME]')
```

Since a trace can be organized in groups, one can create a new group or change the group used to store the data. This can be done using the methods `change_group` and `create_group` whose meaning is self-explaining.

Attention: It is possible to specify in the trace the seed used for all the random steps of the various operations applied on the trace. This can be done right ater the creation/linking of an hdf5 file simply as showed below

```
#...
t.seed = 5
```

Given a `Trace` object linked to some hdf5 file, one can initialize a new variable in the trace, recover the content of a variable tracked on the trace, or delete a variable using the python's standard ways. The example below shows the basic usage of a `Trace` object.

```
from bmmltools.core.trace import Trace


# initialize a trace creating all necessary trace files
t = trace()
t.create(working_folder=r'[SOME FOLDER PATH]')

# add an initialized variable to the trace
t.x = 4

# recover a variable from the trace
print(t.x)

# change value to a variable on the trace
t.x = 5
print(t.x)
```

(continues on next page)

(continued from previous page)

```
# remove a variable from the trace
del t.x
print(t.x) # <- this should give rise to error.
```

It is important to keep in mind that the variable `x` is in RAM only the time necessary to print it: for the rest of the time the variable is stored in an hdf5 file. This is particularly useful when one has to use many different variables containing data occupying a lot of RAM. Note that in the example above the whole content of `x` is loaded in RAM.

Finally, also `Trace` has a method to get information over the trace content, which is `infotrace`. This method can be used to get the names of the variables that are currently under tracking *on the hard disk*, the variable type, the groups available on the trace, and group currently used to store the variables.

```
t.infotrace()
```

2.2.1 Supported variable types

Trace is able to automatically store-read-delete variables on the Hard Disk (i.e. inside the trace hdf5 file) only if they are of specific formats. These formats are listed below.

- **Homogenous numpy array**: namely numpy arrays of any shape and dimension hose elements are numbers and all of the same type, i.e. only boolean,integer,float or complex.

  ```
  import numpy as np
  ...

  # ...
  # [initialization and linking to an hdf5 file of a trace object]
  # ...

  # create an nd array
  arr = np.random.uniform(0,1,size=(10,10,10))

  # store value of arr in x then erase from the RAM
  trace.x = arr
  del arr

  # read the whole x and print the content
  print(trace.x)
  ```

- **Homogenous numeric dataframe**: namely pandas dataframe whose elements are all of the same numeric type. The numeric types supported are the same of the previous data format.

  ```
  import pandas as pd
  ...

  # ...
  # [initialization and linking to an hdf5 file of a trace object]
  # ...

  # create an pandas dataframe
  df = pd.DataFrame({'X':[1,2,3,4],'Y':[5,6,7,8],'Z':[9,10,11,12]})
  ```

(continues on next page)

(continued from previous page)

```
# store value of arr in y then erase from the RAM
trace.y = df
del df

# read the whole y and print the content
trace.y
```

- **Dictionary of the two variable types listed above**: namely a dictionary whose keys are homogenous numpy arrays and/or homogenous numeric dataframe. One can read and write individual keys of the dictionary by using the methods `read_dictionary_key` and `write_dictionary_key`.

```
import numpy as np
import pandas as pd
...

# ...
# [initialization and linking to an hdf5 file of a trace object]
# ...

# create a dictionary to save
dictionary_to_trace = {'x': np.random.uniform(0,1,size=(10,10,10)),
                       'y': pd.DataFrame({'X':[1,2,3,4],'Y':[5,6,7,8],'Z':[9,10,11,
↪12]})}

# store value of arr in x then erase from the RAM
trace.dictionary = dictionary_to_trace
del dictionary_to_trace

# read the whole 'dictionary' and print the content
print(trace.dictionary)

# read just one key of 'dictionary'
trace.read_dictionary_key('dictionary','x')

# write just one key of 'dictionary'
trace.read_dictionary_key('dictionary','x',np.array([1,2,3]))
```

- **External link to dataset in other hdf5 files**: it is used to avoid to copy the content of the input dataset which is present in `Data` object, saving space on the Hard Disk.

  Note: This external link depends on the path to the `Data` object. Therefore if the content of the folder created by `Data`, where its hdf5 file is created, is changed, the external link would not work (see external links in h5py).

What does not fall in these categories can be added to a trace but its content remain in RAM.

Note: The decision on where the variables are stored (in the hdf5 file or in RAM) is done automatically by `Trace` and cannot be selected by the user.

CHAPTER

THREE

OPERATIONS IN GENERAL

In bmmltools operations are the collective name of all the transformations that one can apply to some input dataset to produce some other dataset stored in the same trace of the input. They have all the same structure, which is discussed here, while the details about the specific operations can be found in the “Operations” section of the side-menu’.

Initialization

Operations are always initialized passing to them a trace object. This is the trace object on which they acts, i.e. from which they take the input dataset and save the operation result. For some operation one needs to specify additional information, like python function or classes having a specific structure.

As example consider the code below.

```
# ...
from bmmltools.operations.featore import Binarizer

# ...
trace = ...  # this is some trace object
# ...

binariezer = Binarizer(trace)
```

Setting inputs and outputs

Right after initialization typically one has to specify the inputs and or the outputs of the operation. In particular, one has to specify the names of the dataset in the trace used as input, and (possibly) the name(s) given to the dataset(s) created in the trace hdf5 file where the operation results are saved. When just one input or output is required one may simply specify the name in a string, while when there are more a list of string need to be used. Every operation has 3 methods to deal with these setting:

- method `i`, to specify the inputs;
- method `o`, to specify the outputs;
- methods `io`, to specify at the same time both the inputs (in the first argument) and outputs (in the second argument).

It is not necessary to call the methods every time a new operation is initialized. If the output is not specified, every operation has a default name for the output (i.e. the output is always already specified). Clearly the input method `i` is needed for all the operations one want to *“apply”* (see below), but is not needed when one want just *“read”* the output dataset. It is important to know that these methods returns the operation class itself. In reference to the example above, the specification of the inputs and outputs can be done with the line of code below

```
#...
binarizer.io('input_dataset','binarized_dataset')
```

Note: The previous line of code is equivalent to

```
#...
binarizer.i('input_dataset').o('binarized_dataset')
```

Apply the operation

Every operation has an `apply` methods, which execute the operation on the inputs when is called. the results Operations typically depends on parameters. All the parameters need to be specified as argument of the `apply` method. Differently to the other two situations above, this method does not return the class itself but the list of the outputs. The example below show how to use it, continuing the example above

```
#...
x = binarizer.apply(threshold=0.2)

print(type(x))
print(x)
```

Sometime during the application of the operation certain files are produced. These files are saved in the trace reading folder of the trace, creating a folder having the name `` [OPERATION NAME]_X`` where `X` is a number counting the number of operations applied on that trace since the input layer.

Note: Each operation has an attribute called `pt`, where one can find a dictionary containing all the parameters given to the apply method. These parameters are stored in the trace json file.

Read operation results

The final result of the application of the operation is stored in the hdf5 file of the trace. Despite the user is free to open this file with traditional methods and take the operation result, each operation is equipped with a `read` methods whose goal is to make the operation output "more readable" to the user. When `read` is called, a folder having the name `` [OPERATION NAME]_X`` (`X` is a number counting the number of operations applied on that trace since the input layer) is created in the reading folder of the trace: in this folder all the outputs of the operation are saved in a suitable format. This method return None.

It is important to observe that one do not need to use exactly the same operation used to produce the output dataset (despite the class *has to* be the same). this should be clarified in the continuation of the previous example which can be found below

```
#...
binarizer2 = Binarizer(trace).o('binarized_dataset') # note that this is a new Binarizer␣
↪object
                                                     # which is used to read the result␣
↪of 'binarizer'.
binarizer2.read()
```

3.1 Example of operation usage

The code below is a realistic example of how one should use operations (together `Trace` and `Data` objects). In particular the `Input` (see *input operation page*) and `PatchTransform3D` (see *patch transform operation page*)

```
import numpy as np
from bmmltools.core.data import Data
from bmmltools.core.tracer import Trace
from bmmltools.operations.feature import PatchTransform3d


# input data
data = Data()
data.new(working_folder=r'SOME PATH')
data.from_array(np.arange(0,27).reshape((3,3,3)),'INPUT DATASET NAME')

# initialize the trace
trace = Trace()
trace.create(working_folder=r'SOME OTHER PATH',group_name='SOME GROUP NAME')

# some operation
x = Input(trace).i('INPUT DATASET NAME').apply(data)

f = lambda x: x**2+x+1                                  # function applied to a patch
x = PatchTransform3d(trace,f).io(x,'output_data').apply()
```

Attention: Alternatively the last 3 lines can be replaced with the two lines below

```
f = lambda x: x**2+x+1
PT3D = PatchTransform3d(trace,f).io(x,'output_data')
x = PT3D.apply()
```

The first version can be considered as a sort of "RAM-efficient" version of the code, since the operation class remains in RAM just the time necessary to execute the initialization, setting and the apply method. On the other hand this second approach can be better suited to have access to internal parameters of the operation which can be of some interest.

CHAPTER

FOUR

REFERENCE API

`bmmltools.core.data`	Data object in bmmltools are used to read data coming from external sources and make them compatible with the other bmmltools objects.
`bmmltools.core.tracer`	Trace objects in bmmltools are used to track all the intermediate results (i.e.
`bmmltools.operations.io`	Input/Output operations in bmmltools.
`bmmltools.operations.feature`	Operations used to extract features from some dataset on a trace.
`bmmltools.operations.clustering`	Operations used to do clustering and study some of their properties.
`bmmltools.operations.segmentation`	Operations used to perform segmentation at voxel level using supervised methods.
`bmmltools.operations.explanation`	Operation useful to get an explanation of the clustering obtained starting from a set of understandable features.
`bmmltools.features.dft`	DFT related functions.
`bmmltools.features.dish`	DISH descriptor.
`bmmltools.board.backend.tracegraph`	Graph utils for bmmlboard.
`bmmltools.board.backend.visualization`	Visualization tools used in bmmlboard.
`bmmltools.utils.basic`	Generic utilities used in bmmltools.
`bmmltools.utils.io_utils`	Basic I/O utils used in bmmltools.
`bmmltools.utils.geometric`	Utilities for geometric transformations in bmmltools.
`bmmltools.utils.graph`	Graph related utils.

CHAPTER

FIVE

IO

In `bmmltools.operations.io` all the input-output methods are collected.

5.1 Input

`Input` is used to declare the input data on which the trace have to work *taken from a data object*.

5.1.1 Transfer function

None

5.1.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `data`: (`bmmltools.operations.core.Data`> object) Data object where the input dataset is stored. The dataset name have to be specified in the operation input.

5.1.3 Inputs and outputs

The operation has the following inputs:

description: Input dataset.
data type: numpy array or dataframe.

The operation has the following outputs:

description: Dataset used as input.
data type: numpy array or dataframe.

5.2 InputFromTrace

`InputFromTrace` is used to declare the input data on which the trace have to work *taken from the same trace on which this operation act*.

5.2.1 Transfer function

None

5.2.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `dataset_name`: (str) Name of the dataset used as input.
- `dataset_group`: (str) Name of hdf5 group in which the input dataset is located.

5.2.3 Inputs and outputs

The operation has the following outputs:

> *description*: Dataset used as input.
> *data type*: numpy array or dataframe.

5.3 OutputRawLabels

`OutputRawLabels` is used to produce the output labelling obtained from the *Clustering* or *Clustering_HDBSCAN* operations, applied on the inout dataset.

5.3.1 Transfer function

None

5.3.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `patch_shape`: (tuple[int]) shape of the patch used to perform the clustering, i.e. the patch used to create the patch space.
- `save_separate_masks`: (bool) optional, if True the mask for each label is saved separately, otherwise a colored mask is produced where all the labels are present.

5.3.3 Inputs and outputs

The operation has the following inputs:

description: Input dataset on which the labelling have to be applied.
data type: 3d numpy array.
data shape: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension for the operation input.

description: Dataframe containing the labelling in the patch space, see output of the *Clustering* or *Clustering_HDBSCAN* operation.
data type: pandas dataframe.

The outputs of this operations are saved in the output folders of the trace.

5.4 OutputValidLabels

`OutputValidLabels` is used to produce the output labelling obtained from the *ClusterValidator* operation, applied on the inout dataset.

5.4.1 Transfer function

None

5.4.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `patch_shape`: (tuple[int]) shape of the patch used to perform the clustering, i.e. the patch used to create the patch space.
- `` label_kind``: (str) optional, it can be `'label'` or `'RS_label'` . Kind of label to plot (i.e. the usual one or the one identified via rotational similarity)
- `point_kind`: (str) optional, it can be `'all'`, `'core'`, `'bilayer'` or `'boundary'`. Kind of point in the patch space used to produce the labelling, according to the classification done by the *ClusterValidator* operation.
- `save_separate_masks`: (bool) optional, if True the mask for each label is saved separately, otherwise a colored mask is produced where all the labels are present.

5.4.3 Inputs and outputs

The operation has the following inputs:

description: Input dataset on which the labelling have to be applied.
data type: 3d numpy array.
data shape: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension for the operation input.

description: Dataframe containing the labelling in the patch space, see output of the *ClusterValidator* operation.
data type: pandas dataframe.

The outputs of this operations are saved in the output folders of the trace.

5.5 OutputSegmentation

`OutputSegmentation` is used to produce the output labelling obtained from the *RandomForestSegmenter* operation, applied on the inout dataset.

5.5.1 Transfer function

None

5.5.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `use_RS_labels`: (bool) optional, if True the rotationally similar labels are assumed in the rendering. **Note that if this is True, the operation assumes that also** *RandomForestSegmenter* **was trained with these labels** (i.e. setting `label = 'RS_label'` in this operation).
- `save_separate_masks`: (bool) optional, if True the mask for each label is saved separately, otherwise a colored mask is produced where all the labels are present.

5.5.3 Inputs and outputs

The operation has the following inputs:

description: Input dataset on which the labelling have to be applied.
data type: 3d numpy array.
data shape: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension for the operation input.

description: Labelled 3d dataset., see output of the *RandomForestSegmenter* operation.
data type: 3d numpy array.
data shape: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension for the operation input.

description: Dataframe with the valid clusters, see output of the *ClusterValidator* operation.
data type: pandas dataframe.

The outputs of this operations are saved in the output folders of the trace.

CHAPTER

SIX

FEATURE

In `bmmltools.operations.feature` all the feature extraction methods are collected.

6.1 Binarizer

`Binarizer` is used to produce a *binary image* with the correct values for the operations that are applied after it. The scope of this layer is mainly technical and should be used even on images that have already been binarized. In this way the two values of a binary mask are always 0 and 1, independently on the way the input data has been produced.

6.1.1 Transfer function

Given an input array with entries $I[k,j,i]$, the layer outputs

$$O[k,j,i] = \begin{cases} 1 & \text{if } I[k,j,i] \geq TH, \\ 0 & \text{otherwise,} \end{cases}$$

where TH is a given threshold.

6.1.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `threshold`: (positive float) value of the threshold below which the output is set to zero (default value is 0.5).

6.1.3 Inputs and outputs

The operation has the following inputs:

description: Input 3d dataset.
data type: 3d numpy array.
data shape: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension.

The operation has the following outputs:

> *description*: Binarized 3d dataset.
> *data type*: 3d numpy array.
> *data shape*: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension for the operation input.

6.2 Patch Transform 3D

`PatchTransform3D` is used to apply a function patch-wise on a 3d input data. The function can be specified by the user

6.2.1 Transfer function

Given a patch shape (P_z, P_y, P_x) and an input I having shape (N_z, N_y, N_x), the layers output consist in the collection $\{h_l[k,j,i]\}_{l \in [0,1,\cdots,N_{patches}-1]}$

$$h_l[k,j,i] = f(C_l[I])[k,j,i]$$

where f is the patch transformation function, C_l is the function returning the l-th patch of the input, and $k \in [0,1,\cdots,P_z-1]$, $y \in [0,1,\cdots,P_y-1]$, and $i \in [0,1,\cdots,P_x-1]$. The number of patches is selected by the user when random sampling of patches is used, while when the patch transform is computed along the zyx-grid, the number of patches is equal to

$$N_{patches} = (N_z//P_z) \cdot (N_y//P_y) \cdot (N_x//P_x)$$

where $a//b$ denotes the integer division between a and b.

6.2.2 Initialization and parameters

In the layer initialization one has to specify:

- the trace on which the this operation act;
- the patch transformation function f, which is a function taking the patch as input and returning the transformed patch as output.

The layer parameters of the `apply()` method are:

- `patch_shape`: (tuple[int]) shape of the input patch used.
- `transform_name`: (None or str) optional, name given to the dataset containing the transformed patch.
- `random_patches`: (bool) optional, if True patches are sampled randomly from the regions of the input dataset having non-null volume, otherwise the patches are generated by taking the patches sequentially along the zyx-grid.
- `n_random_patches`: (int) optional, number of patches sampled when the previous field is set True.

6.2.3 Inputs and outputs

The operation has the following inputs:

> *description*: Binarized input dataset.
> *data type*: 3d numpy array.
> *data shape*: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension.

The operation has the following outputs:

> Dictionary with keys:
>
> > *description*: dataset of transformed patches.
> > *data type*: numpy array.
> > *data shape*: $(N_{patches}, x)$ where x is the output shape of the transformation function.
>
> > *description*: dataframe containing the coordinate of each transformed patch in the patch space.
> > *data type*: pandas dataframe.
> > *dataframe shape*: $N_{patches} \times 3$.
> > *columns names*: Z, Y, X.
> > *columns description*: z/y/x coordinate in patch space of the transformed patches contained in the *transformed_patch* array. The correspondence between the three coordinates and the transformed patch has to be understood row-by-row, i.e. the i-th index of the dataframe row correspond to the i-th element along the 0 axis of the *transformed patch* array.

6.3 Patch Discrete Fourier Transform 3D

`PatchDiscreteFourierTransform3D` is used to apply the 3d discrete Fourier Transform (DFT) patch-wise on a 3d input data.

6.3.1 Transfer function

Given a patch shape (P_z, P_y, P_x) and an input I having shape (N_z, N_y, N_x), the layers output consist in the collection $\{h_l[k,j,i]\}_{l\in[0,1,\cdots,N_{patches}-1]}$

$$h_l[k,j,i] = DFT3d(C_l[I])[k,j,i]$$

where $DFT3d$ is the 3d DFT, C_l is the function returning the l-th patch of the input, and $k \in [0,1,\cdots,P_z-1]$, $y \in [0,1,\cdots,P_y-1]$, and $i \in [0,1,\cdots,P_x-1]$. The number of patches is selected by the user when random sampling of patches is used, while when the patch transform is computed along the zyx-grid, the number of patches is equal to

$$N_{patches} = (N_z//P_z) \cdot (N_y//P_y) \cdot (N_x//P_x)$$

where $a//b$ denotes the integer division between a and b.

6.3.2 Initialization and parameters

In the layer initialization one has to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `patch_shape`: (tuple[int]) shape of the input patch used.
- `representation`: (str) optional, specify here the way the output of the DFT is represented: it can be `'module_phase'`, to compute the module and the phase of the DFT coefficients, or `'real_imaginary'`, to compute the real and imaginary part ot the coefficients of the DFT.
- `random_patches`: (bool) optional, if True patches are sampled randomly from the regions of the input dataset having non-null volume, otherwise the patches are generated by taking the patches sequentially along the zyx-grid.
- `n_random_patches`: (int) optional, number of patches sampled when the previous field is set True.
- `use_periodic_smooth_decomposition`: (bool) optional, if True the periodic component of the periodic-smooth decomposition of the 3d DFT is computed to reduce boundary related artifacts in the DFT result.

6.3.3 Inputs and outputs

The operation has the following inputs:

description: Binarized input dataset.
data type: 3d numpy array.
data shape: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension.

The operation has the following outputs:

Dictionary with keys:

description: module (real part) of the 3d DFT patches.
data type: numpy array.
data shape: $(N_{patches}, x)$ where x is the patch shape.

description: phase (imaginary part) of the 3d DFT patches.
data type: numpy array.

data shape: $(N_{patches}, x)$ where x is the patch shape.

description: dataframe containing the coordinate of each patch DFT in the patch space.
data type: pandas dataframe.
dataframe shape: $N_{patches} \times 3$.
column names: Z, Y, X.
column description: z/y/x coordinate in patch space of the module (real) and phase (imaginary part) of the patch DFTs contained in the *module (real)* and *phase (imaginary)* arrays. The correspondence between the three coordinates and these quantities has to be understood row-by-row, i.e. the i-th index of the dataframe row correspond to the i-th element along the 0 axis of the *module (real)* and *phase (imaginary)* arrays.

6.4 Dimensional reduction

`DimensionalReduction` is used to apply a dimensional reduction techniques on a input data. This method should be compatible with all the sklearn matrix decomposition techniques (see here). This operation in bmmltools preinitialized both with PCA and NMF can be used via classes `DimensionalReduction_PCA` and `DimensionalReduction_NMF`.

6.4.1 Transfer function

The input is assumed to be a collection of N n-dimensional objects, $\{i_l\}_{l\in[0,1,\cdots,N]}$. This operation produce as output the sequence $\{o_l\}_{l\in[0,1,\cdots,N]}$, defined as

$$o_l = DR(\text{vec}\ (i_l))$$

where $DR(\cdot)$ is the function performing the dimensional reduction, $vec(\cdot)$ perform the vectorization of the n-dimensional object (i.e. the object is "flatten" in a 1-d vector). The dimensional reduction algorithm can be trained on the same input collection $\{i_l\}_{l\in[0,1,\cdots,N]}$, or on a difference sequence and the applied to the input collection.

6.4.2 Initialization and parameters

In the layer initialization one has to specify:

- the trace on which the this operation act.
- dimensional reduction class, i.e. a scikit-learn compatible class for dimensional reduction with all the standard methods and

The layer parameters of the `apply()` method are:

- `inference_key`: (str) optional, if all the inputs except the last are a dictionary, this is the name of the key of the dictionary where the inference dataset is located.
- `training_key`: (str) optional, if the last input is a dictionary, this is the name of the key of the dictionary where the training dataset is located.
- `n_components`: (int) optional, number of components to keep for dimensional reduction. If the dimensional reduction algorithm does not have this attribute, this parameter can be ignored.

- `p`: (dict) optional, dictionary containing the parameters for the initialization of the dimensional reduction class (if needed).
- `save_model`: (bool) optional, if True the dimensional reduction model is saved using joblib.
- `trained_model_path`: (str) optional, path to a dimensional reduction model saved using joblib. When this field is not None, this operation automatically assume that the model loaded is already trained: therefore no training is anymore performed.

6.4.3 Inputs and outputs

Assuming the operation has N inputs, the inputs are organized as follow:

description: inference datasets, but it is also the training dataset when just a single input is given. When the `inference_key` is given this operation assume the inference dataset to be located at the specified key of a dictionary stored on the trace. When the second input is not given and `inference_key` is given, one need to specify also the `training_key`.
data type: numpy array.
data shape: (N, x), where N is the number examples in the dataset while x is the shape of the data point.

description: (optional) training dataset. When the `training_key` this operation assume the training dataset to be located at the specified key of a dictionary stored on the trace.
data type: numpy array.
data shape: (N, x), where N is the number examples in the dataset while x is the shape of the data point.

The operation has the following outputs:

description: projected datasets (i.e. datasets after the dimensional reduction).
data type: numpy array.
data shape: (N, x), where N is the number examples in the inference dataset specified in the input while x is the shape of the projected data point. If the dimensional reduction class has the attribute `n_components`, x is equal to that number.

6.5 Data standardization

`DataStandardization` is used to standardize in various way the dataset given in input.

6.5.1 Transfer function

Given an input data x organized in an array with shape $(N_1, N_2, \cdots)$, i.e. $[x_{a_1,a_2,\cdots}]_{a_1\in[0,1,\cdots,N_1-1],a_2\in[0,1,\cdots,N_2-1],\cdots}$, for a given axis i selected by the user, this operation computes

$$m_i = \frac{1}{N_i}\sum_{a_i=0}^{N_i-1} x_{a_1,a_2,\cdots,a_i,\cdots} \tag{6.1}$$

$$s_i = \sqrt{\frac{1}{N_i}\sum_{a_i=0}^{N_i-1} (x_{a_1,a_2,\cdots,a_i,\cdots} - m_i)^2} \tag{6.2}$$

The output is an array y having the same shape and the same number of elements of the input data, given by

$$y_{a_1,a_2,\cdots} = \frac{x_{a_1,a_2,\cdots} - m_i}{s_i}.$$

When more than one axis is specified, the normalization above is applied in sequence to each axis according to the order specified by the user.

6.5.2 Initialization and parameters

In the layer initialization one has to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `axis`: (int or tuple[int]) axis along which the standardization take place.
- `save_parameters`: (bool) optional, if True the parameters m_i and s_i used for the standardization are saved.
- `load_parameters`: (bool) optional, if True the parameters are loaded and not computed from the input training dataset.
- `parameters_path`: (str) optional, path tp the precomputed parameters to use when `load_parameters` is True.
- `inference_key`: (str) optional, if not None this operation assume the inputs to be dictionaries and this field specify the key where the inference dataset can be found in each of these dictionary.
- `training_key`: (str) optional, if not None this operation assume the last input to be a dictionary and this field specify the key where the training dataset can be found.

6.5.3 Inputs and outputs

Assuming the operation has N inputs, the inputs are organized as follow:

description: input inference datasets, i.e. the dataset on which the standardization is applied, but it is also the training dataset when just a single input is given.. When the `inference_key` is given this operation assume the inference dataset to be located at the specified key of a dictionary stored on the trace. When the second input is not given and `inference_key` is given, one need to specify also the `training_key`.
data type: numpy array.
data shape: arbitrary but all the same.

description: (optional) training dataset, i.e. the dataset on which the standardization parameters are computed. When the `training_key` this operation assume the training dataset to be located at the specified key of a dictionary stored on the trace.

data type: numpy array.
data shape: the same of all the other inputs.

The operation has the following outputs:

description: standardized datasets.
data type: numpy array.
data shape: equal to the shape of the input datasets.

CHAPTER

SEVEN

CLUSTERING

In `bmmltools.operations.clustering` all the clustering related methods are collected.

7.1 Clusterer

`Clusterer` is the generic methods which can be used to apply some clustering algorithm available in scikit-learn (see here) to some dataset present on a trace. Two pre-initialized clusterer are available:

- `Clusterer_KMean` where the K-Mean clustering is used;
- `Clusterer_DBSCAN` where the DBSCAN clustering is used.

7.1.1 Transfer function

Given an input array $I = \{i_l\}_{l\in[0,1,\cdots,N-1]}$ with shape (N, x), where N is the number of different data point in the input dataset while x is the number of features on which the clustering algorithm is applied, the layer outputs the sequence of labels $\{o[l]\}_{l\in[0,1,\cdots,N]}$ where

$$o[l] = \text{CLU}(i_l)$$

where CLU is the chosen clustering algorithm.

7.1.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act;
- a clustering class among the scikit-learn clustering methods, or a scikit-learn compatible clustering class. This initialization parameter do not need to be specified when one uses `Clusterer_KMean` or `Clusterer_DBSCAN`.

The layer parameters of the `apply()` method are:

- `p`: (dict) dictionary containing the initialization parameters of the clustering algorithm.

7.1.3 Inputs and outputs

The operation has the following inputs:

description: Input dataset on which the clustering is performed.

data type: numpy array.

data shape: (N, x), where N is the number of different data point in the input dataset while x is the number of features on which the clustering algorithm is applied.

description: Input dataframe where the 3d coordinate in the patch space of the features in the input 0 are stored.

data type: pandas dataframe.

dataframe shape: $N \times 3$, where N is the number of different data point contained in the array specified in the input 0.

columns names: Z, Y, X.

columns description: z/y/x coordinate in patch space of the features contained in the input 0. The correspondence between the three coordinates and the features has to be understood row-by-row, i.e. the i-th index of the dataframe row correspond to the i-th element along the 0 axis of the array specified in the input 0.

The operation has the following outputs:

description: dataset with the clustering result.

data type: pandas dataframe.

dataframe shape: $N \times 4$, where N is the number of different data point contained in the array specified in the input 0.

column names: Z, Y, X, label.

column description: the first 3 columns are the z/y/x coordinate in patch space of the input feature, while in the label column the result of the clustering algorithm (i.e. the label associated to the clusters found) is stored.

7.2 Clusterer HDBSCAN

`Clusterer_HDBSCAN` is used to apply the HDBSCAN clustering algorithm (see here) to some dataset present on a trace.

7.2.1 Transfer function

Given an input array $I = \{i_l\}_{l\in[0,1,\cdots,N-1]}$ with shape (N, x), where N is the number of different data point in the input dataset while x is the number of features on which the clustering algorithm is applied, the layer outputs the sequence of labels $\{o[l]\}_{l\in[0,1,\cdots,N]}$ where

$$o[l] = \text{argmax}(\text{softHDBSCAN}(i_l))$$

where softHDBSCAN is the HDBSCAN clustering algorithm used in soft clustering mode (i.e. assigning the each data point the probability to belong to *each* cluster rather than the cluster label itself). The cluster assigned to a given input i_l is the one with the highest probability.

7.2.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `p`: (dict) dictionary containing the initialization parameters of the HDBSCAN clustering algorithm.
- `save_model`: (bool) optional, if True the trained HDBSCAN clustering algorithm is saved using joblib.
- `trained_model_path`: (str) optional, if not None the HDBSCAN algorithm is loaded from a joblib file. Therefore the algorithm is not trained but the loaded model is used instead (which is assumed already trained). Note that when this option is used it is not ensured that the clustering algorithm will work for any combination of initialization parameters used during the training.

7.2.3 Inputs and outputs

The operation has the following inputs:

description: Input dataset on which the clustering is performed.
data type: numpy array.
data shape: (N, x), where N is the number of different data point in the input dataset while x is the number of features on which the clustering algorithm is applied.

description: Input dataframe where the 3d coordinate in the patch space of the features in the input 0 are stored.
data type: pandas dataframe.
dataframe shape: $N \times 3$, where N is the number of different data point contained in the array specified in the input 0.
columns names: Z, Y, X.
columns description: z/y/x coordinate in patch space of the features contained in the input 0. The correspondence between the three coordinates and the features has to be understood row-by-row, i.e. the i-th index of the dataframe row correspond to the i-th element along the 0 axis of the array specified in the input 0.

The operation has the following outputs:

> *description*: dataset with the clustering result.
>
> *data type*: pandas dataframe.
>
> *dataframe shape*: $N \times 4$, where N is the number of different data point contained in the array specified in the input 0.
>
> *column names*: Z, Y, X, label.
>
> *column description*: the first 3 columns are the z/y/x coordinate in patch space of the input feature, while in the label column the result of the clustering algorithm (i.e. the label associated to the clusters with the highest probability) is stored.

7.3 ClusterValidator

`ClusterValidator` is used validate the clustering obtained from the clustering algorithm available in bmmltools. The clustering algorithm does not take explicitly into account the spatial requirements which a true cluster should have, like the spatial continuity and a sufficiently big volume. This operation check that.

7.3.1 Transfer function

Given an input table where the 3d space coordinates in the patch space of a given label are listed, the validity of the label assigned to a given point in the patch space is checked with the 3 following criteria:

- labels are valid if they are sufficiently continuous in patch space, i.e. they survive to a binary erosion followed by a binary dilation in patch space (eventually filling the holes remaining inside the labels);
- after the erosion-dilation process a cluster is considered valid if it has a volume bigger than a given threshold (and similarly is checked checked of the volume of the core part of the cluster is above a certain threshold);
- after the erosion-dilation process a point in the patch space is valid if it is assigned to just one label.

The points are also classified in 3 categories:

- *core point* of a cluster, i.e. the points in the patch space which are not at the boundary with another label. These point are **assumed** to contain a good representation of the component defining the cluster. Core points are defined by eroding 1 time the valid point of the cluster.
- *bilayer points* of a cluster, i.e. the points in the patch space having spatial continuity in at least two of the three dimensions (these points are not considered if the validation is done in 2D mode).
- *boundary points* of a cluster, i.e. the points which valid but are not cor or bilayer points. These points are assumed to be unreliable to study a cluster, since they should contain mixture of different components (being at the boundary of the cluster).

7.3.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `patch_space_volume_th`: (int) optional, minimum volume a cluster should have in the patch space to considered as a valid cluster (default value is 1);
- `patch_space_core_volume_th`: (int) optional, minimum volume the core part of a cluster should have in the patch space to considered as a valid cluster (default value is 1).

7.3.3 Inputs and outputs

The operation has the following inputs:

description: Table where coordinates in patch space and corresponding labels are listed.
data type: pandas dataframe.
data shape: $N \times 4$, where N is the number of data points.
columns names: Z, Y, X, label.
columns description: z/y/x coordinate in patch space of the transformed patches contained in the *transformed_patch* array. The correspondence between the three coordinates and the transformed patch has to be understood row-by-row, i.e. the i-th index of the dataframe row correspond to the i-th element along the 0 axis of the *transformed patch* array.

The operation has the following outputs:

description: Table containing *only the valid clusters* and information about the kind of point in the patch space.
data type: pandas dataframe.
dataframe shape: $N \times 10$, where N is the number of different row contained in the dataframe specified in the input 0.
columns names: Z, Y, X, label, core_point, bilayer_point, z_bilayer, y_bilayer, x_bilayer, boundary_point.
columns description: the first three columns are the z/y/x coordinate in patch space of the feature used to produce the clustering, while the cluster label is saved in the label columns. After that, information about the nature of the the point in the patch space, is indicated with a 1 in the corresponding column (and 0 otherwise).

7.4 ArchetypeIdentifier

`ArchetypeIdentifier` is used to define a set of representative for each cluster found. The elements belonging to this set of representative are called archetypes of the clusters. They can be used to study the properties of a given cluster using less computational resources.

7.4.1 Transfer function

The archetype are defined by sampling a region of the patch space (suitably expanded if needed) according to the probability distribution constructed by normalizing the distance transform of the region in the (expanded) patch space corresponding to a given cluster. The sampling region is the one having probability above a given threshold selected by the user.

7.4.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `patch_shape`: (tuple[int]) shape of the patch used to define the features used for the clustering.
- `archetype_threshold`: (float between 0 and 1) optional, threshold used on the normalized distance transform of each cluster to define the sampling region for the archetypes. The sampling region for a given cluster
- `N_archetype`: (int) optional, number of archetype per cluster.
- `extrapoints_per_dimension`: (tuple[int]) optional, dilation factor for each dimension of the patch space used to have a more realistic spatial region for the sampling of the archetypes.
- `filter_by_column`: (str) optional, when this field is not None, this is name of the column to filter dataframe specified in the input dataset. Only the points having 1 in this column are used to define the archetypes of each cluster. When this field is None, all dataframe given in the input is used.
- `save_archetype_mask`: (bool) optional, if True a mask showing for each cluster the actual region sampled for the archetype is saved in the trace file folder.

7.4.3 Inputs and outputs

The operation has the following inputs:

> *description*: Table containing *only the valid clusters* and information about the kind of point in the patch space.
> *data type*: pandas dataframe.
> *dataframe shape*: $N \times 10$, where N is the number of different row contained in the dataframe specified in the input 0.
> *columns names*: Z, Y, X, label, core_point, bilayer_point, z_bilayer, y_bilayer, x_bilayer, boundary_point.
> *columns description*: the first three columns are the z/y/x coordinate in patch space of the feature used to produce the clustering, while the cluster label is saved in the label columns. After that, information about the nature of the the point in the patch space, is indicated with a 1 in the corresponding column (and 0 otherwise).

description: Dataset from which the archetype are sampled.
data type: 3d numpy array.
data shape: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension.

The operation has the following outputs:

Dictionary with keys:

description: Dataset containing the archetypes of all the labels.
data type: numpy array.
data shape: (N, P_z, P_y, Px), where (P_z, P_y, Px) is the patch shape and N is total number of archetype is equal to the number of archetype (parameter selected by the user) multiplied by the number of valid clusters.

description: Table where coordinates in patch space and corresponding labels of the sampled archetype are listed.
data type: pandas dataframe.
data shape: $N \times 4$, where N is the number of data points.
columns names: Z, Y, X, label.
columns description: The first three columns are z/y/x coordinate in patch space of sampled archetype, while in the label column the corresponding label is stored. The correspondence between the three coordinates and le label and the archetypes stored in the *archetype* key of the output dictionary has to be understood row-by-row, i.e. the i-th index of the dataframe row correspond to the i-th element along the 0 axis of the archetype array.

7.5 RotationalSimilarityIdentifier

`RotationalSimilarityIdentifier` is used to suggest possible identification of the clusters based on similarity under rotation of two labels.

7.5.1 Transfer function

Two labels are considered similar under rotation when the procedure describe below give a positive result.

1. All the archetypes of two different clusters, say A and B, are taken and evaluated in the spherical coordinates.
2. The radial distribution of each archetype is computed by integrating over the angles for both the clusters.
3. The mean value and the covariance matrix of the radial distribution is computed for both clusters.
4. The one-value Hotelling T-test is run to see if the archetype of the cluster A can be sampled from a probability distribution with mean value B

5. The one-value Hotelling T-test is run to see if the archetype of the cluster B can be sampled from a probability distribution with mean value A
6. When both test performed at point 4 and 5 give positive answer, the identification of the two labels is suggested.

After that the identification test is executed the angles among two clusters are computed by looking at the mean modulus of the 3d DFT of archetypes considered and performing a weighted correlation among the angular parts of the archetype at various radii. More about the rotational identification procedure and the angles among different archetypes can be found in [LINK TO PAGE IN THE MISCELLANEOUS SECTION OR WHATEVER] section.

7.5.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `p_threshold`: (float between 0 and 1) optional, threshold below which two cluster are considered rotationally similar.
- `smooth`: (bool) optional, if True the modulus of the 3d DFT is smoothed using a gaussian filter.
- `sigma`: (float) optional, standard deviation of the gaussian filter used to smooth the modulus of the 3ed DFT.
- `spherical_coordinates_shape`: (tuple[int]) optional, shape of the modulus of the 3d DFT once evaluated in spherical coordinates. The $\rho\theta\phi$-ordering is assumed.
- `bin_used_in_radial_dist`: (tuple[int]) optional, tuple indicating the start and stop bin of the radial distribution considered for the statistical test.

7.5.3 Inputs and outputs

The operation has the following inputs:

description: Dictionary containing the output of the *ArchetypeIdentifier*. Refer to that for more detail.

description: Table containing the valid clusters and information about the kind of point in the patch space.
data type: pandas dataframe.
dataframe shape: $N \times 10$, where N is the number of different row contained in the dataframe specified in the input 0.
columns names: Z, Y, X, label, core_point, bilayer_point, z_bilayer, y_bilayer, x_bilayer, boundary_point.
columns description: the first three columns are the z/y/x coordinate in patch space of the feature used to produce the clustering, while the cluster label is saved in the label columns. After that, information about the nature of the the point in the patch space, is indicated with a 1 in the corresponding column (and 0 otherwise).

The operation has the following outputs:

Dictionary with keys:

description: Table where the identification probability of each pair of labels is written.
data type: pandas dataframe.
data shape: $N_l \times N_l$, where N_l is the number of valid labels.
columns names: for each cluster with label x a column called `Label_:math:`x`` is present.

description: Table where the power of the statistical test performed among the two labels is written.
data type: pandas dataframe.
data shape: $N_l \times N_l$, where N_l is the number of valid labels.
columns names: for each cluster with label x a column called `Label_:math:`x`` is present.

description: Table where theta angle among each pair of labels is written. Clearly this number make sense only for the pairs that can be identified.
data type: pandas dataframe.
data shape: $N_l \times N_l$, where N_l is the number of valid labels.
columns names: for each cluster with label x a column called `Label_:math:`x`` is present.

description: Table where phi angle among each pair of labels is written. Clearly this number make sense only for the pairs that can be identified.
data type: pandas dataframe.
data shape: $N_l \times N_l$, where N_l is the number of valid labels.
columns names: for each cluster with label x a column called `Label_:math:`x`` is present.

description: Table where if two labels can be identified or not according to the user setting is written.
data type: pandas dataframe.
data shape: $N_l \times N_l$, where N_l is the number of valid labels.
columns names: for each cluster with label x a column called `Label_:math:`x`` is present.

In addition the dataframe specified by the input 1 a column called `RS_label` is added where the labels considered similar under rotation by the algorithm are identified. The other columns of this dataframe are left unchanged.

CHAPTER

EIGHT

SEGMENTATION

In `bmmltools.operations.segmentation` all the feature segmentation methods are collected. Segmentation methods in this model are assumed to work at voxel level.

8.1 RandomForestSegmenter

`RandomForestSegmenter` is used to produce a segmentation at voxel level by training a random forest in a supervised manner to assign to each voxel a given label.

8.1.1 Transfer function

The segmentation is done by computing a set of feature (see below) from the data to label and train a random forest in order to classify different voxels in different labels, using the valid labels identified by the ClusterValidator after clustering. The feature used are the quantities below:

- intensity feature, i.e. the input data convolved with a gaussian kernel (1 feature).
- edge features, i.e. the sobolev filter applied to the input data convolved with a gaussian kernel (1 feature).
- texture features, i.e. the eigenvalues of the Hessian matrix computed at each voxels computed using the input data convolved with a gaussian kernel (3 features).
- directional features, i.e. the elements of the structure matrix computed at each voxel computed using the input data convolved with a gaussian kernel (6 features, see here for more details).

The user can select which feature use or not. All these features are computed at different scales, i.e. by convolving the input data with a gaussian kernel having different σ (scale parameter).

8.1.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `patch_shape`: (tuple[int]) shape of the patch used to define the valid clusters given in the input 0.
- `label_type`: (str) optional, it can be `'label'` or `'RS_label'` (for rotationally similar labelling if available).
- `reference_points`: (str) optional, it can be `None`, `'core_point'`, `'bilayer_point'` or `'boundary_point'`. It is kind of point in the patch space used to define the labels on which the random forest classifier is trained. The default value is `'core_point'`.

- `sigma_min`: (float) optional, minimum scale at which the features are computes for the training of the random forest classifier.
- `sigma_max`: (float) optional, maximum scale at which the features are computes for the training of the random forest classifier.
- `n_sigma`: (int) optional, number of different scales at which the features for the training of the random forest classifier are computed.
- `intensity`: (bool) optional, if True the "intensity features" are computed and used for the training of the random forest. By "intensity features" the input data convolved with a gaussian kernel having sigma equal to the scale parameters specified by the user is understood.
- `edge`: (bool) optional, if True the "edge features" are computed and used for the training of the random forest. By "edge features" the the Sobolev filter applied to the input data convolved with a gaussian kernel having sigma equal to the scale parameters specified by the user is understood.
- `texture`: (bool) optional, if True the "texture features" are computed and used for the training of the random forest. See here for more about this kind of features.
- `direction`: (bool) optional, if True the "direction features" are computed and used for the training of the random forest. See here for more about this kind of features.
- `N_training_samples_per_label`: (int) optional, number of voxels with all the computed features (i.e. number of samples) used to train *each* estimator (decision tree) of the random forest for each label.
- `inference_split_shape`: (tuple[int]) number of split of the input dataset per dimension used to do the inference. This is needed especially when the input data *plus* all the features computed cannot fit in RAM.
- `n_estimators`: (int) optional, number of estimators (decision tree) used in the random forest classifier.
- `save_trained_random_forest`: (bool) if True the trained random forest is saved in the trace file folder.

8.1.3 Inputs and outputs

The operation has the following inputs:

description: Input 3d dataset to segment.
data type: 3d numpy array.
data shape: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension.

description: Table containing only the valid clusters and information about the kind of point in the patch space (i.e. the typical output of *ClusterValidator*).
data type: pandas dataframe.
dataframe shape: $N \times 10$, where N is the number of different row contained in the dataframe specified in the input 0.
columns names: Z, Y, X, label, core_point, bilayer_point, z_bilayer, y_bilayer, x_bilayer, boundary_point.
columns description: the first three columns are the z/y/x coordinate in patch space of the feature used to produce the clustering, while the cluster label is saved in the label columns. After that, information about the nature of the the point in the patch space, is indicated with a 1 in the corresponding column (and 0 otherwise).

The operation has the following outputs:

> *description*: Labelled 3d dataset. **Note that 1 is added to all the labels, since 0 is also typically the convention about empty space in the data to segment**.
> *data type*: 3d numpy array.
> *data shape*: (N_z, N_y, N_x), where N_i is the number of voxels along the i-th dimension for the operation input.

CHAPTER

NINE

EXPLANATION

In `bmmltools.operations.explanation` are collected all the methods which can be used to get an explanation of some clusters in terms of a series of human-understandable input features.

9.1 MultiCollinearityReducer

`MultiCollinearityReducer` is used to detect and reduce the multicollinearity in the dataset used for the explanation in order to get a more stable and reliable explanation in terms of the features selected by the user.

9.1.1 Transfer function

Given an input dataset organized in tabular form, where each columns correspond to a feature and each row correspond to a different data point, the multicollinearity reduction consist in selecting a subset of feature which are not to much linearly related among each other. This happens by computing for each feature i the Variance Inflation Factor (VIF) defined as

$$VIF_i = \frac{1}{1-R_i^2}$$

where R_i^2 is the usual R^2 coefficient used to evaluate the goodness of fit of a linear regression model trained to predict the i-th feature in terms of all the other features in the dataset. Once the VIF is computed for all the features, if some of them are above a certain threshold value, the feature with the highest VIF is removed from the dataset. This procedure is repeated till the VIFs of all the remaining variables are below the chosen threshold. The surviving features are approximately linear independent among each other.

9.1.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `data_columns`: (list[str]) list containing the name of the columns of the dataframe specified in the input 0 of this operations.
- `target_columns`: (str or list[str]) name(s) of the column(s) containing the target variable(s) to explain.
- `VIF_th`: (float) threshold one the Variance Inflation Factor.
- `return_linear_association`: (str) optional, it can None or 'pairwise' or 'full'. When is not None, the linear association dictionary is saved as json file in the trace file folder. Two possible linear association are available:

- *pairwise*, where the coefficients of a linear regression model to predict the value of each eliminated feature in term of **one of the feature surviving the multicollinarity screening** are saved.
- *full*, where the coefficients of a linear regression model to predict the value of each eliminated feature in term of **all of the features surviving the multicollinarity screening** are saved.

9.1.3 Inputs and outputs

The operation has the following inputs:

description: Input dataframe where as set of input features and target variable(s) are stored

data type: pandas dataframe.

dataframe shape: $N \times x$, where N is the number of different data point from which the features are computed, and x is the number of columns for the features plus the number of columns for the targets plus possibly other columns.

columns names: selected by the user, but some of them will be the features and some of them the target(s).

The operation has the following outputs:

description: Output dataframe where the surviving features are approximately linear independent.

data type: pandas dataframe.

data shape: $N \times M$, where N is the number of different data point from which the features are computed, and M is the number of columns for the features plus the number of columns for the targets.

columns names: names of the selected features and name(s) fo the target(s).

9.2 ExplainWithClassifier

`ExplainWithClassifier` is used to get an explanation of a target variable using a certain set of features in set theoretic terms (if possible).

9.2.1 Transfer function

This operation computes the permutation importance (PI) and the partial dependency (PD) of each feature from a of classifier trained to recognize a given target label using a set of input feature. To reduce the model dependency of these quantities are computed for an ensemble of different classifier (trained with different hyperparameters) is used and mean and standard deviation of each PI and PD are returned (weighted using the model F1 score to recognize the target variable). By looking to these two quantities one can deduce an explanation for a given cluster variable as follow:

- From the PI one can deduce the features which are most important for the recognition of a given label as the one having the highest positive values.

- From the PD of the most important features identified with the PI, and knowing that the classifiers output 1 when when the target variable is recognized, the range of value of a given feature where the PD is high, is likely to be the region of values which that feature should have in order to be recognized as belonging to the given label.

In this way one can deduce for each label a series of intervals of a small subset of label which can be used to define the label in term of human understandable features.

9.2.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `test_to_train_ratio`: (float between 0 and 1) optional, fraction of dataset used for the performance evaluation of the random forest trained.
- `model_type`: (str) optional, use just 'RandomForest' for the moment (which is also the default value). In principle this should be the model type used to get the explanation, but currently only random forest is available.
- `n_kfold_splits`: (int) optional, number of train-validation split used to select the best random forest for each parameter combination.
- `n_grid_points`: (int) optional, number of different values of a feature on which the partial dependence is computed.
- `save_graphs`: (bool) optional, if True the graph of the feature permutation importance and the partial dependency for each label are saved in the trace file folder.

9.2.3 Inputs and outputs

The operation has the following inputs:

description: Input dataframe where as set of input features and a target variable is stored. The target variable is **always** assumed to be the last column of the dataframe.
data type: pandas dataframe.
dataframe shape: $N \times x$, where N is the number of different data point from which the features are computed, and x is the number of columns for the features plus one.
columns names: names of the features and name of the target.

The operation has the following outputs:

Dictionary. The key below is present.

description: Dataframe containing the F1 score of the classifier trained for the various target.
data type: pandas dataframe.
dataframe shape: $N \times 1$, where N is the number of labels.
columns names: for each label Label_x, where x is label.

*columns description: average F1 score for the trained classifiers, which can be used to deduce if the input features contains enough information to learn to recognize the various labels.

For each label the key below are present:

description: Dataframe containing the permutation importance for the considered label.

data type: pandas dataframe.

dataframe shape: $N \times 2$, where N is the number of features used.

columns names: estimated_permutation_importance_mean, estimated_permutation_importance_std.

*columns description: mean and standard deviation of the permutation importance.

For each label and feature the key below is present:

description: Dataframe containing the partial dependency of a given feature for the considered label.

data type: pandas dataframe.

dataframe shape: $N \times 2$, where N is the number of features used.

columns names: estimated_partial_dependency_mean, estimated_partial_dependency_std.

*columns description: mean and standard deviation of the partial dependency of a given feature.

9.3 InterpretPIandPD

`InterpretPIandPD` is used to get automatically an interpretation of the PI and PD (computed with the operation *ExplainWithClassifier*) in terms of simple intervals of the of features.

9.3.1 Transfer function

This operation tries to perform the following two operations:

1. *Estimate the most relevant features for the definition of a label.* This is done by sorting the features in decreasing order with respect the permutation importance of the label. The most important features are selected by taking the sorted features till the total positive permutation importance reach a certain threshold set by the user is reached.
2. *Estimate the intervals of the most relevant features which can be used to define the label.* This is done using an histogram based estimation techniques, since no a priori functional behavior can be assumed for the partial dependency. This estimation techniques work as follow: an histogram is contracted from the values of the partial dependency and then the Otsu threshold for this histogram is computed. The normal Otsu threshold is used if the

histogram has just two peaks, when more than two peaks are detected an Otsu multithreshold technique is used and the highest threshold is considered. The intervals one is looking for are the one where the partial dependency is above the found threshold.

The intervals found in the point 2, can be used to define an interpretable model, i.e. a model where the classification can be understood in terms of if-else condition on the input features (which are assumed human understandable). However, the result the interval defined in the point 2 it is likely to be suboptimal, i.e. the accuracy of the interpretable model is not maximal. For a limited number of cases (i.e. for intervals which can be defined with one or two threshold) a bayesian optimization procedure which find the best intervals, in the sense of maximize the (balanced) accuracy of the interpretable model.

9.3.2 Initialization and parameters

In the layer initialization one have to specify:

- the trace on which the this operation act.

The layer parameters of the `apply()` method are:

- `positive_PI_th`: (float between 0 an 1) optional, fraction of positive permutation importance which the most relevant features need to explain in order to be selected for the explanation of a given label.
- `n_bins_pd_mean`: (int) optional, number of bins of the partial dependency used in the histogram based detection method for the interval definition.
- `prominence`: (None or float) optional, prominence parameter of the peak finder algorithm used in the histogram based detection method for the interval definition.
- `adjust_accuracy`: (bool) optional, if True the adjusted balanced accuracy is used to compute the interpretable model performance
- `bayes_optimize_interpretable_model`: (bool) if True the interval deduced using the histogram based detection method are optimized using bayesian optimization in order to maximize the accuracy.
- `bo_max_iter`: (int) optional, maximum number of iterations of the bayesian optimization.
- `save_interpretable_model`: (bool) optional, if True the interpretable model is saved.

9.3.3 Inputs and outputs

The operation has the following inputs:

description: Output of the *ExplainWithClassifier* operation.
data type: dictionary.

description: Output of the *MultiCollinearityReducer* operation.
data type: pandas dataframe.

The operation has the following outputs:

Dictionary. The key below is present.

description: Dataframe containing information about the feature relevance for all the labels.
data type: pandas dataframe.
dataframe shape: $N \times M$, where N is the number of labels and M the number of features.
columns names: features name.
*columns description: for each label (row of the dataframe) as 1 is present if the feature is relevant for the label description and 0 otherwise.

description: Dataframe containing the accuracy of the interpretable model.
data type: pandas dataframe.
dataframe shape: $N \times 1$, where N is the number of labels.
columns names: balanced_accuracy (or balanced_adjusted_accuracy, depending on the user input).
*columns description: for each label (row of the dataframe) the balanced (adjusted) accuracy is stored.

For each label the key below are present:

description: Interval of the fearure f for the definition of the label l .
data type: pandas dataframe.
dataframe shape: $2 \times x$, where x is the number of threshold used to define the interval.
columns names: thresholds, post_thresholds_value.
columns description: in the 'thresholds' column the value where the thresholds defining the interval, while in the 'post_thresholds_value' the value assumed by an indicator function *after* the threshold.

CHAPTER TEN

BMMLBOARD

bmmlboard can be used to visualize intermediate results stored on trace. Not all the intermediate result can be visualized via bmmlboard, since it is not always possible to find an useful representation of the result without knowing the specific scope of the visualization. In this case the user need to access to these data directly from the trace if a visualization is needed.

To run bmmlboard, simpy run the Ananconda propt and in the same python environment where bmmltools is installed simply run

```
> python -m bmmltools.run_bmmlboard
```

It is a streamlit based app (therefore it need a web browser to work) and organized in 3 different module. Once bmmltools open, specify in the main page the **absolute** path of the trace to inspect. On the menu' on the left the available modules are listed. By clicking on one of them the different visualization tools can be used. For all the modules one need to specify the group name used in the trace to store the intermediate result. Some module require to select the dataset used as input data.

10.1 Label Visualizer

It is the module used to visualize the intermediate result produced by the operations listed below:

- *PatchDiscreteFourierTransform3d*;
- *Clusterer*;
- *Clusterer_HDBSCAN*;
- *ClusterValidator*;
- *ArchetypeIdentifier*;
- *RotationalSimilarityIdentifier*.

10.2 Segmentation Visualizer

It is the module used to visualize the intermediate result produced by the operations listed below:

- *RandomForestSegmenter*.

10.3 Explainer Visualizer

It is the module used to visualize the intermediate result produced by the operations listed below:

- *MultiCollinearityReducer*;
- *ExplainWithClassifier*;
- *InterpretPIandPD*.

CHAPTER

ELEVEN

CASE OF STUDY: SEGMENTATION PIPELINE

In this tutorial it is described a possible sequence of operations for the segmentation in a 3d binary images of a biological sample of regions having different structural/texture properties.

11.1 Goals

The sequence of operations (pipeline) described in this tutorial has been developed to achieve the following goal.

- Perform the segmentation of a 3d binary image coming from various biological samples studied for their structural/textural properties. In practice this means to color in different manner the regions of the sample whose structures are similar.

However this have to be done under a series of requirements which heavily influence the way to proceed, as will be clear from the next sections. The main requirement is the following:

- The segmentation does not have a ground true reference. This means that the user does not have to specify in advance which are the different structures present in the sample, or a set of criteria which can be used to identify the different regions composing the sample.

At a first look this requirement look very unrealistic, by the way this is a quite common situation for big 3d samples since labelling some them (i.e. producing a ground true reference) is a very time consuming operation and more importantly even the expert in the field may not label them in an objective way. This means that the labelling can be highly subjective, making hard to consider it as a solid ground true reference for the problem. Indeed these samples are typically produced to study their properties, therefore it is natural that their defining properties are not known in advance.

11.2 General structure of the pipeline

Once the goals and the requirements under which they have to be achieved are set, one can start to design a general solution for the problem under consideration.

First of all the lack of a ground true reference force to work with a clustering algorithm. However, not all the clustering algorithm are suitable for the case considered here. Since no ground true is available, all the clustering algorithms where the number of cluster has to be specified in advance are excluded. Therefore the lack of ground true force to use density-based or hierarchical clustering algorithm.

It is quite reasonable to assume that assignment of a voxel to a given cluster can be based on the local environment around that voxel, i.e. the cluster assigned to a given voxel depends not only on the voxel itself but also to a set of voxels surrounding the one considered. This local environment when assumes a sufficiently regular shape (e.g. a cube or a rectangular parallelepiped) is called *patch*. Therefore, in principle, one may define the dataset for the clustering algorithm, as the set of all the possible patches centred around all the voxels of the sample. By the way this makes the problem intractable from may point of view.

First of all, a dateset constructed in this way would be huge, for the typical 3d images collected in biological setting. Indeed, assuming the typical 3d binary image has $\sim 10^3$ voxels per dimension, the whole image has in total $\sim 10^9$ voxels. Therefore the dataset for the clustering algorithm would have $\sim 10^9$ datapoint, and each datapoint would have the same number of voxels the patch would have. A reasonable patch dimension is of the order $\sim 10^3$ (i.e. ~ 10 voxels per dimension). With these numbers the dataset is typically too big for a clustering algorithm to perform well. Moreover, it is not at all guaranteed that the clustering algorithm is able to identify easily the various components in the input data from patches containing simply a small portion of the input data. It can be that a better representation can be found, in order to recognize different regions based on structural/texture properties. All this force first to extract suitable features from the patches and then eventually use dimensional reduction techniques *before* the training of the clustering algorithm in order to reduce the dimension of the training dataset.

Once the clusters are obtained one needs to understand if the clusters found make sense or not. Indeed any clustering algorithm depends on parameters. It is very unlikely that the parameters can be deduced from an inspection of the input data alone, and a trial-and-error approach has to be followed. This implies two things:

1. The clustering algorithm should give the clustering result in a reasonable amount of time, so that the user can inspect the result and change the parameters rapidly and find the best combination to achieve a reasonable result. This can be done by reducing the number of datapoint of the training dataset in a reasonable manner. A good approach is the one to construct the dataset from patches which does not superimpose among each other but still filling whole volume of the input data. In practice this means to divide the input data according to a 3d grid which use the chosen patch as unit cell. In this way by assigning a cluster label to each patch "taken from the 3d grid" one can obtain a coarse-grained representation of the final clustering which can be used to evaluate the "goodness" of the clusters obtained using a given set of parameters. This coarse-grained representation is what in bmmltools is called *patch space*.

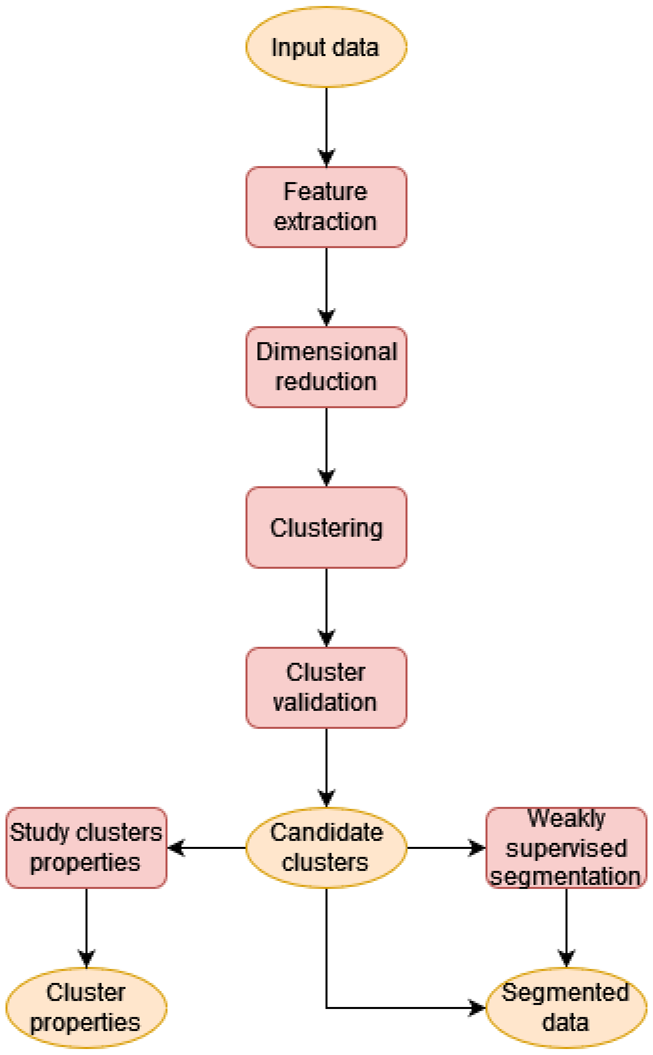
2. The clusters obtained need to be validated, i.e. they need to fulfil a basic set of requirements in order to be considered valid and not just noise (keep in mind that a clustering algorithm is able to find clusters also from noise when certain parameter combinations are used). The basic requirements one may require can be a certain degree of spatial continuity or a volume above a given threshold. Controls like these can be executed fast if they are done in the patch space. The clusters respecting the requirements chosen are called in bmmltoos *valid clusters* or *candidate clusters*.

Once the valid clusters are obtained one may proceed in different way depending on then needs. A simple segmentation result can be obtained by assigning different colors to the regions of the original data whose patches has different labels. In this way one would obtain a segmentation whose contour are based one the geometric shape of the patch chosen This segmentation in bmmltools is called *raw labels segmentation* or *valid labels segmentation*, depending if it is constructed using the cluster before or after validation.

To go beyond the segmentation done at the level of the patches mentioned before, one can perform a segmentation at voxel level by using the clusters (validated) as *weak labels* for a supervised segmentation algorithm. The segmentation obtained in this way is called in bmmltools *refined segmentation*: it should look "nicer" from the border point of view but it is typically very time consuming: therefore one should do this step only when one is certain about the clusters obtained. Moreover, the result obtained does not add nothing from the quantitative point of view since all the information about the clusters can be derived already from the valid labels segmentation.

Finally, one can start to study various cluster properties. An example of study of the cluster properties is described in the *"Case of study: identification for similarity under rotation"* section of this manual, where the rotational properties of the various clusters found are studied in order to suggest which clusters can be identified among each other, because they differ just by a rotation.

The whole procedure is schematized in the diagram below.



11.3 The segmentation pipeline in detail

In this section a possible pipeline of operation based on the general scheme described in the previous section is discussed in details.

11.3.1 Feature extraction

The aim of the feature extraction step is to find a good representation of the portion of input data contained in each patch. The structural/texture properties are well encoded in the modulus of the 3d discrete Fourier transform. Two reason can be given for that (at lest from a qualitative point of view):

1. Mathematically the structural/texture properties of a 3d binary mask can be well described in terms of almost-periodicity [Amerio1989]. In Fourier space the representation of almost-periodic function is particularly compact, since in this space the modulus of an almost periodic function would be peaked on the (pseudo-)frequencies associated with the main almost-periodicity. Therefore, the modulus of the 3d discrete Fourier transform should contain most of the information about the structure/texture.
2. Structural/texture properties are properties that are highly delocalized in space. Therefore it is reasonable to expect that in Fourier space the information is much more concentrated. This is compatible with the entropic uncertainty relations holding for the discrete Fourier transform [DeBrunner2005], since the lower bound on the entropy in the Fourier space would decrease, if the entropy in the real space is high (as one should expect for functions highly delocalized in space).

A technical point need to be done here. A patch is typically a small object, and this has consequences on its discrete Fourier transform. In particular the lack of periodicity at the boundary would imply the presence of a series of boundary artifacts, which should be removed to "display" better the information about the almost periodicity of the structural/texture properties of the sample. A possible technique to remove the boundary artefacts present in the discrete Fourier transform is to consider just the periodic component of the periodic-smooth decomposition of the discrete Fourier transform (pDFT for short) [Moisan2011].

From the arguments above, one can conclude that the *PatchDiscreteFourierTransform3D* has to be used wit the following parameters in the `apply` method:

- `patch_shape = (A,B,C)`, i.e. choosing a patch with dimension $A \times B \times C$ using the zyx ordering;
- `representation = 'module-phase'`;
- `use_periodic_smooth_decomposition = True`.

With this setting, the operation will perform the pDFT on a 3d grid constructed using the patch chosen as building block producing the dataset used for (the inference of, see below) dimensional reduction step.

According to the general scheme given in the previous section the next step will be the dimensional reduction. As will be clear from the previous, this operation need to be trained on some dataset. In order to avoid to have any unreal grid-dependent feature in the training dataset of the dimensional reduction techniques, one can use the pDFT of patches sampled randomly from the input dataset. In practice this means to use the *PatchDiscreteFourierTransform3D* a second time on the same input dataset but with the setting below in the `apply` method.

- `patch_shape = (A,B,C)`;
- `representation = 'module-phase'`;
- `use_periodic_smooth_decomposition = True`;
- `random_patches = True`;
- `n_random_patches = N`, where N is the number of random patches to sample.

With this setting a training dataset for the dimensional reduction step can be produced, while the dataset obtained previously can be used as inference dataset for the next step.

11.3.2 Dimensional reduction

Once that the modulus of the pDFT has been computed one obtain an object having the same dimension of the original patch, which is too big and its dimensionality need to be reduced. The dimensional reduction technique used should be as much as possible agnostic on the nature of the input data: in this way the lack of specific criteria for the identification of different components of the sample is not a problem. Dimensional reduction techniques whose working principle are based on the statistical properties of a dataset are a good candidate for that step.

In general, deep autoencoders [Hinton2006] should be able to find a lower dimensional representation of the dataset on which are trained. This representation is "adapted" to the dataset during the autoencoder training. The simplest autoencoder one can imagine is a linear autoencoder trained using the mean square error as loss function to quantify the reconstruction error of the decoder. It is known that this kind of autoencoder is equivalent (up to a rotation in the latent space, i.e. the vector space in which the low-dimensional representation exist) to a PCA [Bourlard1988] [Chicco2014]. Therefore one can simply use a PCA to get a first dimensional reduction, and if the representation found is not good one can try with more complex autoencoders (currently not implemented in bmmltools).

Therefore, one should use the operation *DimensionalReduction_PCA* with the setting below in the `apply` method.

- `inference_key = 'module'`, since only the module of the pDFT is used;
- `training_key = 'module'`, for the same reason of above;
- `n_components = M`, where M is the number of PCA component to keep, i.e. the dimension of the latent space. It should be a small number compared to the number of voxels of in the patch.

As anticipated in the previous subsection, to avoid to have a grid-dependent low dimensional representation, the training dataset is produced sampling randomly the input data, while the inference is done on grid.

A further step before clustering is needed: the data standardization. No particular justification for that can be given, but it has been observed that the clustering gives better result. This can be done by using the operation *DataStandardization* with the setting in the `apply` method below.

- `axis = (1,0)`, which is the configuration giving the best results in the case tested.

11.3.3 Clustering and validation

A quite powerful clustering algorithm is the HDBSCAN, which is a hierarchical density-based clustering algorithm [Campello2013] [McInnes2017]. It does not need to know in advance the number of clusters, but it needs a series of parameters whose meaning can be found here. The most significant parameters to modify in order to influence the clustering are `min_cluster_size`, `metric` and `min_samples`.

The operation performing HDBSCAN in bmmltools is *Clusterer_HDBSCAN*. In the `apply` method the HDBSCAN parameter need to be specified as dictionary in the `p` parameter.

Attention: The HDBSCAN parameters are probably the parameters that more frequently need to be modified by the user, till a reasonable result is obtained in the patch space. If no reasonable clustering is found by modifying these parameters try to increase the number of components used in the dimensional reduction step.

After that the clusters are found, one needs to validate them for the reason explained in the previous section. Validation in bmmltools con be done using the *ClusterValidator*. This operation check for (a certain degree of) spatial continuity of the clusters found in patch space and the minimum volume a cluster should have in order to be considered valid. The default setting are the less restrictive one.

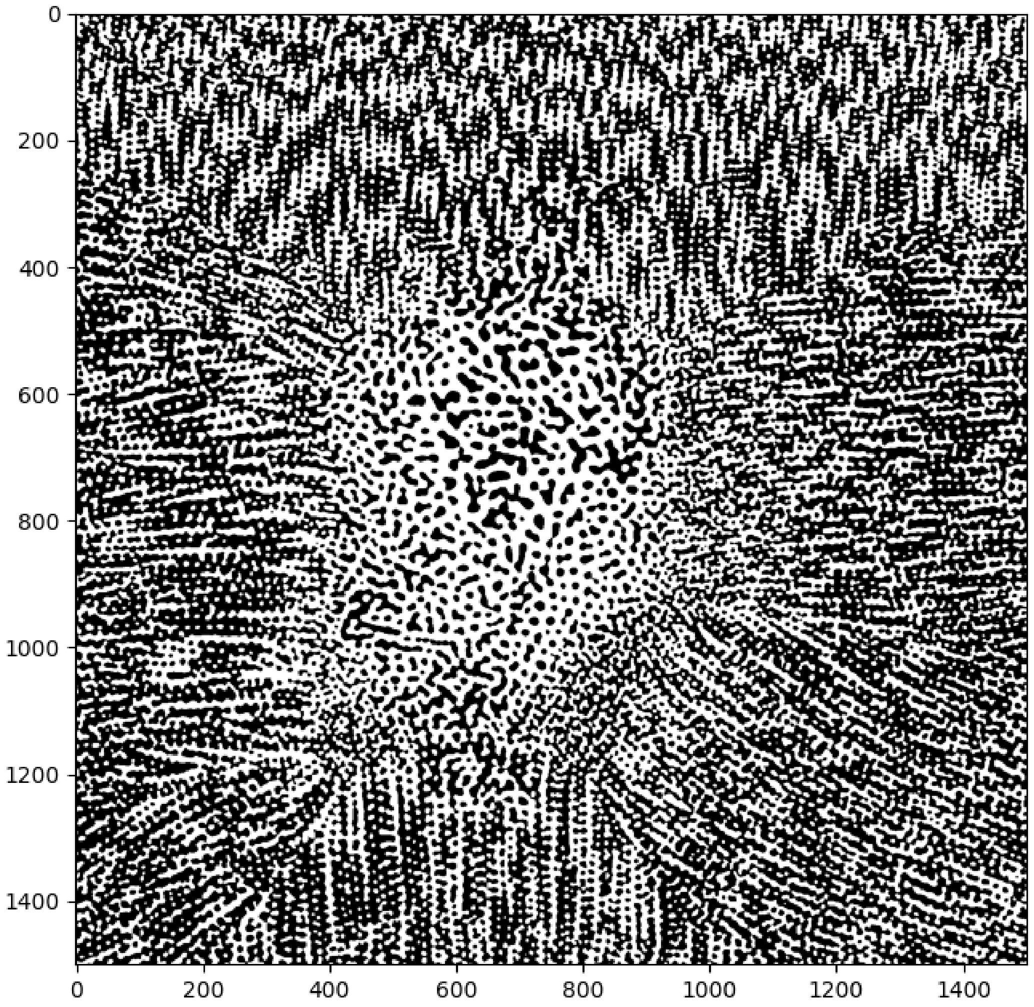
11.3.4 Weakly supervised segmentation

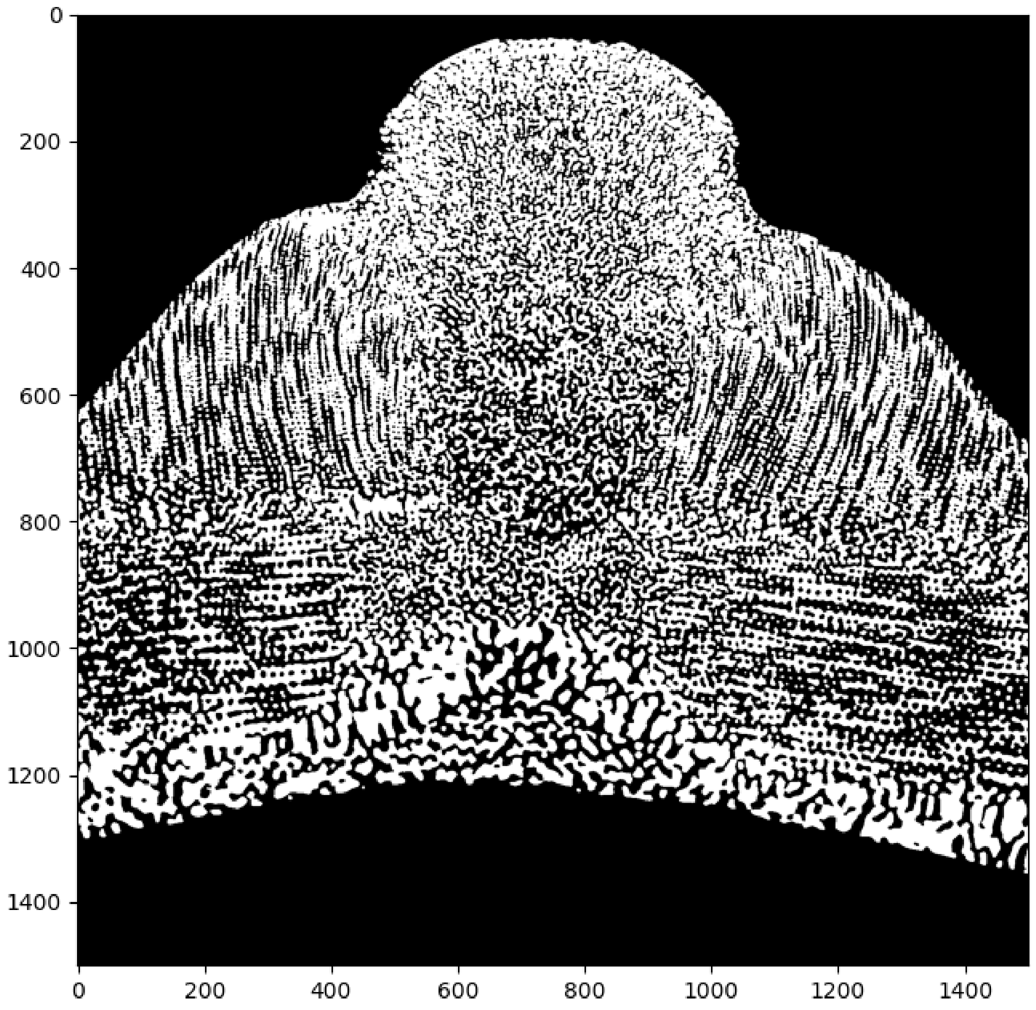
The segmentation one can obtain from the valid clusters, can be found to train in weakly supervised manner a classifier to predict for each voxel of the sample the corresponding cluster. This kind of segmentation should have nicer borders compared to the one can obtain by labeling the patches. In bmmltools a Random Forest classifiers is used.

The operation *RandomForestSegmenter* can be used to perform this kind of segmentation.

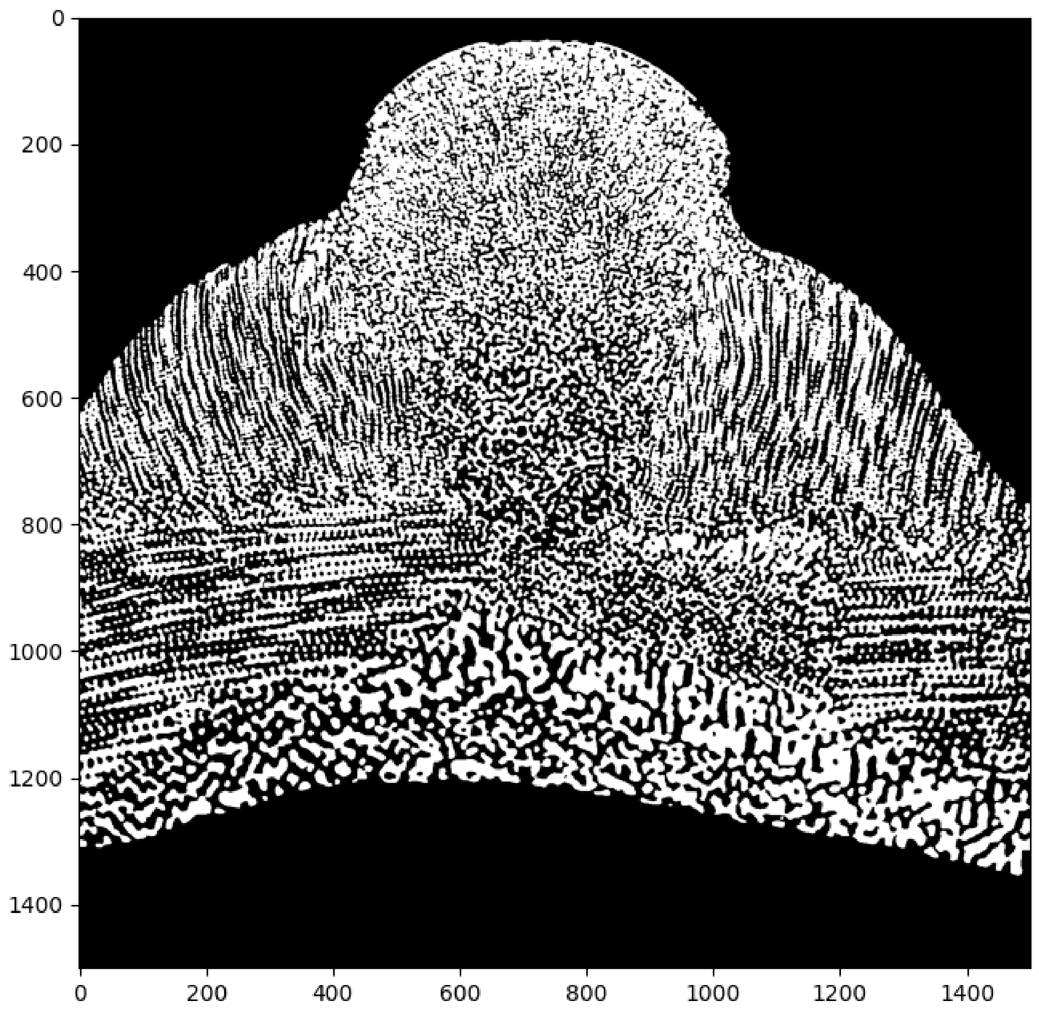
11.4 Example

Below three slice of some input data to label are shown.





0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400



The code implementing the pipeline described in the previous sections is reported below. In addition to the operations described in the previous section, there are two operations are done before to apply the operation performing the pDFT. Input and Binarizer operations are there for technical reason: the first is used to specify the input dataset, while Binarized is used to standardize the input binary mask to 0-1 values.

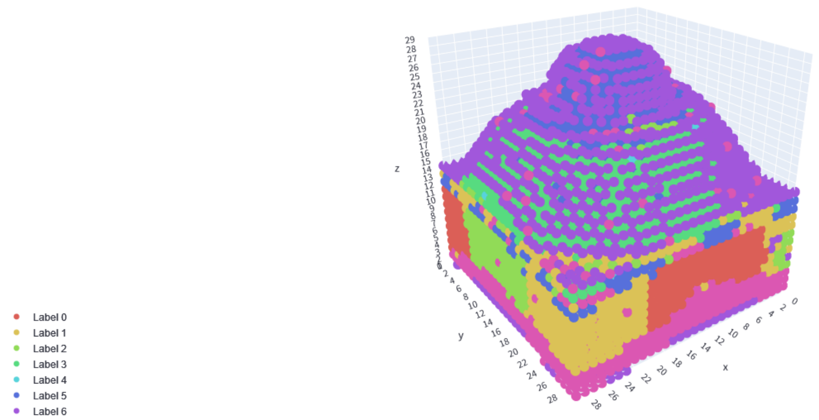
```
from bmmltools.operations.io import Input,OutputRawLabels,OutputValidLabels,
↪OutputSegmentation
from bmmltools.operations.feature import Binarizer,PatchDiscreteFourierTransform3D,
↪DimensionalReduction_PCA,\
                                          DataStandardization
from bmmltools.operations.clustering import Clusterer_HDBSCAN,ClusterValidator,
↪ArchetypeIdentifier,\
                                             RotationalSimilarityIdentifier
from bmmltools.operations.segmentation import RandomForestSegmenter
```

(continues on next page)

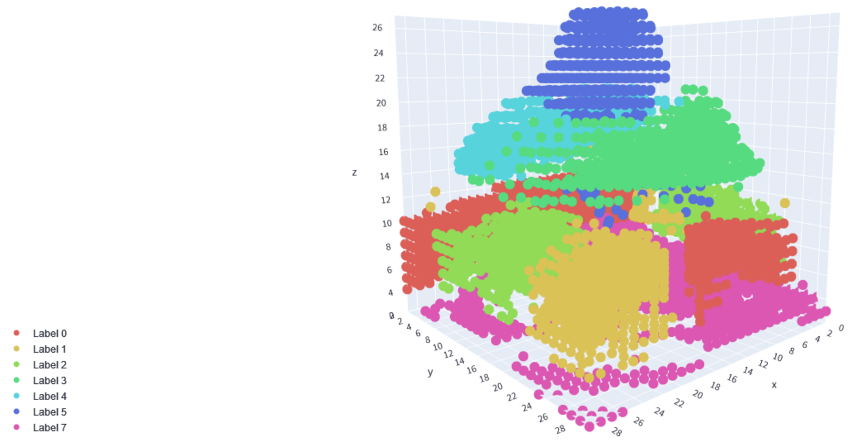
(continued from previous page)

```
# load data
# data = ...

# create a trace
# trace = ...

## segmentation pipeline
 x = Input(trace).i('reg_sea_urchin').apply(data)
 x = Binarizer(trace).i(x).apply()
 x1 = PatchDiscreteFourierTransform3D(trace)\
         .io(x,'post_pdft3d_inference_dataset')\
         .apply(patch_shape=(50,50,50))
 x2 = PatchDiscreteFourierTransform3D(trace)\
         .io(x,'post_pdft3d_training_dataset')\
         .apply(patch_shape=(50,50,50),random_patches=True,n_random_patches=4000)
 x = DimensionalReduction_PCA(trace)\
         .io(x1+x2,['post_dm_pca_inference_dataset'])\
         .apply(inference_key='module',training_key='module',save_model=True)
 x = DataStandardization(trace).io(x[0],'post_ds_inference_dataset').apply(axis=(1,0))
 x = Clusterer_HDBSCAN(trace)\
         .io(x+['post_pdft3d_inference_dataset',],'raw_labels_dataset')\
         .apply(p=dict(min_cluster_size=15,prediction_data=True,metric='euclidean',min_
↪samples=2),save_model=True)
 x = ClusterValidator(trace).io(x,'valid_labels_dataset').apply()
 x = RandomForestSegmenter(trace)\
     .io(['post_b_dataset']+x,'segmentation_result')\
     .apply(patch_shape=(50,50,50),inference_split_shape=(3,3,3),
            n_estimators=50,N_training_samples_per_label=1000,n_sigma=5,
            temp_folder_path=r'SOME PATH FOR TEMPORARY FILE')

 ## outputs
 OutputRawLabels(trace).i(['post_b_dataset','raw_labels_dataset']).apply(patch_shape=(50,
↪50,50))
 OutputValidLabels(trace).i(['post_b_dataset','valid_labels_dataset']).apply(patch_
↪shape=(50,50,50))
 OutputSegmentation(trace).i(['post_b_dataset','segmentation_result','valid_labels_
↪dataset']).apply(use_rgb=False)
```

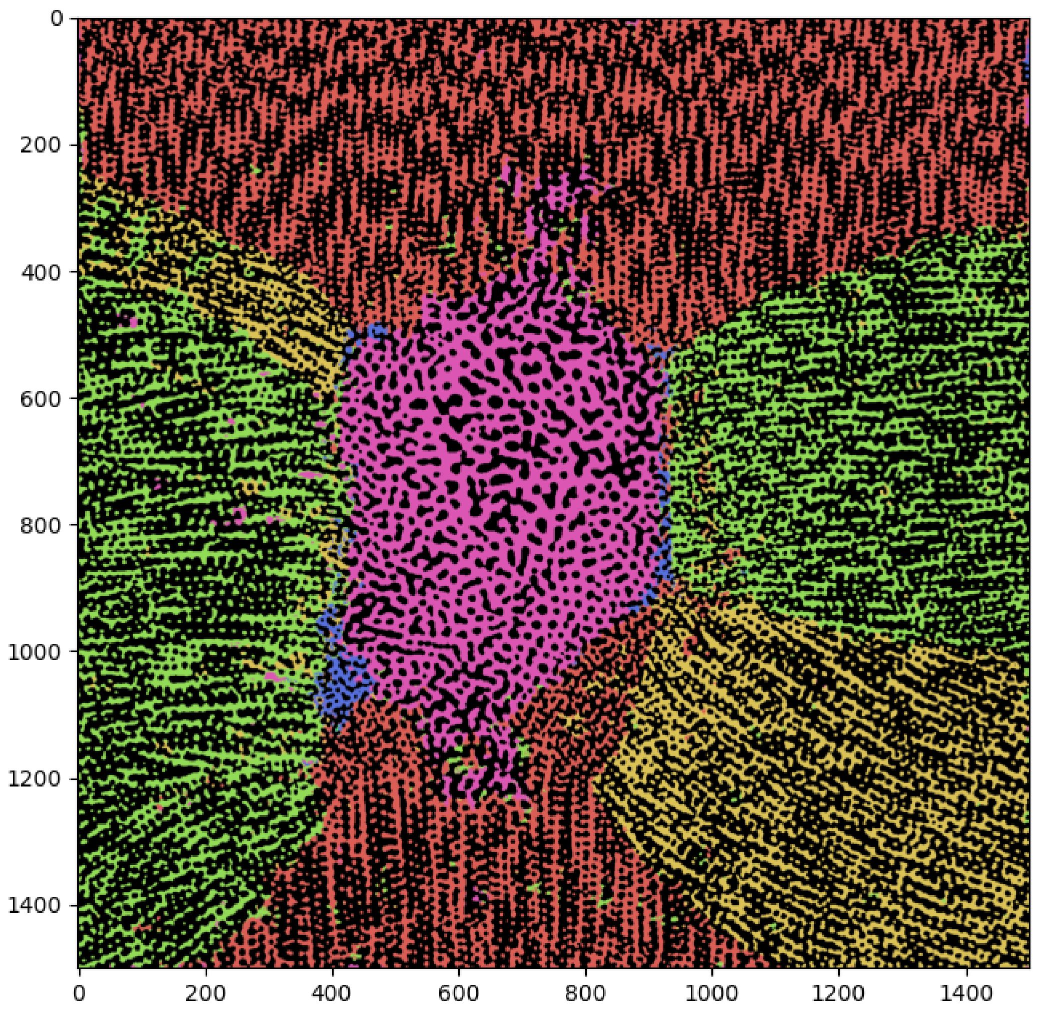
The clusters produced by the HDBSCAN_Clusterer are shoed below in patch space

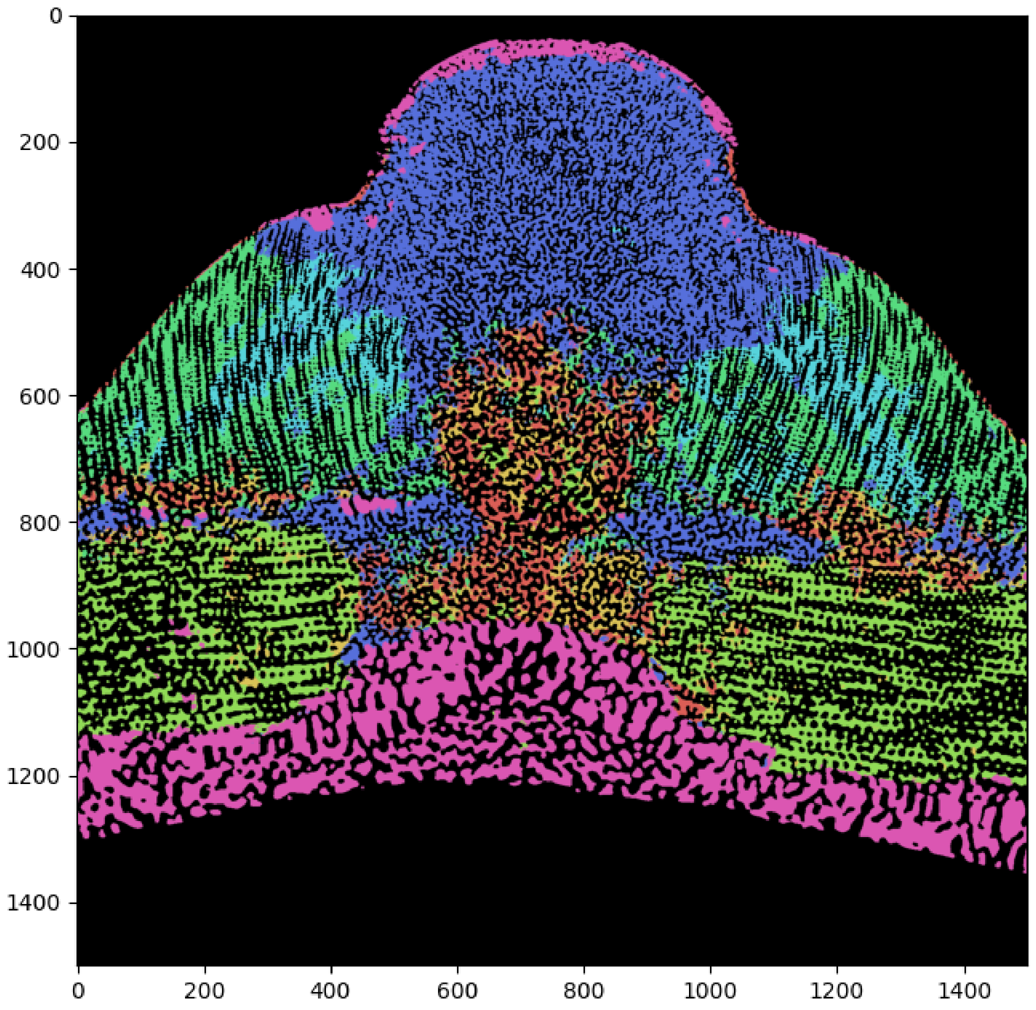


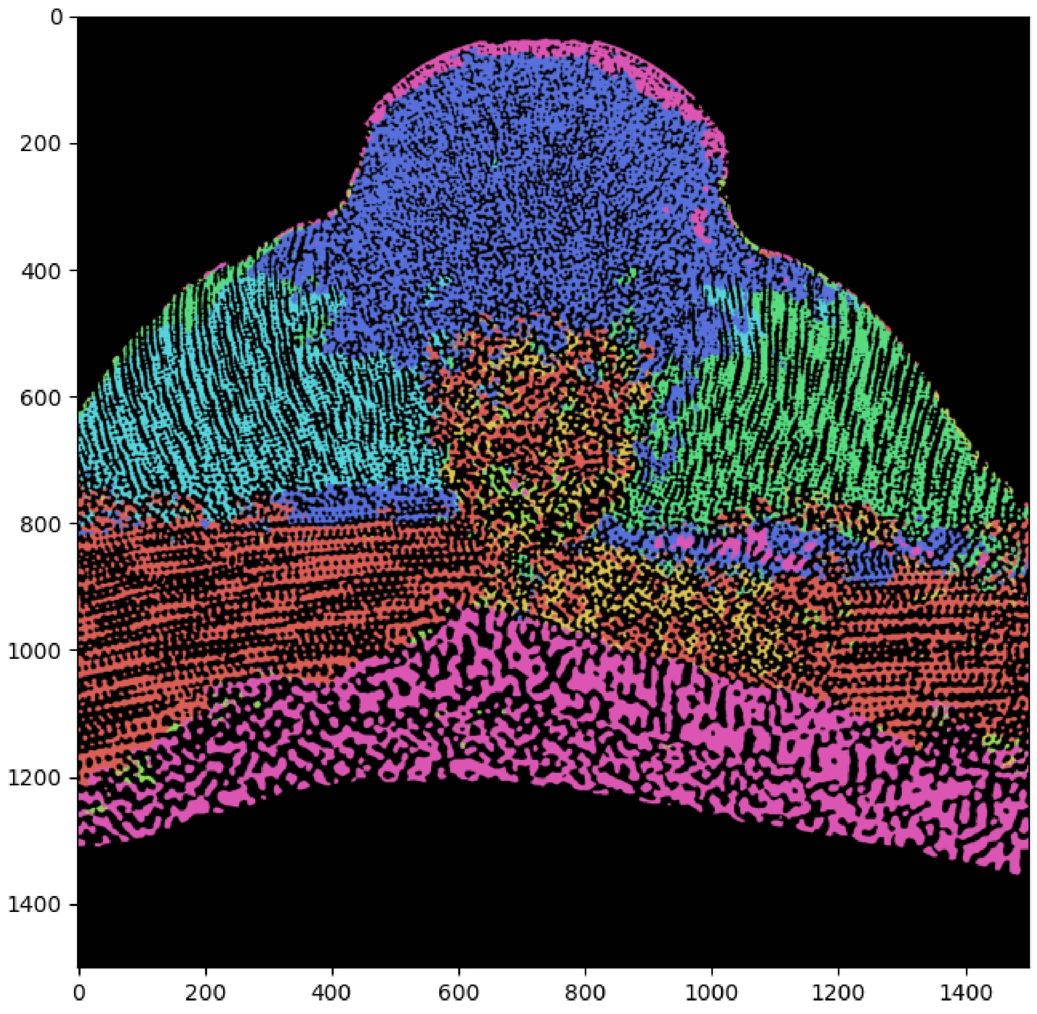
With the clustering algorithm 8 different cluster can be identified, but only 7 are real. The validation step is able to eliminate the dark violet one (cluster number 6), which is unrealistic since it is a border artifact. The core points of the valid clusters are showed below.



The final segmentation using the RandomForestSegmenter can be seen below.


0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400


0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400



11.5 Reference

CHAPTER

TWELVE

CASE OF STUDY: IDENTIFICATION FOR SIMILARITY UNDER ROTATION

Once a set of reasonable valid clusters is obtained following the procedure explained in the tutorial "*Case of study: segmentation pipeline*", the cluster properties can be investigated. One of the most popular request is to understand if two clusters are similar up to a rotation in real space. In other words, given the structures/textures associated to two valid clusters, the question the procedure presented here attempt to answer is if they look similar and cannot anymore distinguished once one of the two is rotated by a suitable angle.

In this tutorial, the functioning of the operations developed to answer this question in bmmltools is explained. The result will be a suggestion on the possible identification which can be done. Note that one speak bout "suggestions" because as will be clear at the end ot this tutorial many things can go wrong, in particular when one checks if the suggested equivalent relation is a proper equivalence relation in mathematical terms. Another important thing to keep in mind, is that the procedure presented here has been developed in order to get an answer in a reasonable amount of times using a modest amount of computational resources, due to the typical dimension of the input data on which these analysis should be carried out.

12.1 Archetypes

As observed at the end of the previous paragraph, a practical constrain on the for the solution of the rotational similarity problem, is to get an answer in a limited amount of time and with a limited amount of computational resources. This lead to the notion of *archetype of a given cluster*, which is a set of patches containing parts of the input data located in regions of the data associated to the considered cluster. Archetypes are considered as good representative of a given cluster, and are used to study the properties of a given clusters. To fulfil the requirements mentioned above, the number of archetypes should be small compared to all the possible patches that can be extracted from the regions associated to a given cluster.

[How they are defined] To ensure that they well capture the properties of a cluster, archetypes are defined according to the requirements below:

- Archetypes are sampled randomly from the central (core) part of cluster, since it is assumed that in this region the cluster properties are less ambiguous more regular. Randomness ensures that the criteria used to sample the regions of the cluster does not influence the properties encoded in the archetypes.
- The number of archetype considered is equal for all the valid clusters, in order to estimate all the statistical properties with similar level of uncertainty.

In practice, in bmmltools the distance transform is used to define the regions for each clusters from which archetypes are uniformly samples in equal number for each cluster. Given the output of the *ClusterValidator*, from the regions corresponding to a valid cluster is reconstructed in the *patch space*, which contains a discrete coarse-grained representation of the clustering result (see *here* for more details), the distance transform is computed. In this way one can just compute for each point in the patch space the smallest distance from the voxel and the cluster border. In in this way one can put a threshold on the result obtained to select the regions having distance to the cluster border at least equal to the threshold (actually the data is normalized to 1, therefore this threshold is a number between 0 and 1). Some time the patch space is too small to have a meaningful distance transform: in this case the patch space is expanded by a certain factor N_i for each dimension, so that each point along a given axis i corresponds to N_i points in the expanded patch space. The distance transform and the threshold operation are then taken in this expanded patch space. Once that the most central region of the cluster is found, it is sampled randomly obtaining the set of archetypes corresponding to the cluster considered.

In bmmltools the archetypes for each clusters are obtained using the *ArchetypeIdentifier* operation. According to the description given, the most important parameter to set are:

- `archetype_threshold`, which is the threshold on the normalized distance transform defining the central part of the cluster in the (eventually expanded) patch space.
- `N_archetype`, which clearly specify the number of archetype sampled for each cluster.
- `extrapoints_per_dimension`: which is where one should declare if needed the expansion factor (one number for each dimension) used to construct the expanded patch space.

Finally, in addition to that in this operation one has always to specify the number of voxels per dimensions of the patch used during the segmentation step in the `patch_shape` field.

12.2 Rotational Similarity measured with a statistical test

In this section the criteria upon which the rotational similarity measure are explained, therefore it represent the core part of the whole procedure used to identify clusters because rotationally similar. Given the set of archetypes found for each label, the identification is suggested based on the similarity of the radial distribution of the modulus of the periodic component of the 3d discrete Fourier transform. More precisely, given the discrete nature of the data, given a patch $\Phi(i_z, i_y, i_x)$, the modulus of the periodic component of the 3d Fourier transform computed, i.e.

$$\hat{\Phi}(k_z, k_y, k_x) = |\text{pDFT}(\Phi)|(k_z, k_y, k_x).$$

This quantity is then evaluated in spherical coordinate obtaining $\hat{\Phi}(k_\rho, k_\theta, k_\phi)$. Then the radial distribution of $\hat{\Phi}(k_\rho, k_\theta, k_\phi)$ is computed obtaining the vector $\rho = (\rho_0, \cdots, \rho_{N_\rho - 1})$. Each component of the vector is given by

$$\rho_i = \sum_{\rho_\theta, \rho_\phi} \hat{\Phi}(i, k_\theta, k_\phi) \sin(\theta(k_\theta))$$

which can be seen as the right Riemann sum approximating the integration over the angles of $\hat{\Phi}(k_\rho, k_\theta, k_\phi)$ assuming $\Delta\theta = \Delta\phi = 1$ and where $\theta(k)$ is k-th value of the θ angle.

Attention: Note that ρ is not able to distinguish two different object which have a different angular organization but the same radial distribution. It is not to difficult to imagine how this can be achieved. Given two real number a and b, consider the two field below:

$$\hat{\Phi}_A(k_\rho, k_\theta, k_\phi) = a\delta_{k_{\theta_0}, k_{\theta_1}} + b\delta_{k_{\theta_2}, k_{\theta_3}} \tag{12.1}$$

$$\hat{\Phi}_B(k_\rho, k_\theta, k_\phi) = (a+b)\delta_{k_{\theta_0}, k_{\theta_1}} \tag{12.2}$$

for some k_{θ_0}, k_{θ_1}, k_{θ_2}, and k_{θ_3}. It is not difficult to see that $\rho_A = \rho_B$, despite the two field clearly have different angular distribution, therefore no rotation can map $\hat{\Phi}_A$ in $\hat{\Phi}_B$, meaning that no rotation can make the patch A similar to the patch B. This lack in the discrimination ability of the the measure used for the rotational identification step is one of the reason why the result of this tutorial can be considered only as a suggestion. In practice, the situation above is not so likely to happens, since $\hat{\Phi}$ have to correspond to the modulus of a pDFT of a real structure/texture which is not trivial at all. Note also that this kind of problem would affect any other measure which removes the angular degrees of freedom by summing over them.

For each valid cluster, given the collection of archetypes produced by the *ArchetypeIdentifier* operation $\{\Phi^{(i)}(k_z, k_y, k_i)\}_{i=1}^{N}$, the radial distribution is computed for each patch as explained above obtaining the collection $\{\rho^{(i)}\}_{i=1}^{N}$. The identification among two or more cluster is based on the radial distributions of the archetypes. In particular, the identification among two clusters is suggested if the radial distributions of the archetype of two clusters can be considered as samples of a random variable coming from the same probability distribution.

To check that, the Hotelling t^2 test is performed, which is the statistical test used to understand if given a p-dimensional random variable X sampled N times, it has mean value μ. In the case considered here given two valid clusters A and B, with radial distributions $\{\rho_A^{(i)}\}_{i=1}^{N}$ and $\{\rho_B^{(i)}\}_{i=1}^{N}$, one computes the mean values μ_A and μ_B and the covariance matrices Σ_A and Σ_B having entries

$$\mu_{q,i} = \frac{1}{N_{\rho_q}} \sum_{k=0}^{N_{\rho_q}-1} \rho_{q,i}^{(k)} \tag{12.3}$$

$$\Sigma_{q,i,j} = \sqrt{\frac{1}{N_{\rho_q}-1} \sum_{k=0}^{N_{\rho_q}-1} (\rho_{q,i}^{(k)} - \mu_{q,j})^2} \tag{12.4}$$

with $q = A, B$. The test is performed symmetrically on A and B, namely assuming A as sample and (the mean of) B as reference, and viceversa. To do that the following quantities are defined

$$t_{AB}^2 = (\mu_A - \mu_B)\Sigma_A(\mu_A - \mu_B)^T \tag{12.5}$$

$$t_{BA}^2 = (\mu_B - \mu_A)\Sigma_B(\mu_B - \mu_A)^T \tag{12.6}$$

which are the statistics used in the Hotelling t^2 statistical tests. Given the centred F-distribution with degrees of freedom p and $N-p$, $F_{p,N-p}(x)$, one can define the confidence level γ_{AB} of the test for the null hypothesis, i.e. the statement that the radial distributions $\{\rho_A^{(i)}\}_{i=1}^{N}$ have mean value μ_B, as

$$\gamma_{AB} = 1 - \alpha_{AB} \tag{12.7}$$

$$= 1 - \frac{N_{\rho_A}(N - N_{\rho_A} - 1)}{N-1} F_{N_{\rho_A}, N-N_{\rho_A}}(t_{AB}^2) \tag{12.8}$$

where N is the number of archetypes and N_{ρ_A} is the number of values used to represent the radial distribution of the cluster A. Similarly one can define γ_{BA} as

$$\gamma_{BA} = 1 - \frac{N_{\rho_B}(N - N_{\rho_B} - 1)}{N-1} F_{N_{\rho_B}, N-N_{\rho_B}}(t_{BA}^2)$$

For the Hotelling t^2 statistical test the statistical power of the test η can be computed using the *non-centred* F-distribution with parameter p, $N-p$ and non-centrality parameter δ, $F_{p,N-p,\delta}^{nc}(x)$, as follow

$$\eta_{AB} = 1 - \frac{N_{\rho_A}(N - N_{\rho_A} - 1)}{N-1} F_{N_{\rho_A}, N-N_{\rho_A}, \delta_{AB}}^{nc}(1 - \gamma_{AB}) \tag{12.9}$$

$$\eta_{BA} = 1 - \frac{N_{\rho_B}(N - N_{\rho_B} - 1)}{N-1} F_{N_{\rho_B}, N-N_{\rho_B}, \delta_{BA}}^{nc}(1 - \gamma_{BA}) \tag{12.10}$$

(12.11)

where $\delta_{AB} = (\mu_A - \mu_B)\Sigma_A(\mu_A - \mu_B)^T$ and $\delta_{BA} = (\mu_B - \mu_A)\Sigma_B(\mu_B - \mu_A)^T$ are the two non-centrality parameters of the two non-central F-distributions used to compute the statistical power for the two tests.

Attention: The Hotelling t^2 test has very low performance when the number of archetype for each valid cluster is very close to the number of points used to define the radial distribution, i.e. when $N \approx N_\rho$. Moreover the centred F-distribution is not defined for $N_\rho > N$. In *RotationalSimilarityIdentifier* the user set the value N_ρ when the shape of the archetype in spherical coordinate is declared, i.e. choosing the `spherical_coordinates_shape`. The number of archetype N can be selected by changing the `N_archetype` parameter of the *ArchetypeIdentifier* operation.

At this point in bmmltools the *identification probability between the cluster A and B* $P(A = B)$ is defined as

$$P(A = B) = \text{minimum}(\gamma_{AB}, \gamma_{BA})$$

and while the *power of the statistical test* β is equal to

$$\eta = \begin{cases} \eta_{AB} & \text{if } \gamma_{AB} < \gamma_{BA} \\ \eta_{BA} & \text{otherwise.} \end{cases}$$

In both cases one can see that the result of the whole test is symmetric if A and B are exchanged. This has been chosen in order to ensure the symmetry of the equivalence relation among the two clusters which can be derived from $P(A = B)$ (see below).

Note: The Hotelling t^2 tests used here are one-sample. Alternatively one can use the two-sample Hotelling t^2 test, but this is currently not implemented in any bmmltools operation.

It is reasonable to expect that two or more clusters are identified if their identification probability is *above a certain threshold* p_{TH}, i.e. define a *candidate* equivalence relation eq. as

$$A \text{ eq. } B \iff P(A = B) > p_{TH}.$$

Despite this definition seem reasonable, things are not so straightforward, in particular when the relation defined above involves more than 2 clusters. In this case additional constraints need to be fulfilled in order to speak about real equivalence among the clusters. For example, consider 3 clusters A,B,C such that $P(A = B) > p_{TH}$, $P(B = C) > p_{TH}$ but $P(A = C) < p_{TH}$.Naively, one could conclude from the first two that A eq. B, B eq. C but A not eq. C. This is clearly not possible since if A eq. B and B eq. C for a true equivalence relation among the three clusters one would expect A eq. C, which contradicts the identification criteria chosen since $P(A = C) < p_{TH}$. This is the reason why eq. is said 'candidate'. Indeed one needs to check that the relation obtained by thresholding the identification probabilities, is a real equivalence relation in the sense of the precise mathematical definition, which is reported below.

> **Definition**: Given a set U a binary relation between two of its elements $x, y \in U$, in symbol x Rel. y, is an equivalence relation if:
>
> - x Rel. x (reflexivity),
> - x Rel. y implies y Rel. x and viceversa (symmetry),
> - if x Rel. y and y Rel. z then x Rel. z (transitivity),
>
> where $x, y, z \in U$.

In the case considered here the relation eq. defined via the identification probability is reflexive (it is easy to see that $P(A = A) = 1$ always) and symmetric by construction, as already observed. Transitivity need to be checked.

In bmmltools transitivity is checked by using the following idea. First, one constructs the graph G having a vertex for each cluster and an edge between two vertices if and only if their identification probability is above a certain threshold (i.e. the relation eq. defined above holds among the two considered clusters). Then the transitivity can be checked by looking at the adjacency matrix of the graph G, namely a matrix of $N_{vertices} \times N_{vertices}$ having 1 in the (i, j)-entry if and only if the vertex i is connected with the vertex j (note that the (i, i)-entry is always 1, because as noted above eq. is reflexive). Each connected component of G (and singleton of G too) defines a *candidate* equivalence relation in the sense of the definition given above. It is really an equivalence relation if and only if the adjacency matrix of the connected component has all the entries equal to 1, which would implies that the relation eq. holds among all the pairs needed to ensure transitivity.

In bmmltools the identification procedure described above is performed by the *RotationalSimilarityIdentifier* operation. According to the description given, the crucial parameters to set are:

- `p_threshold`, which corresponds to :math:`p_{TH}`.

- `spherical_coordinates_shape`, which determine the shape of the archetype in spherical coordinates, therefore the first element of the tuple corresponds to :math:`N_\rho`.
- `bin_used_in_radial_dist`, which is the minimum and maximum number of bins used for the for the statistical tests. This means that this parameter can reduce N_ρ, since it reduces the number of values in the (part of the) radial distribution used for the statistical test.

Attention: In the current implementation of the *RotationalSimilarityIdentifier* operation, in case of lack of transitivity for the candidate equivalence relation is fully disregarded and only for the pair with the highest identification probability the identification is suggested. How to deal this situation is arbitrary and different policy are possible (the one adopted at the moment is the most conservative one). This is another reason why the final result of the procedure explained in this tutorial can only suggest a possible identification.

The outputs of the *RotationalSimilarityIdentifier* are a matrix containing the identification probabilities for each pair of clusters, a matrix with the statistical power of each test used to produce the corresponding entry of the identification probability matrix and the identification matrix, where the suggestions about possible identification (based on thresholding of the identification probabilities and the transitivity check) are reported. In addition to these, also the (θ,ϕ) angles among each pair of clusters are reported (but clearly only the angles between cluster which are identifiable make sense). These angles are computed by taking the maximum of the weighted sum of the correlation computed in the $\theta\phi$-plane for different radii among the each cluster pairs. More precisely, let $\hat{\Phi}_A(k_\rho,k_\theta,k_\phi)$ and $\hat{\Phi}_B(k_\rho,k_\theta,k_\phi)$ be mean the modulus of the pDFT of the archetypes of two clusters A and B. The angles between the structure associated to the two clusters is computed according to the formulas below.

$$(\theta,\phi) = \text{argmax}_{\theta,\phi} \sum_k w_{A,k} \text{corr}\,[\hat{\Phi}_A,\hat{\Phi}_B](k)$$

where $\text{corr}[\hat{\Phi}_A,\hat{\Phi}_B]$ is the 2d correlation computed in the $\theta\phi$-plane of the mean archetype of the two clusters. The $w_{A,k}$ are weights which tends to suppress the correlation at those radii where $\hat{\Phi}_A$ does not vary too much in the $\theta\phi$-plane. This is done using the definition below.

$$w_{A,k} = \frac{1}{N}\sqrt{\frac{1}{N_\theta N_\phi}\sum_{k_\theta,k_\phi}\left(g^2_{A,\theta}(k,k_\theta,k_\phi)+g^2_{A,\phi}(k,k_\theta,k_\phi)\right)} \tag{12.12}$$

$$g_{A,\theta}(k_\rho,k_\theta,k_\phi) = \nabla_{k_\theta}\hat{\Phi}_A(k_\rho,k_\theta,k_\phi) \tag{12.13}$$

$$g_{A,\phi}(k_\rho,k_\theta,k_\phi) = \nabla_{k_\theta}\hat{\Phi}_A(k_\rho,k_\theta,k_\phi) \tag{12.14}$$

where N_θ and N_ϕ is the number of different values along the θ and ϕ axis of $\hat{\Phi}$, while N is such that $\sum_i w_{A,i} = 1$.

12.3 Example

The code below show how to use the two operations described above for the identification of clusters due to their similarity under rotation.

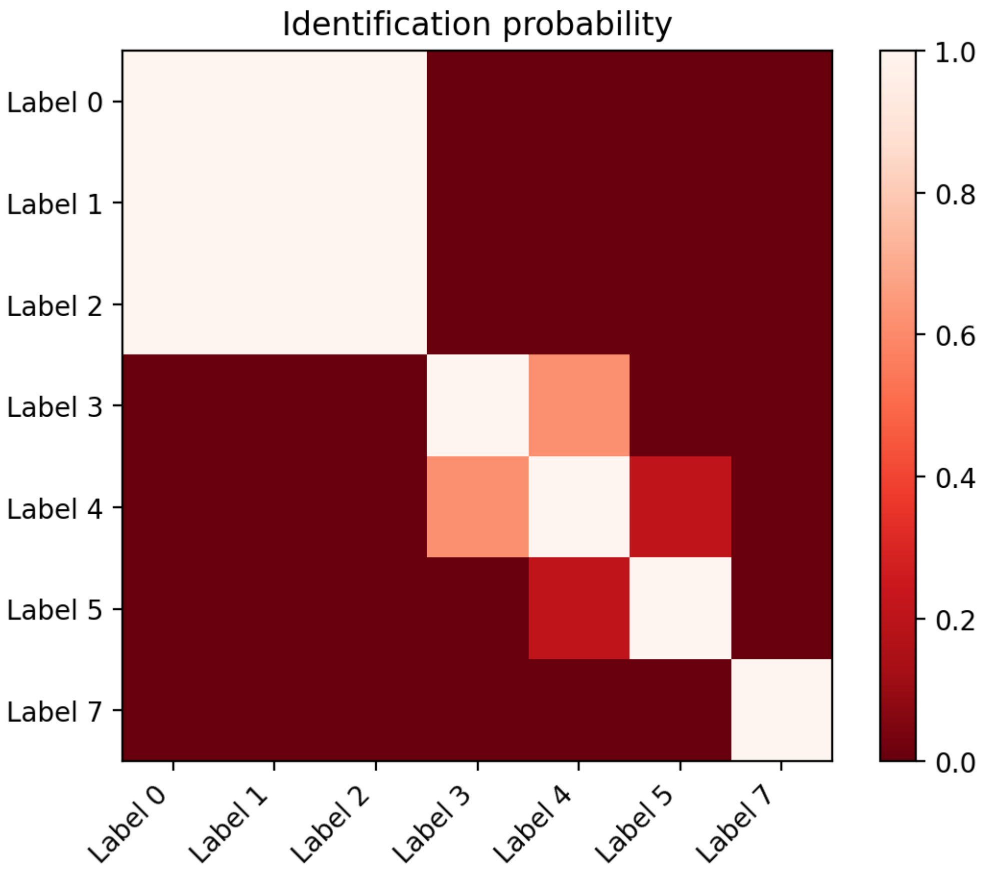
```
#...
from bmmltools.operations.clustering import ArchetypeIdentifier,
↪RotationalSimilarityIdentifier

# link a trace containing the ClusterValidator output.
# trace = ...

## segmentation pipeline
```

(continues on next page)

(continued from previous page)

```
x = ArchetypeIdentifier(trace)\
     .io('ClusterValidator OUTPUT DATASET NAME','archetype_dataset')\
     .apply(patch_shape=(50,50,50), extrapoints_per_dimension=(2,2,2), save_archetype_
↪mask=False)
x = RotationalSimilarityIdentifier()\
     .io(x+['Binarizer OUTPUT DATASET NAME',],'post_rsi_dataset')\
     .apply()

## intermediate result readings
RotationalSimilarityIdentifier(trace).o(x+['ClusterValidator OUTPUT DATASET NAME',]).
↪read()
```

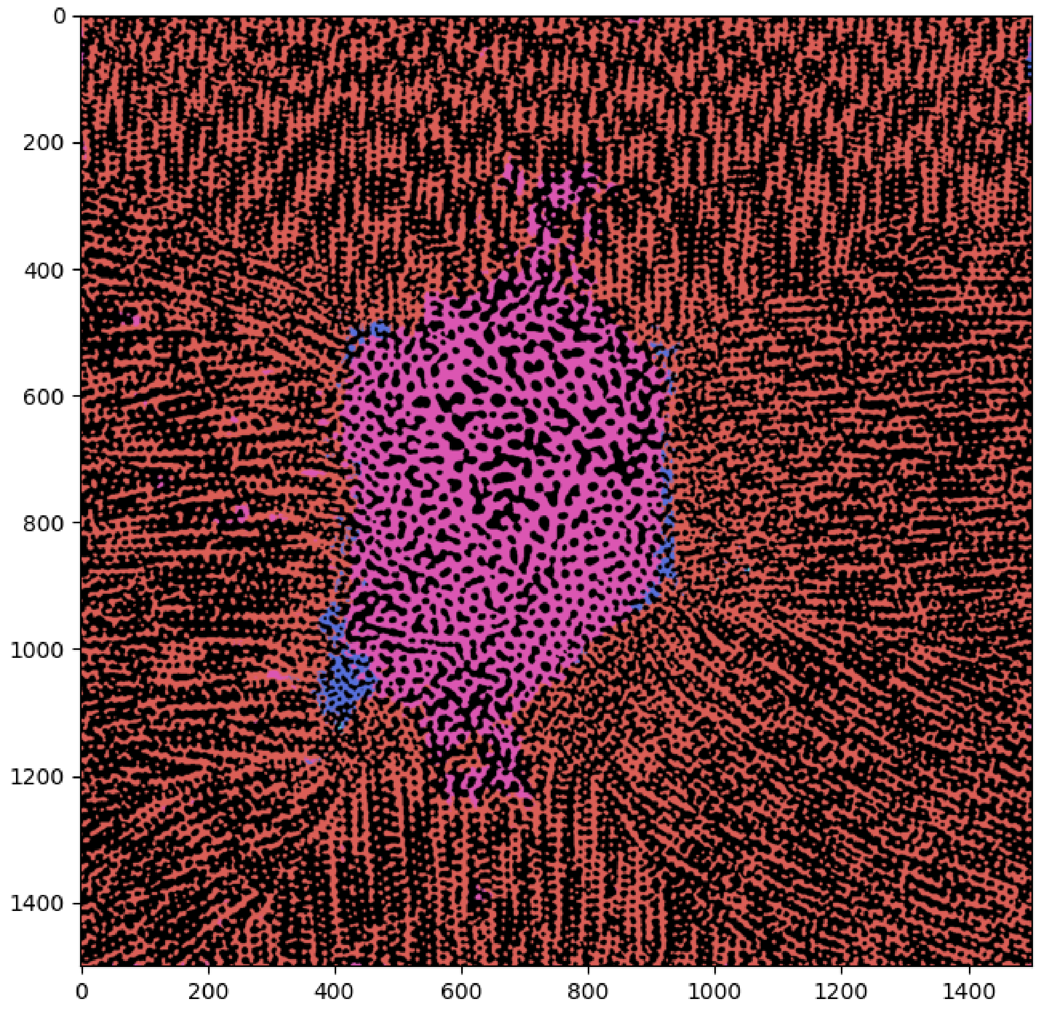
Continuing the *example of the "Case of study: segmentation pipeline"*, the following identification probabilities can be obtained

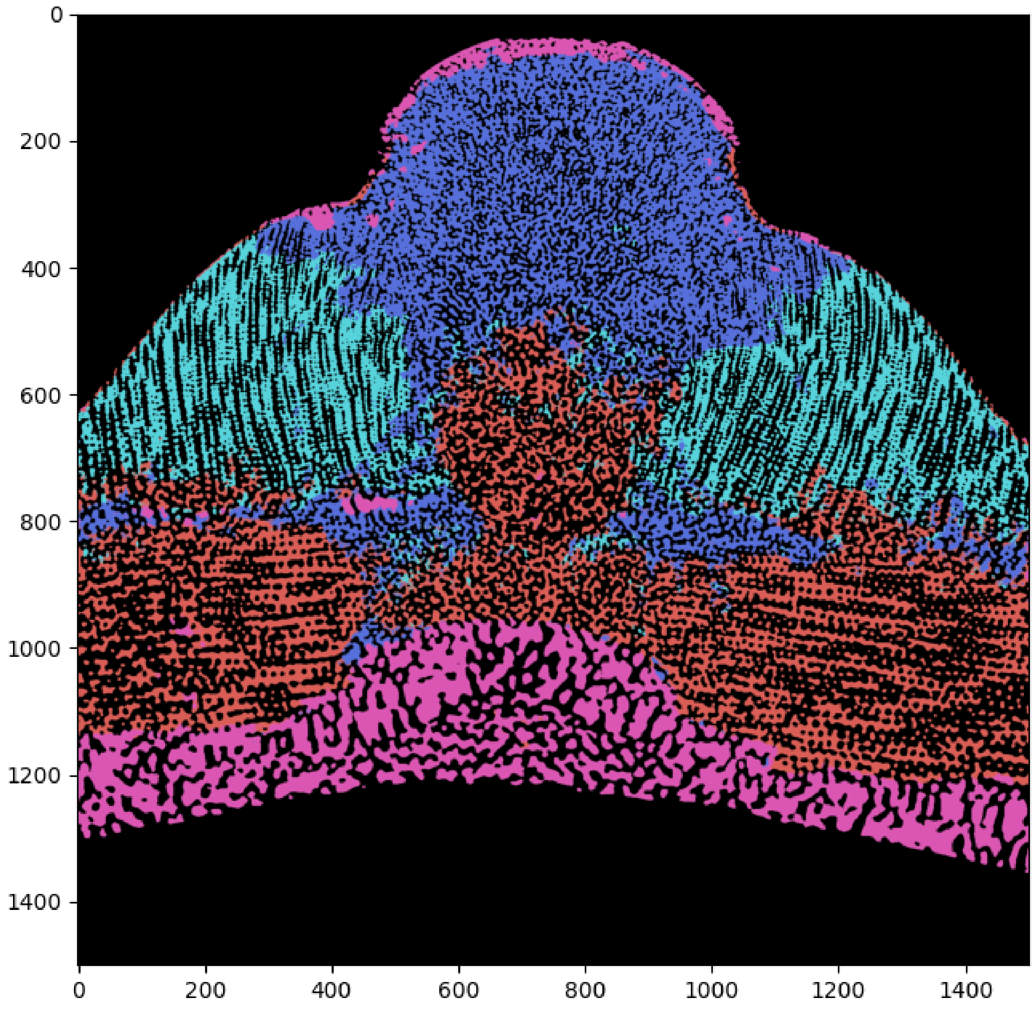


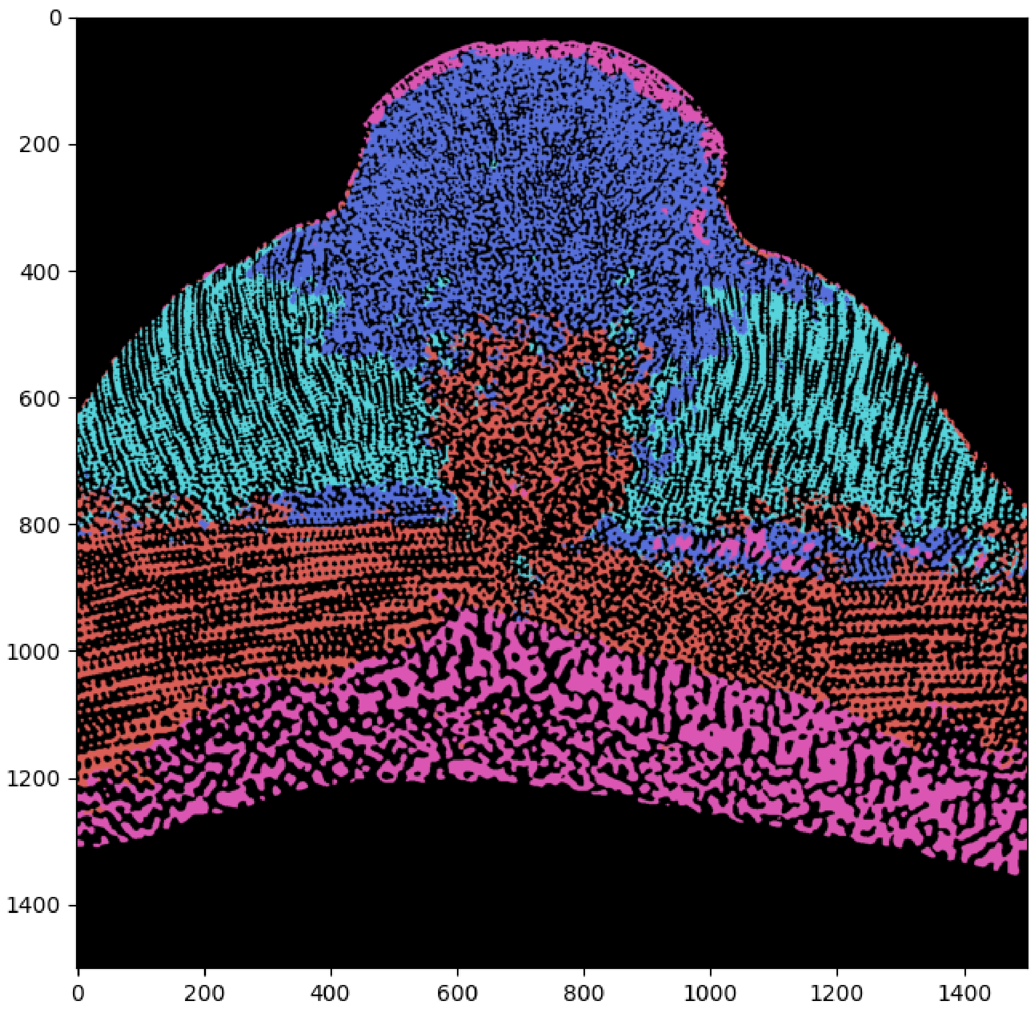
By checking transitivity of the candidate equivalence relation obtained by thresholding (with a threshold of 0.6) these identification probabilities, the following equivalence relation is suggested:

- Label 0, Label 1 and Label 2 can be grouped in a unique cluster;
- Label 3 and Label 4 can be grouped in a unique cluster.

By using these identification, on the final segmentation obtained one obtains




0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400


0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400

CHAPTER

THIRTEEN

CASE OF STUDY: EXPLANATION PIPELINE

By using the segmentation pipeline clusters are obtained, however it is not at all clear according to which criteria are they defined. This is typical of machine learning models, which are often regarded as black-boxes taking something as input and outputting the desired result (if the model is good enough). The situation is even worst in case of unsupervised machine learning models, as the one used in the segmentation pipeline due to the requirements under which the segmentation problem was solved. In this tutorial, a methodology to try to understand how the clusters found can be defined is discussed, so that the user can use this knowledge in order to judge if a certain cluster is reasonable or not. In this sense, the validation of the segmentation model can be done through explanation.

Note: Recently it has been observed the lack of reproducibility of many works applying machine learning results to different fields, like medicine, bioinformatics, neuroimaging, etc… . The *segmentation pipeline proposed in another tutorial* of this documentation would trivially fall in one of the data leakage case analysed there, since no test set is available. However the lack of a test set to evaluate the pipeline performance is a common situation for the kind of data which are typically used in 3d segmentation problems of large biological samples. Indeed the goal of a segmentation pipeline is used exactly to avoid to label the image by hand, since it is a time consuming and non-trivial operation due to the 3d nature of the problem. In this sense the "validation through explanation" approach of the segmentation result, can be seen as a way to evade the lack of test set problem, since the user can decide if the various components found by the segmentation pipeline are reasonable or not based on its knowledge in the field.

13.1 General strategy

The general strategy used to obtain an explanation is the one to train in a supervised manner a classifier to identify the various cluster found in the segmentation pipeline from a dataset of features, whose meaning is known and clear to the user. At this point explainable AI techniques are used in order to understand the relation existing between the model output and the input features, so that the identification of a given cluster is explained in therm of (a subset of) input features. These explainable AI techniques are often based on the assignment of a score to the model inputs, so that when the value of this score is close to zero, this feature does not contribute in significant manner to the model output. From scores of this kind it is possible to obtain a definition of the clusters found by the segmentation pipeline in terms of the input features used.

To reduce the model dependency of the explanation, the explanation results are not obtained from a single model, but considering an ensemble of different models (more precisely by averaging the scores assigned to a feature coming from different models).

13.2 Dataset and its preparation

The input dataset for the explanation pipeline is constructed by computing a series of features having a clear meaning for the user for a sequence of points/patches of the dataset used as input in the segmentation step. For example one can divide the 3d binary image to segment according to a regular grid whose unit element is a chosen patch (it can be a good idea to use the same kind of patch used in the segmentation pipeline). Then compute all the features for each patch: the results organized in table where the coordinates of the patch, the value of all the features, and the corresponding label are present. This table can be considered as an example of input dataset for the explanation pipeline

Which kind of feature should one compute? The answer to this question is problem dependent, and it is something that the user need to specify according to its knowledge on the sample under study or similar problems. By the way, these features need to fulfill certain requirements in order to be useful:

- they need to have a *clear meaning for the user*: the final explanation of the clusters identified will be in terms of these features and if the meaning is not clear to the user within the context of the problem under study, they are useless.
- they need to be *numerical*: this means that they are number the user know how to compute. For categorical variables, some form of encoding need to be used, like for example by using dummy variables.

Note that it is not necessary that all the features listed are useful for the description of a certain cluster. In this sense the user *does not have to decide* which are the most relevant features among the one used to typically describe a certain sample: this is a result of the procedure explained in this tutorial. In this sense, this "feature specification step" should be rather simple and not particularly demanding for the user: it should be enough to specify and compute the set of quantities typically used in problems similar to the one for which the sample is analysed.

The first thing to check is if the features selected are too much linearly correlated. Linear correlation among the input features of a model may helps during the training but it can be misleading when one tries to find a relation between the same input feature and the model output. If one tries to assign to some input feature x some score to deduce the input feature-output relation, the score can be arbitrarily distributed among scores of all the other feature which are linearly correlated with x, making the score and any interpretation based on that ambiguous. To avoid this problem, the amount of linear correlation (also called *multicollinearity*) need to be reduced.

To detect the presence of multicollinearity an effective method is the one of computing the *variance inflation factor* (VIF) of each feature [Snee1981] [Obrien2007]. Given the i-th feature, this quantity is defined as

$$VIF_i = \frac{1}{1-R_i^2}$$

where R_i^2 is the R^2 coefficient of a linear regression which use all the feature in the dataset except the i-th one, which is used as target of the regression. The VIF is a number between 1 and $+\infty$: it is equal to 1 when the considered feature is not linearly correlated to any other feature present in the dataset, while it goes towards infinity as the linear correlation increase.

Therefore putting a threshold on the maximal allowed VIF for a given dataset can be a way to quantify the maximal amount of liner correlation among the features in the dataset. Note that the VIF of one feature depends on all the feature present in the dataset when is computed. Keeping this in mind, the multicollinearity in the dataset can be reduced with the simple procedure below:

1. Chose a threshold VIF_{TH} value for the VIF;
2. Compute the VIF for arr the features;
3. If the VIF of some feature is above VIF_{TH}, remove the feature with the biggest VIF.
4. If a feature was eliminated in the previous step, repeat the step 2 and 3.

The set of feature surviving to this procedure generate a dataset having a lower amount linear correlation. One can say that the various feature are almost linear independent among each other, in the sense of linear algebra, which means that the information contained in the dataset can be approximately well represented in a vector space having dimension equal to the number of surviving features.

The simple procedure described till now is implemented in the *MultiCollinearityReducer* operation. The most important parameters of this operation are.

- `data_columns`, where one specify which columns of the input dataset need to be considered as input features (note that in this operation one need to specify also the column of the target in `target_columns`).
- `VIF_th` which is the VIF threshold (typical values are 5 or 10).

Note: Other output of this operation can be quite interesting: the so called linear association. It can be obtained by setting `return_linear_association = 'full'` or `return_linear_association = 'pairwise'`, depending if the *full linear association* or the *pairwise linear association* is required. With these two setting, in the reading folder of the trace a dictionary is saved in json format.

In the full linear association for each eliminated feature, the coefficients of the linear model describing the eliminated feature are stored in the dictionary saved. More precisely, if Y_k is the eliminated feature and $X_0, \cdots, X_n$ are the n surviving features, then in the full linear association dictionary for Y_k the coefficients $w_{k,0}, \cdots, w_{k,n}, q_k$ of the linear model

$$Y_k = w_{k,0}X_0 + \cdots + w_{k,n}X_n + q_k,$$

are saved.

On the other hand, in the pairwise linear association for each eliminated feature, the coefficients of the linear models describing the eliminated feature one of the surviving features are saved. This means that pairwise linear association dictionary for each eliminated feature Y_k the coefficients $w_{k,i}, q_{k,i}$ of the linear models

$$Y_k = w_{k,i}X_i + q_{k,i},$$

for any surviving feature X_i, are saved.

Note that, since these linear models are produced using as inputs features with a low level of multicollinearity, the coefficients of the linear models can be used to rank which of the surviving feature can be *substituted* with one of the features eliminated, without altering too much the maximal VIF of the dataset.

13.3 Tools for the explanation

The basic idea behind this explanation pipeline is to train in a supervised setting a classifiers, from a set of meaningful features to predict the clusters found by the segmentation pipeline, and get the explanation of the clusters by explaining the classifiers decision in terms of the input feature. The classification problem should have certain features:

- for each cluster one should train a different ensemble of classifiers: the kind of classifier, hyperparameters, etc… should be all the same for all the cluster but vary within the ensemble associate to it.
- classification problem should be binary, i.e. the classifiers for a given label should be trained to predict 1 when the classifier input features corresponds to label and 0 otherwise.

The first requirement is needed in order to have an explanation that is tailored for each label, and the use of an ensemble rather than a single model is done in order to reduce a possible model dependency in the explanation obtained, as mentioned above. The second requirement is needed for the interpretation of the scores, as will be clear in the next section.

From a practical point of view, the model selected should have enough “power” to work well for generic classification problem: neural networks or random forests seem to be good choice (despite simple system can be used). Another practical requirement is that the amount of computational resources needed for the training il modest, since training happens for each label and for each model of the ensemble. This last requirement favor random forest over neural networks.

In bmmltools random forest are used, and ensembles are created by training random forests with different hyperparameters combinations.

Note: As will be clear from the next section, it is fundamental to check if the model generalize correctly or simply overfit over the training dataset. To check that first the full dataset of input feature need to be split into a training and validation dataset. The classifier is then trained with a shuffled stratified K-fold cross-validation in order to select the best model, whose performance can be than evaluated with the validation dataset.

The explanation is derived by computing certain scores from the trained ensemble of classifiers. In the explanation pipeline presented in this tutorial, various tools turns out to be particularly useful: the classifier *accuracy* and *F1-score*, the *permutation importance*, and the *partial dependency* for each features. How they are defined is briefly reported below, while how to use them to get an explanation will be discussed in the next section.

Accuracy

The Accuracy is a simple and popular metric to evaluate binary classifier. In the case considered here, it is used to evaluate the performance of the trained models. In a binary classification problem, the following considered:

- *true positive*, TP, i.e. the number of all the examples in the test dataset which are labeled by 1 and the classifier correctly predict as 1;
- *true negative*, TN, i.e. the number of all the examples in the test dataset which are labeled by 0 and the classifier correctly predict as 0;
- *false positive*, FP, i.e. the number of all the examples in the test dataset which are labeled by 1 but the classifier predict them as 0;
- *false negative*, FN, i.e. the number of all the examples in the test dataset which are labeled by 0 but the classifier predict them as 1;

The accuracy is defined as

$$\text{Acc} = \frac{TP+TN}{TP+TN+FP+FN}$$

which is the ratio between number of correctly classified examples and the total number of examples in the dataset. It can take values between 0 and 1, reaching 1 in case all the examples are correctly classified.

Suppose to have a dataset having an equal number of example corresponding to the label 0 and the label 1, then a classifier which output at random 0 or 1 whatever is the input (therefore a very bad classifier), would have an accuracy of 0.5, since on average half of the time will predict the correct result. This would not be so, if the dataset is not balanced, which makes the interpretation more tricky (the very bed classifier would have an accuracy equal to the fraction of label 1 present in the dataset, inflating or contracting the accuracy). Therefore, in this case it is better to use the *balanced accuracy*, which is defined below

$$\text{bal-Acc} = \frac{1}{2}\left(\frac{TP}{TP+FN}+\frac{TN}{TN+FP}\right)$$

which would return 0.5 for the random classifier even for unbalanced dataset. Since the random classifier is a very bad case, one may want that the "adjust" the balanced accuracy values, so that the random classifier case takes value 0. This is the so called *adjusted accuracy*, which is simply

$$\text{adj-Acc} = 2\text{bal-Acc} - 1$$

The adjusted accuracy is also called *informedness* or *Youden's J statistic*.

F1-score

Recalling the previous classification of the prediction of the classifier, the F1-score is defined as the ratio below,

$$F_1 = \frac{2TP}{2TP + FP + FN},$$

and can be seen as the harmonic mean between the classifier precision and recall. Typically is more robust than the accuracy in quantifying the model prediction but is less transparent ot a direct interpretation.

Permutation importance

The permutation importance, PI, measure how the model performance changes when the value of a given feature is replaced with a random value *sampled* from the same distribution [Breiman2001]. The model performance are measured by means of a given metrics.

Let $X = \{X_n\}_{n=0}^{N-1}$ be the input dataset used to train a model f, and let $X_n = (x_{n,0}, \cdots, x_{n,m-1})$ be the n-th example composed by m features. Each example X_n can be seen as the result of the sampling of an m-dimensional probability distribution $p(x) = p(x_0, \cdots, x_{m-1})$ characterizing the dataset. *Assuming (statistical) independence among the input features* one can write

$$p(x_0, \cdots, x_{m-1}) = p_0(x_0) \cdot p_1(x_1) \cdot \ldots \cdot p_{m-1}(x_{m-1}).$$

To compute the PI of the k-th feature one as to evaluate the model using a series of examples $\tilde{X}_n = (x_{n,0}, \cdots, x_{n,k-1}, \tilde{x}_{n,k}, x_{n,k+1}, \cdots, x_{n,m-1})$, where $\tilde{x}_{n,k} \sim p_k(x_k)$, i.e. $\tilde{x}_{n,k}$ is sampled from the distribution of the k-th feature $p_k(x_k)$. The probability distribution $p(x)$ is not known, therefore to generate new samples for x_k one is forced to use the permutation trick. More precisely, given a permutation of the numbers $\{0, 1, \cdots, N-1\}$, σ, one can define $X^{(k-perm)} = \{X_n^{(k-perm)}\}_{n=0}^{N-1}$ as the dataset with examples $X_n^{(k-perm)} = (x_{n,0}, \cdots, x_{n,k-1}, x_{\sigma(n),k}, x_{n,k+1}, \cdots, x_{n,m-1})$. Note that by permuting the k-th feature one is effectively sampling $p_k(x_k)$. At this point, the PI of the k-th feature is defined as

$$PI_k = L(y, f(X^{(k-perm)})) - L(y, f(X))$$

where L is the selected metric, and $y = \{y_n\}_{n=0}^{N-1}$ are the targets corresponding to each example of the dataset. In bmmltools the permutation importance is computed according to the sklearn implementation uses the mean accuracy as metric for the model used as classifier (which is a Random Forest).

Note: Sometimes the PI can assumes negative values for certain feature. This means that the performance of the model increase if the noise is given as input rather than the actual value of the feature (clearly, the values of all the other features are given as input as well). This means that the feature considered actually confuse the model rather than help: the model would probably perform better, if this feature is removed from the inputs. When the dataset has a low level of multicollinearity, this problem should be minimal or nor happen at all.

Note: The assumption of independence of the feature, is not easy to ensure in general. This is why the permutation importance should be computed for model trained on dataset with no or low amount of multicollinearity: so that the correlation is reduced. Keep in mind that lack of correlation is a necessary but not sufficient condition for independence among features.

Partial dependency

To gain insight on the relation between the classifier prediction and the input features the partial dependencies (also called partial dependency plots [Friedman2001] [Molnar2022]) turns out to be very useful.

Given the input dataset $X = \{X_n\}_{n=0}^{N-1}$ composed by examples $X_n = (x_{n,0}, \cdots, x_{n,m-1})$ having m features, consider a subset of indices $S = \{\alpha_1, \cdots, \alpha_p\}$, where $\alpha_i \in I = \{0, 1, \cdots, m-1\}$ and its complement with respect to the set of all indices :math:'S^c = I/S'. Since to each index k corresponds to a feature, one can split each example X_n as

$$X_n = (x_n^S, x_n^{I/S}) \tag{13.1}$$

$$x_n^S = (x_{n,\alpha_1}, \cdots, x_{n,\alpha_p}) \text{ with } \alpha_i \in S \tag{13.2}$$

$$x_n^{I/S} = (x_{n,\beta_1}, \cdots, x_{n,\beta_{m-p}}) \text{ with } \beta_i \in I/S \tag{13.3}$$

which effectively split the input feature on which the model is trained in two group. The partial dependency of a model f for the features corresponding to S, is defined as the expectation value of the marginal probabiltiy distribution of the I/S features, namely

$$PD_S(z_1, \cdots, z_p) = E_{x^{I/S}}[f(z_1, \cdots, z_p, x^{I/S})] \tag{13.4}$$

$$= \int f(z_1, \cdots, z_p, x^{I/S}) p(x_{\beta_1}, \cdots, x_{\beta_{m-p}}) dx_{\beta_1}, \cdots, dx_{\beta_{m-p}} \tag{13.5}$$

where $(z_1, \cdots, z_p)$ is possible value of the features in S. The probability distribution association to the dataset is typically unknown, and so is its marginal, therefore further assumptions need to be done. *Assuming again (statistical) independence among the input feature* one can write that

$$p(x_{\beta_1}, \cdots, x_{\beta_{m-p}} = p_{\beta_1}(x_{\beta_1}) \cdot \ldots \cdot p_{\beta_{m-p}}(x_{\beta_{m-p}})$$

which allow to estimate the partial dependency from the dataset directly, since the integral in this case can be approximated with the empirical average over the I/S features, namely

$$PD_S(z_1, \cdots, z_p) \approx \frac{1}{N} \sum_{n=0}^{N-1} f(z_1, \cdots, z_p, x_{n,\beta_1}, \cdots, x_{n,\beta_{m-p}})$$

where $x_{n,\beta_1}, \cdots, x_{n,\beta_{m-p}}$ are the values assumed by the I/S features in the example n, while $z_1, \cdots, z_p$ is the point in which the partial dependency is computed.

Particularly interesting for this tutorial is the case where S contains a single feature, which is here called *partial dependency of the feature* k, which is simply computed as

$$PD_k(z) \approx \frac{1}{N} \sum_{n=0}^{N-1} f(x_{n,0}, \cdots, x_{n,k-1}, z, x_{n,k+1}, \cdots, x_{n,m})$$

where z varies over the range of the feature in the dataset, i.e. $z \in [\min_n x_{n,k}, \max_n x_{n,k}]$. It is interesting to observe that the relation detected from the partial dependency admit a causal interpretation provided that certain additional requirements are satisfied by the features [Qingyuan2021].

Note: The assumption of independence among features is the one allowing to compute the PD of a given feature as an empirical average. That is why one should be computed it for model trained on dataset with no or low amount of multicollinearity: so that the correlation is reduced, which is a necessary (but not sufficient) condition for statistical independence. It is known that the PD works well when the features are not correlated among each other [Molnar2022].

13.4 Interpreting F1 scores, PI and PD

The first to do once that a classifier of the ensemble associated to a given label is trained is to check if the model are able to generalize on the validation set or not. This is measured by the F1 score, which should be close to 1 in order to be sure that the model correctly generalize. In bmmltools, for an ensemble of classifier the average F1 score is be used rather than the individual F1 scores.

This is very important since if the model overfit on the training, the explanation obtained does not make sense. The lack of generalization or more generally a small value of the F1 score can happen essentially for two reasons (assuming the model used is powerful enough):

1. The cluster does not make sense, therefore the model is not able to find any relation between the input feature and the given label;
2. The cluster make sense but the input feature are not able to describe it.

Clearly a low F1 score alone is not enough to conclude that a clustering does not make sens, by the way if the user is for various reason certain about the "goodness" of the features used, this can be a strong indication that the clusters whose F1 score is low, are probably not real clusters.

Attention: Before to completely rule out a cluster, it can be a good idea to enlarge/change the set of features used as input features for the explanation pipeline and check if the F1 score still remain low.

If there are no reason to invalidate the models ensemble associated to a cluster, the next step to get an explanation is to look at the PI of the features. Feature with an high permutation importance are likely to be the most influential features of the model, i.e. change their value would likely to change in a significant manner the model output (measured via a suitable metric). Naively, one can say that, highest is the PI of a feature and more this feature is relevant for the model in order to recognize the corresponding cluster.

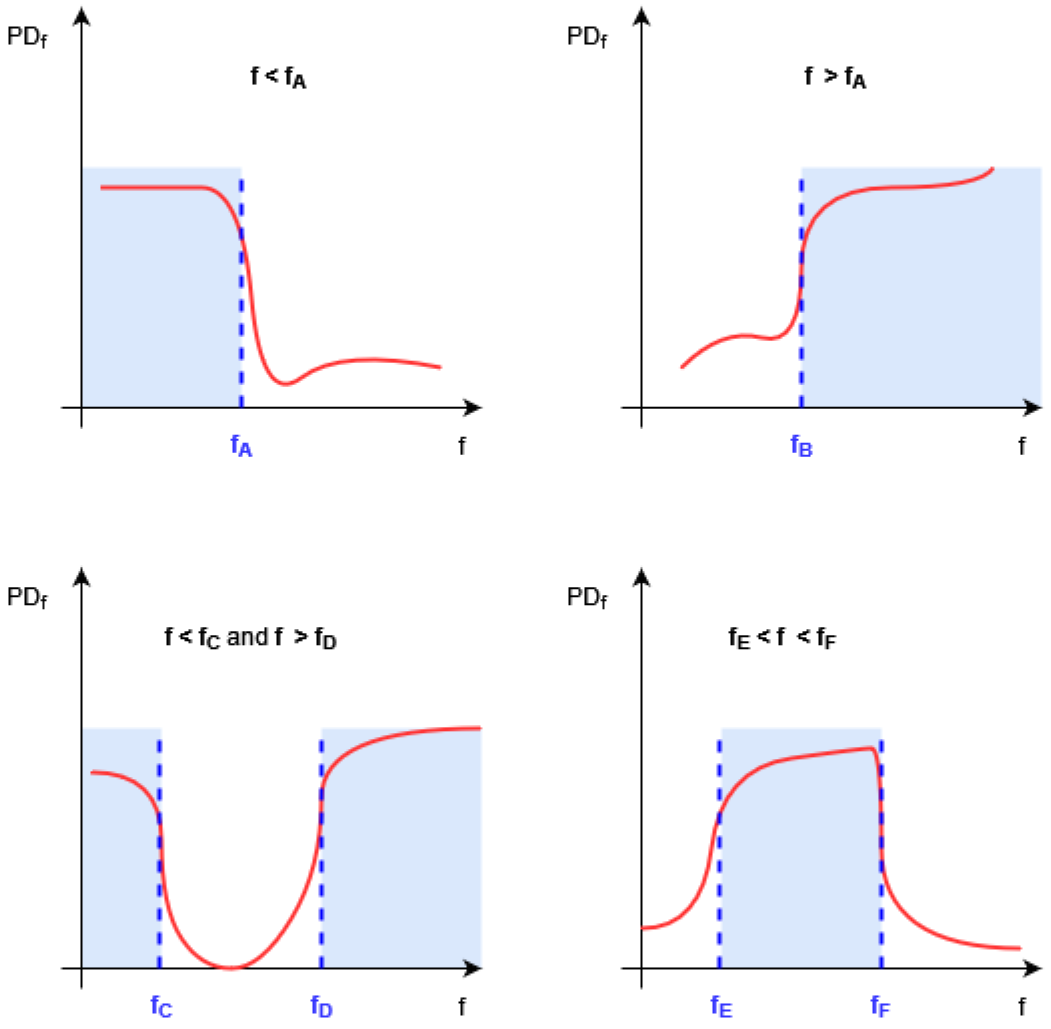
A simple criteria to select the most relevant feature is to sort the feature according to its permutation importance in decreasing order. By selecting a given threshold TH, one can keep the first $M < p$ sorted features such that

$$\frac{1}{N}\sum_{k=0}^{M-2} PI_{\text{sorted }k} \leq TH \tag{13.6}$$

$$\frac{1}{N}\sum_{k=0}^{M-1} PI_{\text{sorted }k} > TH \tag{13.7}$$

where $N = \sum_{k=0}^{p} \max(0, PI_{\text{sorted }k})$ is a normalization constant such that the sum of the positive permutation importance is equal to 1. In this way one can interpret $100 \cdot TH$ as the percentage of positive permutation importance explained by the M selected features. The features selected in this way are the one that will be used to explain the cluster.

Once that the most relevant features has been identified, the next step is to find the interval of values which can be used to define the cluster. This can be done by looking at the partial dependency of the feature. Recalling that the model outputs 1 when the cluster is detected, and that the partial dependency capture how the model output changes on (partial) average as the feature value change within its range, one can easily define the characteristic interval for the feature value as the region of the feature range where the partial dependency is high with respect to the surrounding values. Consider the examples in the picture below



In the example some common behaviors of the partial dependency for some feature is showed. According to the principle explained above, the regions where the partial dependency is sufficiently high with respect of its surrounding, is the region one is looking for to explain the cluster.

In the first two graph above, describe the situation where a single threshold is needed to describe the interval, namely situation like $[\min_n f_{n,k}, f_{th}]$ or $[f_{th}, \max_n f_{n,k}]$. In the second line, two threshold are needed, i.e. for intervals like $[f_{th_1}, f_{th_2}]$ or $[\min_n f_{n,k}, f_{th_1}] \cap [f_{th_2}, \max_n f_{n,k}]$. Clearly this discussion can be generalized easily to more complex intervals. Note that all the intervals are always limited to the feature range, i.e. $[\min_n f_{n,k}, \max_n f_{n,k}]$, since nothing can be said outside this range.

At this point the proper explanation of the cluster can be obtained. Let L be the cluster label and assume that $f_1, \cdots, f_M$ are the most relevant feature found using the PI. Let $\Delta_1, \cdots, \Delta_M$ be the intervals found using the PDs of each relevant feature. Then the explanation of the considered cluster is:

$$\text{cluster L} \iff f_1 \in \Delta_1, \cdots, f_M \in \Delta_M.$$

Defining a cluster in this way allow the user to judge if the cluster make sense or not for its specific problem by using its specific knowledge of the problem.

It is also possible to assign a score to the explanation obtained. This can be trivially done constructing an *if-else classifier* for each cluster, which simply classify a given input as belonging to a given cluster if and only if the most relevant features fall in the intervals corresponding to the considered cluster, and evaluate the performance of the classifier using some metric, since the ground true result are known. In bmmltools, the balanced accuracy (or the adjusted one) is used to evaluate the "goodness" of the explanation obtained, since it is easier to interpret (with respect to the F1 score).

Almost all the thing described till now are performed in bmmltools by the *ExplainWithClassifier* operation. According to the discussion done till now, the main parameters to set are:

- `test_to_train_ratio`, which is the fraction of sample in the input dataset which is used as validation set;
- `n_kfold_splits`, which is the number of K-fold splits used to find the best classifier for a given hyperparameter configuration which is used for the validation step (i.e. the computation of the average F1 score);
- `n_grid_points`, which is the number of points in the feature range used to evaluate the partial dependency of a feature.

This operation outputs the PIs and PDs for each label which is stored on the trace. They can be exported from the trace by using the `.read()` of course, but with the option `save_graphs = True` the graphs of the PIs and all the PDs (the mean F1 score is printed in the title of the PIs graphs).

For cases showed in the picture above, the intervals can be derived in automatic manner and optimized using a suitable bmmltools operation: *InterpretPIandPD*. More precisely, this operation performs for all the clusters the following tasks:

1. Select the first M most relevant feature according to the simple criteria explained above;
2. For each of the selected features, the intervals which can be used to define a given cluster;
3. If required by the user, the intervals derived in the step 2 are optimized using bayesian optimization in order to maximize the balanced accuracy of the if-else model;
4. Compute the balanced or adjusted accuracy of the if-else model to evaluation the goodness of the explanation obtain;

In light of the discussion done till now, the main parameters to set in this operation are:

- `positive_PI_th`, which correspond to the threshold used to select the M most relevant features;
- `adjust_accuracy`, if True evaluate the if-else classifier with the adjusted accuracy instead of the balanced accuracy;
- `bayes_optimize_interpretable_model` if True perform the bayesian optimiation of the intervals briefly described in the point 3 above.

13.5 Example

In this example, the explanation for the clusters obtained in the *segmentation pipeline proposed in this documentation* after *identification of the ones that are similar under rotation*, is derived. The features used fro the explanation are 22 and listed below.

- vf, volume fraction.
- sa, surface area.
- ssa, (sort of) specific surface area in m2 cm-3 of material.
- sad, surface area per mu^3 (patch volume).
- tmt, trabecular minimum thickness.
- $tmtsd$, trabecular minimum thickness standard deviation.

- tth, trabecular thickness (um), i.e. average thickness weighted by the length of the trabecular.
- $tthsd$, trabecular thickness standard deviation (um).
- bml, beam mean length, i.e. mean length of the skeleton branches (um) excluding those going through
- $bmlsd$, beam mean length standard deviation.
- bmt, beam mean type of those beams not going through the boundaries (0 - endpoint to endpoint, 1 - endpoint to joint, 2 - joint to joint, 3 - circular).
- bd, number density of beams (N/mu^3).
- mjt, mean joint connectivity.
- $mjtsd$, joint connectivity standard deviation.
- nj, number density of joint (with connectivity > 2) (N/mu^3).
- epd, endpoints density (connectivity = 1) (N/mu^3).
- mt, minimum trabecular thickness along a beam.
- $phim$, mean of the beam phi angles (in spherical coordinates).
- $phisd$, standard deviation of the beam phi angles (in spherical coordinates).
- $thetam$, mean of the beam theta angles (in spherical coordinates).
- $thetas$, standard deviation of the beam theta angles (in spherical coordinates).
- gc, integral gaussian curvature.
- $gcsd$, standard deviation of the integral gaussian curvature.
- mc, mean curvature.
- $mcsd$, standard deviation of the mean curvature.

The code below show the pipeline used to get the explanation, constructed according to the principles described above.

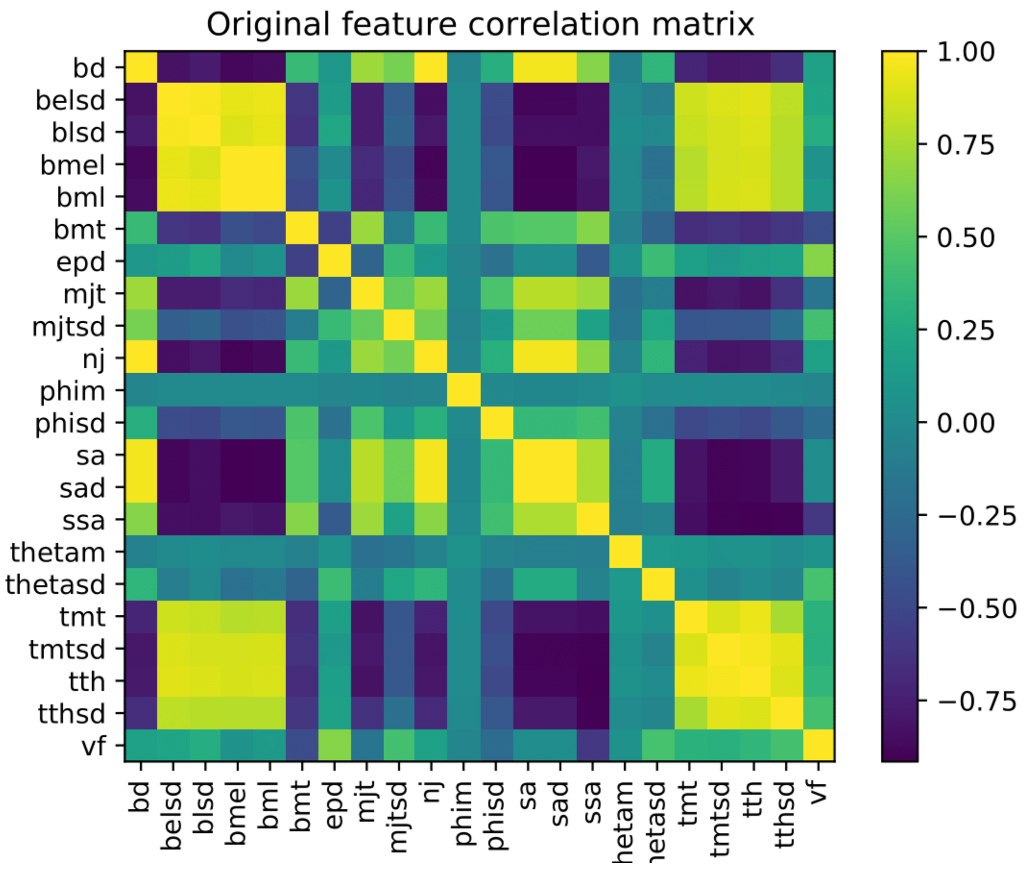
```
from bmmltools.core.data import Data
from bmmltools.core.tracer import Trace
from bmmltools.operations.io import Input
from bmmltools.operations.explanation import MultiCollinearityReducer,
↪ExplainWithClassifier,InterpretPIandPD

#### MAIN


## Load data
data = Data(working_folder=r'ml_explain3/test/data2')
data.load_pandas_df_from_json(r'ml_explain3/skan_features.json','skan_features',drop_
↪columns=['cube_space_coord'])

## create a trace
trace = Trace()
trace.create(working_folder=r'ml_explain3/test/op',group_name='explainer')

## machine learning model
x = Input(trace).i('skan_features').apply(data)
x = MultiCollinearityReducer(trace)\
        .io(x,'post_mcr_dataset')\
```

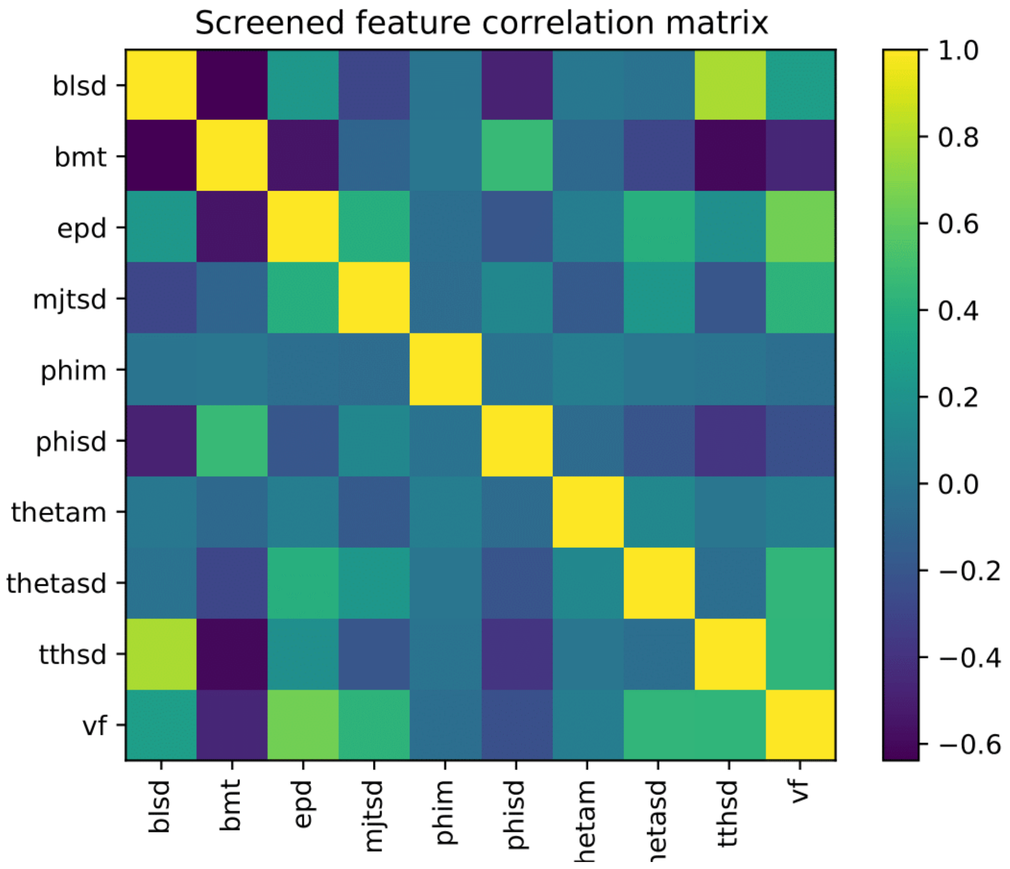
(continues on next page)

(continued from previous page)

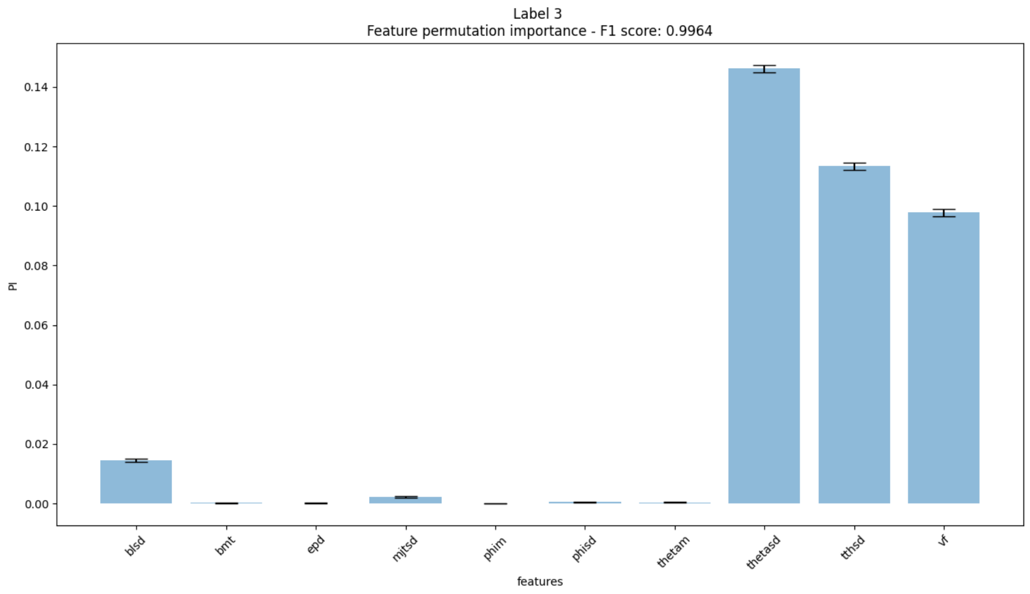
```
        .apply(data_columns = ['bd','belsd','blsd','bmel','bml','bmt','epd','mjt','mjtsd
↪','nj','phim','phisd',
                               'sa','sad','ssa','thetam','thetasd','tmt','tmtsd','tth',
↪'tthsd','vf'],
               target_columns = ['RI_label'],
               VIF_th= 5,
               return_linear_association='full')
x_ref = x
x = ExplainWithClassifier(trace).io(x,'post_ewc_dataset').apply(save_graphs=True)
x = InterpretPIandPD(trace)\
        .io(x+x_ref,'label_interpretation')\
        .apply(bayes_optimize_interpretable_model=True,save_interpretable_model=True)


## Result readings
MultiCollinearityReducer(trace).o('post_mcr_dataset').read()
ExplainWithClassifier(trace).o('post_ewc_dataset').read()
InterpretPIandPD(trace).o('label_interpretation').read()
```

The MultiCollinearityReducer is able to reduce the number of feature from 22 to 10: only 10 features are approximately uncorrelated. This can be checked by looking at the correlation matrix and it looks before and after the reduction of multicollinearity below.

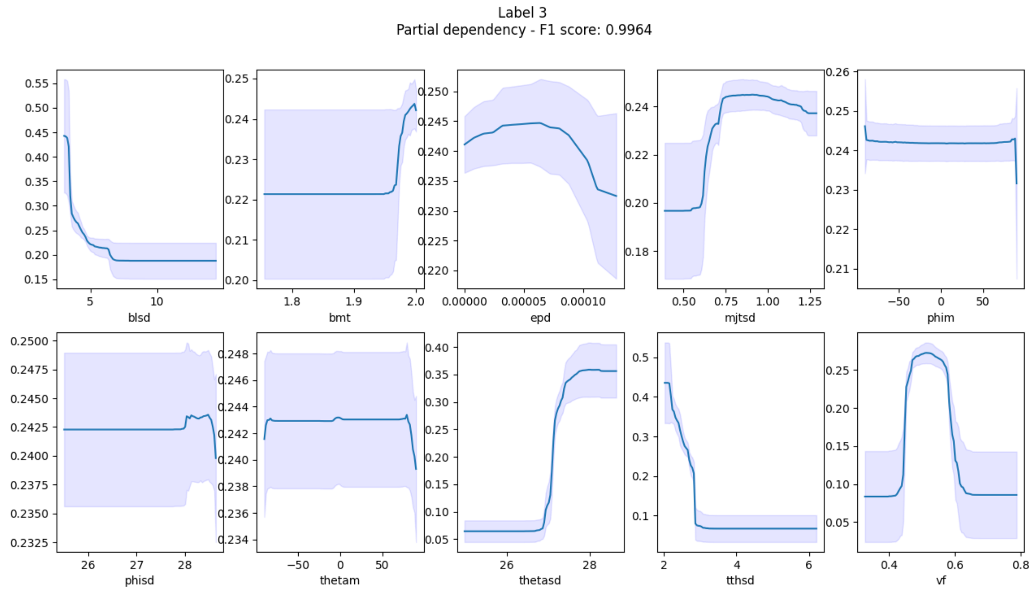




As example consider the cluster 3. The PIs for the various features are reported in the histogram below.



The most relevant feature explaining the 80% of the total positive PI are vf,:math:*tthsd*, and $thetasd$. The partial dependencies of the model are showed below.



By looking at the last three plots one can derive a definition for the cluster 3. Running the InterpretPIandPD operation, after bayesian optimization of the if-else classifier, the cluster 3 can be defined as follow.

$$\text{cluster 3} \iff 0.44 < vf < 0.61\ ,\ tthsd < 2.90\ ,\ thetasd > 27.1.$$

The explanation of all the other clusters can be found in similar manner and is summarized in the table below, where also the classification accuracy of the if-else classifier is reported.

Cluster	Definition	Balanced classification accuracy
0	$vf < 0.51\ ,\ tthsd < 3.63\ ,\ thetasd < 24.33$	0.92
3	$0.44 < vf < 0.61\ ,\ tthsd < 2.90\ ,\ thetasd > 27.1$	0.97
5	$0.54 < vf < 0.79\ ,\ blsd < 6.91\ ,\ thetasd < 28.35$	0.89
7	$blsd < 6.01\ ,\ tthsd < 3.19$	0.92

The fact that the cluster can be well explained using the definition above, is an indication that the cluster found by the segmentation pipeline proposed are reasonable.

13.6 References

CHAPTER FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Amerio1989] Amerio, Luigi. "Almost-periodic functions in banach spaces." in The Harald Bohr Centenary: proceedings of a Symposium held in Copenhagen April 24-25, 1987 - C. Berg and B. Fuglede - Matematisk-fysiske Meddelelser 42:3, 1989.

[DeBrunner2005] DeBrunner, Victor, et al. "Entropy-based uncertainty measures for $L^2(R^n)$, $l^2(Z)$, and $l^2(Z/NZ)$ with a Hirschman optimal transform for $l^2(Z/NZ)$." IEEE Transactions on Signal Processing 53.8 (2005): 2690-2699.

[Moisan2011] Moisan, Lionel. "Periodic plus smooth image decomposition." Journal of Mathematical Imaging and Vision 39.2 (2011): 161-179.

[McInnes2017] McInnes, Leland, and John Healy. "Accelerated hierarchical density based clustering." 2017 IEEE International Conference on Data Mining Workshops (ICDMW). IEEE, 2017.

[Campello2013] Campello, Ricardo JGB, Davoud Moulavi, and Jörg Sander. "Density-based clustering based on hierarchical density estimates." Pacific-Asia conference on knowledge discovery and data mining. Springer, Berlin, Heidelberg, 2013.

[Bourlard1988] Bourlard, Hervé, and Yves Kamp. "Auto-association by multilayer perceptrons and singular value decomposition." Biological cybernetics 59.4 (1988): 291-294.

[Chicco2014] Chicco, Davide, Peter Sadowski, and Pierre Baldi. "Deep autoencoder neural networks for gene ontology annotation predictions." Proceedings of the 5th ACM conference on bioinformatics, computational biology, and health informatics. 2014.

[Hinton2006] Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." science 313.5786 (2006): 504-507.

[Snee1981] Snee, Ron (1981). "Origins of the Variance Inflation Factor as Recalled by Cuthbert Daniel" (Technical report), Snee Associates.

[Obrien2007] O'brien, Robert M. "A caution regarding rules of thumb for variance inflation factors." Quality & quantity 41.5 (2007): 673-690.

[Breiman2001] Breiman, Leo. "Random forests." Machine learning 45.1 (2001): 5-32.

[Friedman2001] Friedman, Jerome H. "Greedy function approximation: a gradient boosting machine." Annals of statistics (2001): 1189-1232.

[Qingyuan2021] Zhao, Qingyuan, and Trevor Hastie. "Causal interpretations of black-box models." Journal of Business & Economic Statistics 39.1 (2021): 272-281.

[Molnar2022] C. Molnar, Interpretable Machine Learning (Second Edition), 2022