pyXpcm Documentation

Release 0.4.1

pyXpcm Developers

Feb 15, 2024

GETTING STARTED

pyXpcm is a python package to create and work with ocean **Profile Classification Model** that consumes and produces Xarray objects. Xarray objects are N-D labeled arrays and datasets in Python.

An ocean **Profile Classification Model** allows to automatically assemble ocean profiles in clusters according to their vertical structure similarities. The geospatial properties of these clusters can be used to address a large variety of oceanographic problems: front detection, water mass identification, natural region contouring (gyres, eddies), reference profile selection for QC validation, etc... The vertical structure of these clusters furthermore provides a highly synthetic representation of large ocean areas that can be used for dimensionality reduction and coherent intercomparisons of ocean data (re)-analysis or simulations.

CHAPTER

ONE

DOCUMENTATION

Getting Started

1.1 Overview

1.1.1 What is an ocean PCM?

An ocean PCM is a **Profile Classification Model for ocean data**, a statistical procedure to classify ocean vertical profiles into a finite set of "clusters". Depending on the dataset, such clusters can show space/time coherence that can be used in many different ways to study the ocean.

1.1.2 Statistic method

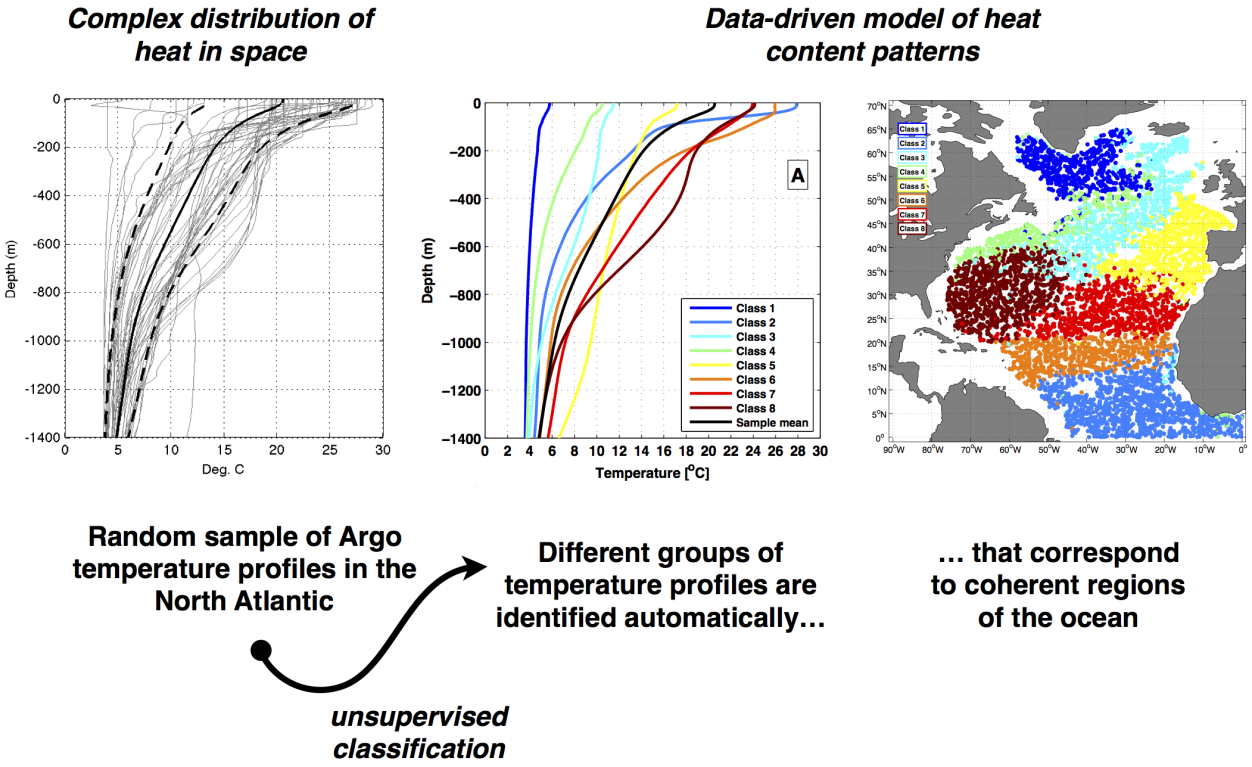
It consists in conducting **un-supervised classification** (clustering) with vertical profiles of one or more ocean variables.

Each levels of the vertical axis of each ocean variables are considered a **feature**. One ocean vertical profile with ocean variables is considered a **sample**.

All the details of the Profile Classification Modelling (PCM) statistical methodology can be found in Maze et al, 2017.

1.1.3 Illustration

Given a collection of Argo temperature profiles in the North Atlantic, a PCM analysis is applied and produces an optimal set of 8 ocean temperature profile classes. The PCM clusters synthesize the structural information of heat distribution in the North Atlantic. Each clusters objectively define an ocean region where dynamic gives rise to an unique vertical stratification pattern.



Maze et al, 2017 applied it to the North Atlantic with Argo temperature data. Jones et al, 2019, later applied it to the Southern Ocean, also with Argo temperature data. Rosso et al (in prep) has applied it to the Southern Indian Ocean using both temperature and salinity Argo data.

1.1.4 pyXpcm

pyXpcm is an Python implementation of the PCM method that consumes and produces Xarray objects (`xarray.Dataset` and `xarray.DataArray`), hence the *x*.

With **pyXpcm** you can conduct a PCM analysis for a collection of profiles (gridded or not), of one or more ocean variables, stored in an `xarray.Dataset`. **pyXpcm** also provides basic statistics and plotting functions to get you started with your analysis.

The philosophy of the **pyXpcm** toolbox is to create and be able to use a PCM from and on different ocean datasets and variables. In order to achieve this, a PCM is created with information about ocean variables to classify and the vertical axis of these variables. Then this PCM can be fitted and subsequently classify ocean profiles from any datasets, as long as it contains the PCM variables.

The **pyXpcm** procedure is to preprocess (stack, scale, reduce and combine data) and then to fit a classifier on data. Once the model is fitted **pyXpcm** can classify data. The library uses many language and logic from Scikit-learn but doesn't inherit from a `sklearn.BaseEstimator`.

1.2 Installation

1.2.1 Required dependencies

- Python 3.6
- Xarray 0.12
- Dask 0.16
- Scikit-learn 0.19

Note that Scikit-learn is the default statistic backend, but that if Dask_ml is installed you can use it as well (see *API reference*).

For full plotting functionality (see the *Plotting* API) the following packages are required:

- Matplotlib 3.0 (mandatory)
- Cartopy 0.17 (for some methods only)
- Seaborn 0.9.0 (for some methods only)

1.2.2 Instructions

For the latest public release:

```
pip install pyxpcm
```

For the latest development version:

```
pip install git+http://github.com/obidam/pyxpcm.git
```

1.3 Pre-trained PCM

We'll try to list here pre-trained PCM models ready for you to use and classify your data with.

Features	K[1]	Relevant domain	Training data	Access link	Reference
Temperature (0-1400m,5m)	8	North Atlantic	Argo (2000-2014)	Archimer	Maze et al (2017)
Temperature (15-980db,5db)	8	Southern Ocean	Argo (2001-2017)	Zenodo	Jones et al (2019)

User guide

- *Standard procedure*
- *PCM properties*
- *Save and load PCM from local files*

[1] Number of classes

1.4 Standard procedure

Here is a standard procedure based on pyXpcm. This will show you how to create a model, how to fit/train it, how to classify data and to visualise results.

1.4.1 Create a model

Let's import the Profile Classification Model (PCM) constructor:

[2]:
```
from pyxpcm.models import pcm
```

A PCM can be created independently of any dataset using the class constructor.

To be created a PCM requires a number of classes (or clusters) and a dictionary to define the list of features and their vertical axis:

[3]:
```
z = np.arange(0.,-1000,-10.)
pcm_features = {'temperature': z, 'salinity':z}
```

We can now instantiate a PCM, say with 8 classes:

[4]:
```
m = pcm(K=8, features=pcm_features)
m
```

[4]:
```
<pcm 'gmm' (K: 8, F: 2)>
Number of class: 8
Number of feature: 2
Feature names: odict_keys(['temperature', 'salinity'])
Fitted: False
Feature: 'temperature'
         Interpoler: <class 'pyxpcm.utils.Vertical_Interpolator'>
         Scaler: 'normal', <class 'sklearn.preprocessing._data.StandardScaler'>
         Reducer: True, <class 'sklearn.decomposition._pca.PCA'>
Feature: 'salinity'
         Interpoler: <class 'pyxpcm.utils.Vertical_Interpolator'>
         Scaler: 'normal', <class 'sklearn.preprocessing._data.StandardScaler'>
         Reducer: True, <class 'sklearn.decomposition._pca.PCA'>
Classifier: 'gmm', <class 'sklearn.mixture._gaussian_mixture.GaussianMixture'>
```

Here we created a PCM with 8 classes (K=8) and 2 features (F=2) that are temperature and salinity profiles defined between the surface and 1000m depth.

We furthermore note the list of transform methods that will be used to preprocess each of the features (see the *preprocessing documentation page* for more details).

Note that the number of classes and features are PCM properties accessible at *pyxpcm.pcm.K* and *pyxpcm.pcm.F*.

1.4.2 Load training data

pyXpcm is able to work with both *gridded* datasets (eg: model outputs with longitude,latitude,time dimensions) and *collection* of profiles (eg: Argo, XBT, CTD section profiles).

In this example, let's import a sample of North Atlantic Argo data that come with *pyxpcm.pcm*:

[5]:
```
import pyxpcm
ds = pyxpcm.tutorial.open_dataset('argo').load()
print(ds)
```

```
<xarray.Dataset>
Dimensions:    (DEPTH: 282, N_PROF: 7560)
Coordinates:
  * DEPTH      (DEPTH) float32 0.0 -5.0 -10.0 -15.0 ... -1395.0 -1400.0 -1405.0
Dimensions without coordinates: N_PROF
Data variables:
    LATITUDE   (N_PROF) float32 ...
    LONGITUDE  (N_PROF) float32 ...
    TIME       (N_PROF) datetime64[ns] ...
    DBINDEX    (N_PROF) float64 ...
    TEMP       (N_PROF, DEPTH) float32 ...
    PSAL       (N_PROF, DEPTH) float32 ...
    SIG0       (N_PROF, DEPTH) float32 ...
    BRV2       (N_PROF, DEPTH) float32 ...
Attributes:
    Sample test prepared by:  G. Maze
    Institution:              Ifremer/LOPS
    Data source DOI:          10.17882/42182
```

1.4.3 Fit the model on data

Fitting can be done on any dataset coherent with the PCM definition, in a sense that it must have the feature variables of the PCM.

To tell the PCM model how to identify features in any `xarray.Dataset`, we need to provide a dictionary of variable names mapping:

[6]:
```
features_in_ds = {'temperature': 'TEMP', 'salinity': 'PSAL'}
```

which means that the PCM feature `temperature` is to be found in the dataset variables `TEMP`.

We also need to specify what is the vertical dimension of the dataset variables:

[7]:
```
features_zdim='DEPTH'
```

Now we're ready to fit the model on the this dataset:

[8]:
```
m.fit(ds, features=features_in_ds, dim=features_zdim)
m
```

[8]:
```
<pcm 'gmm' (K: 8, F: 2)>
Number of class: 8
Number of feature: 2
Feature names: odict_keys(['temperature', 'salinity'])
```

(continues on next page)

(continued from previous page)

```
Fitted: True
Feature: 'temperature'
         Interpoler: <class 'pyxpcm.utils.Vertical_Interpolator'>
         Scaler: 'normal', <class 'sklearn.preprocessing._data.StandardScaler'>
         Reducer: True, <class 'sklearn.decomposition._pca.PCA'>
Feature: 'salinity'
         Interpoler: <class 'pyxpcm.utils.Vertical_Interpolator'>
         Scaler: 'normal', <class 'sklearn.preprocessing._data.StandardScaler'>
         Reducer: True, <class 'sklearn.decomposition._pca.PCA'>
Classifier: 'gmm', <class 'sklearn.mixture._gaussian_mixture.GaussianMixture'>
         log likelihood of the training set: 38.784750
```

Note: pyXpcm can also identify PCM features and axis within a `xarray.DataSet` with variable attributes. From the example above we can set:

```
ds['TEMP'].attrs['feature_name'] = 'temperature'
ds['PSAL'].attrs['feature_name'] = 'salinity'
ds['DEPTH'].attrs['axis'] = 'Z'
```

And then simply call the fit method without arguments:

```
m.fit(ds)
```

Note that if data follows the CF the vertical dimension `axis` attribute should already be set to `Z`.

1.4.4 Classify data

Now that the PCM is fitted, we can predict the classification results like:

[9]:
```
m.predict(ds, features=features_in_ds, inplace=True)
ds
```

[9]:
```
<xarray.Dataset>
Dimensions:     (DEPTH: 282, N_PROF: 7560)
Coordinates:
  * N_PROF      (N_PROF) int64 0 1 2 3 4 5 6 ... 7554 7555 7556 7557 7558 7559
  * DEPTH       (DEPTH) float32 0.0 -5.0 -10.0 -15.0 ... -1395.0 -1400.0 -1405.0
Data variables:
    LATITUDE    (N_PROF) float32 ...
    LONGITUDE   (N_PROF) float32 ...
    TIME        (N_PROF) datetime64[ns] ...
    DBINDEX     (N_PROF) float64 ...
    TEMP        (N_PROF, DEPTH) float32 27.422163 27.422163 ... 4.391791
    PSAL        (N_PROF, DEPTH) float32 36.35267 36.35267 ... 34.910286
    SIG0        (N_PROF, DEPTH) float32 ...
    BRV2        (N_PROF, DEPTH) float32 ...
    PCM_LABELS  (N_PROF) int64 1 1 1 1 1 1 1 1 1 1 1 ... 3 3 3 3 3 3 3 3 3 3 3
Attributes:
    Sample test prepared by:  G. Maze
    Institution:              Ifremer/LOPS
    Data source DOI:          10.17882/42182
```

Prediction labels are automatically added to the dataset as `PCM_LABELS` because the option `inplace` was set to `True`. We didn't specify the `dim` option because our dataset is CF compliant.

pyXpcm use a GMM classifier by default, which is a fuzzy classifier. So we can also predict the probability of each classes for all profiles, the so-called *posteriors*:

[10]:
```
m.predict_proba(ds, features=features_in_ds, inplace=True)
ds
```

[10]:
```
<xarray.Dataset>
Dimensions:     (DEPTH: 282, N_PROF: 7560, pcm_class: 8)
Coordinates:
  * N_PROF      (N_PROF) int64 0 1 2 3 4 5 6 ... 7554 7555 7556 7557 7558 7559
  * DEPTH       (DEPTH) float32 0.0 -5.0 -10.0 -15.0 ... -1395.0 -1400.0 -1405.0
Dimensions without coordinates: pcm_class
Data variables:
    LATITUDE    (N_PROF) float32 27.122 27.818 27.452 26.976 ... 4.243 4.15 4.44
    LONGITUDE   (N_PROF) float32 -74.86 -75.6 -74.949 ... -1.263 -0.821 -0.002
    TIME        (N_PROF) datetime64[ns] 2008-06-23T13:07:30 ... 2013-03-09T14:52:58.
↪124999936
    DBINDEX     (N_PROF) float64 1.484e+04 1.622e+04 ... 8.557e+03 1.063e+04
    TEMP        (N_PROF, DEPTH) float32 27.422163 27.422163 ... 4.391791
    PSAL        (N_PROF, DEPTH) float32 36.35267 36.35267 ... 34.910286
    SIG0        (N_PROF, DEPTH) float32 ...
    BRV2        (N_PROF, DEPTH) float32 ...
    PCM_LABELS  (N_PROF) int64 1 1 1 1 1 1 1 1 1 1 1 ... 3 3 3 3 3 3 3 3 3 3 3
    PCM_POST    (pcm_class, N_PROF) float64 0.0 0.0 0.0 ... 1.388e-19 6.625e-20
Attributes:
    Sample test prepared by:  G. Maze
    Institution:              Ifremer/LOPS
    Data source DOI:          10.17882/42182
```

which are added to the dataset as the `PCM_POST` variables. The probability of classes for each profiles has a new dimension `pcm_class` by default that goes from 0 to K-1.

Note: You can delete variables added by pyXpcm to the `xarray.DataSet` with the *pyxpcm.xarray.pyXpcmDataSetAccessor.drop_all()* method:

```
ds.pyxpcm.drop_all()
```

Or you can split pyXpcm variables out of the original `xarray.DataSet`:

```
ds_pcm, ds = ds.pyxpcm.split()
```

It is important to note that once the PCM is fitted, you can predict labels for any dataset, as long as it has the PCM features.

For instance, let's predict labels for a gridded dataset:

[12]:
```
ds_gridded = pyxpcm.tutorial.open_dataset('isas_snapshot').load()
ds_gridded
```

[12]:
```
<xarray.Dataset>
Dimensions:      (depth: 152, latitude: 53, longitude: 61)
```

(continues on next page)

(continued from previous page)

```
Coordinates:
  * latitude     (latitude) float32 30.023445 30.455408 ... 49.41288 49.737103
  * longitude    (longitude) float32 -70.0 -69.5 -69.0 ... -41.0 -40.5 -40.0
  * depth        (depth) float32 -1.0 -3.0 -5.0 ... -1960.0 -1980.0 -2000.0
Data variables:
    TEMP         (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    TEMP_ERR     (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    TEMP_PCTVAR  (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    PSAL         (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    PSAL_ERR     (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    PSAL_PCTVAR  (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    SST          (latitude, longitude) float32 dask.array<chunksize=(53, 61), meta=np.
↪ndarray>
```

[13]:
```
m.predict(ds_gridded, features={'temperature':'TEMP','salinity':'PSAL'}, dim='depth',␣
↪inplace=True)
ds_gridded
```

[13]:
```
<xarray.Dataset>
Dimensions:      (depth: 152, latitude: 53, longitude: 61)
Coordinates:
  * latitude     (latitude) float64 30.02 30.46 30.89 ... 49.09 49.41 49.74
  * longitude    (longitude) float64 -70.0 -69.5 -69.0 ... -41.0 -40.5 -40.0
  * depth        (depth) float32 -1.0 -3.0 -5.0 ... -1960.0 -1980.0 -2000.0
Data variables:
    TEMP         (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    TEMP_ERR     (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    TEMP_PCTVAR  (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    PSAL         (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    PSAL_ERR     (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    PSAL_PCTVAR  (depth, latitude, longitude) float32 dask.array<chunksize=(152, 53, 61),
↪ meta=np.ndarray>
    SST          (latitude, longitude) float32 dask.array<chunksize=(53, 61), meta=np.
↪ndarray>
    PCM_LABELS   (latitude, longitude) float64 1.0 1.0 1.0 1.0 ... 7.0 7.0 7.0
```

where you can see the adition of the `PCM_LABELS` variable.

1.4.5 Vertical structure of classes

One key outcome of the PCM analysis if the vertical structure of each classes. This can be computed using the :meth:`pyxpcm.stat.quantile` method.

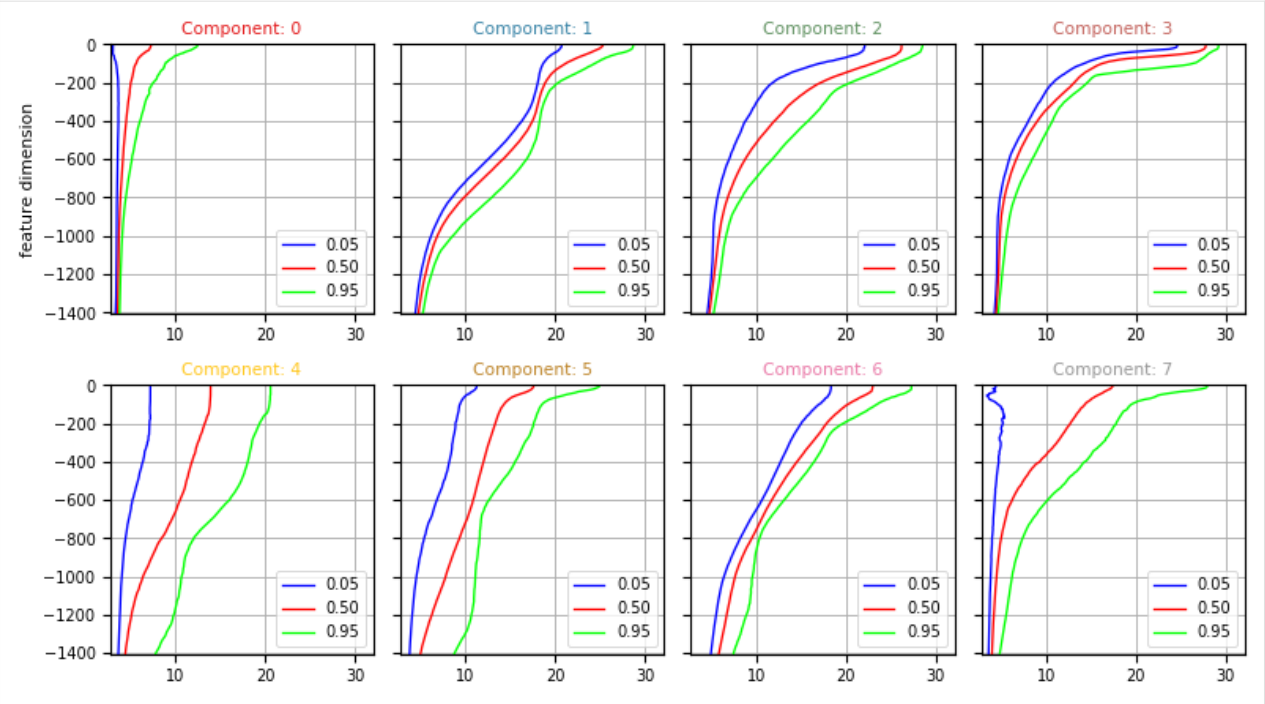
Below we compute the 5, 50 and 95% quantiles for temperature and salinity of each classes:

[14]:
```
for vname in ['TEMP', 'PSAL']:
    ds = ds.pyxpcm.quantile(m, q=[0.05, 0.5, 0.95], of=vname, outname=vname + '_Q', keep_
↪attrs=True, inplace=True)
ds
```

[14]:
```
<xarray.Dataset>
Dimensions:     (DEPTH: 282, N_PROF: 7560, pcm_class: 8, quantile: 3)
Coordinates:
  * pcm_class   (pcm_class) int64 0 1 2 3 4 5 6 7
  * N_PROF      (N_PROF) int64 0 1 2 3 4 5 6 ... 7554 7555 7556 7557 7558 7559
  * DEPTH       (DEPTH) float32 0.0 -5.0 -10.0 -15.0 ... -1395.0 -1400.0 -1405.0
  * quantile    (quantile) float64 0.05 0.5 0.95
Data variables:
    LATITUDE    (N_PROF) float32 27.122 27.818 27.452 26.976 ... 4.243 4.15 4.44
    LONGITUDE   (N_PROF) float32 -74.86 -75.6 -74.949 ... -1.263 -0.821 -0.002
    TIME        (N_PROF) datetime64[ns] 2008-06-23T13:07:30 ... 2013-03-09T14:52:58.
↪124999936
    DBINDEX     (N_PROF) float64 1.484e+04 1.622e+04 ... 8.557e+03 1.063e+04
    TEMP        (N_PROF, DEPTH) float32 27.422163 27.422163 ... 4.391791
    PSAL        (N_PROF, DEPTH) float32 36.35267 36.35267 ... 34.910286
    SIG0        (N_PROF, DEPTH) float32 23.601229 23.601229 ... 27.685583
    BRV2        (N_PROF, DEPTH) float32 0.00029447526 ... 4.500769e-06
    PCM_LABELS  (N_PROF) int64 1 1 1 1 1 1 1 1 1 1 1 ... 3 3 3 3 3 3 3 3 3 3 3
    PCM_POST    (pcm_class, N_PROF) float64 0.0 0.0 0.0 ... 1.388e-19 6.625e-20
    TEMP_Q      (pcm_class, quantile, DEPTH) float64 3.07 3.07 ... 4.83 4.814
    PSAL_Q      (pcm_class, quantile, DEPTH) float64 34.03 34.03 ... 35.01 35.01
Attributes:
    Sample test prepared by:  G. Maze
    Institution:              Ifremer/LOPS
    Data source DOI:          10.17882/42182
```

Quantiles can be plotted using the :func:`pyxpcm.plot.quantile` method.

[15]:
```
fig, ax = m.plot.quantile(ds['TEMP_Q'], maxcols=4, figsize=(10, 8), sharey=True)
```

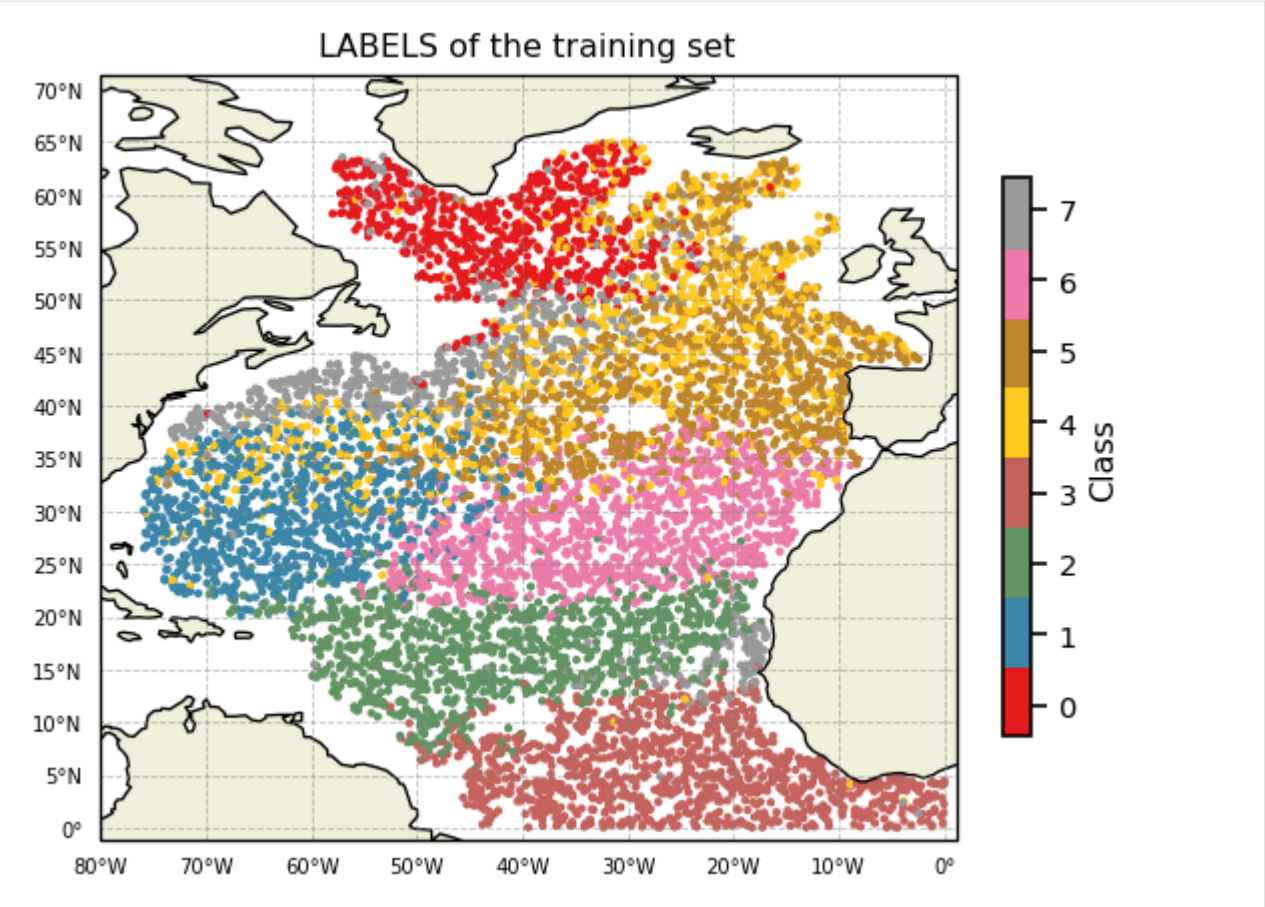


1.4.6 Geographic distribution of classes

Warning: To follow this section you'll need to have Cartopy installed and working.

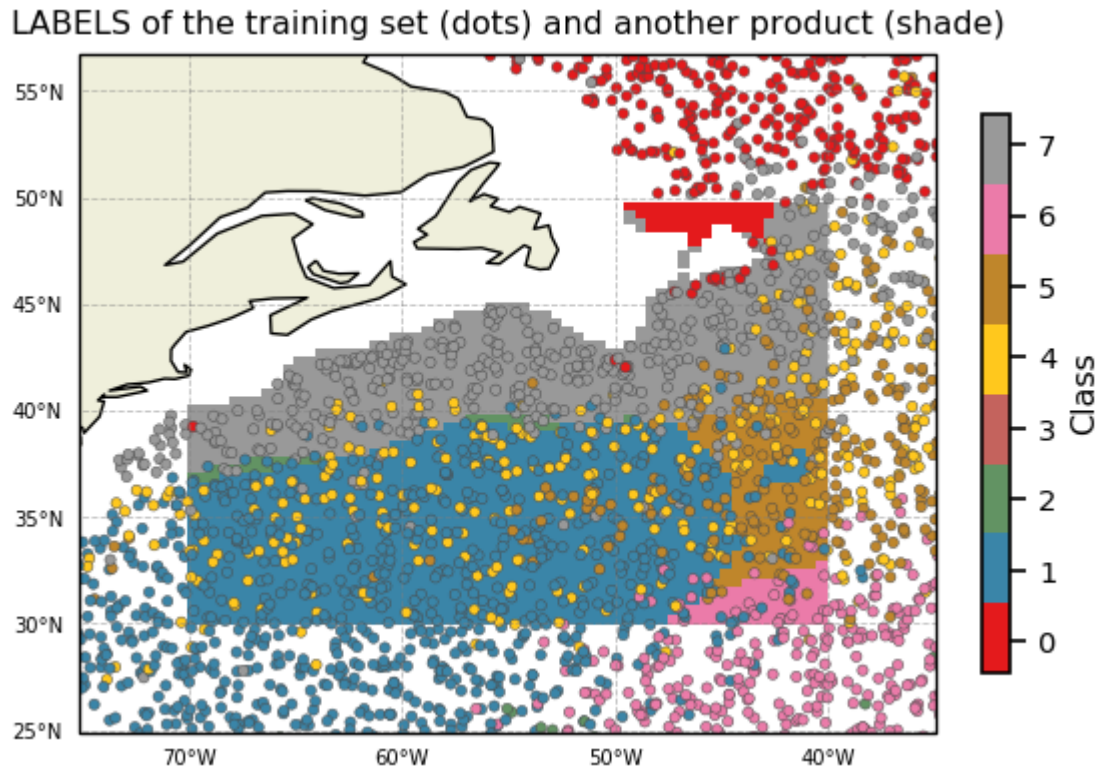
A map of labels can now easily be plotted:

[16]:
```
proj = ccrs.PlateCarree()
subplot_kw={'projection': proj, 'extent': np.array([-80,1,-1,66]) + np.array([-0.1,+0.1,-
↪0.1,+0.1])}
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(5,5), dpi=120, facecolor='w',␣
↪edgecolor='k', subplot_kw=subplot_kw)

kmap = m.plot.cmap()
sc = ax.scatter(ds['LONGITUDE'], ds['LATITUDE'], s=3, c=ds['PCM_LABELS'], cmap=kmap,␣
↪transform=proj, vmin=0, vmax=m.K)
cl = m.plot.colorbar(ax=ax)

gl = m.plot.latlongrid(ax, dx=10)
ax.add_feature(cfeature.LAND)
ax.add_feature(cfeature.COASTLINE)
ax.set_title('LABELS of the training set')
plt.show()
```



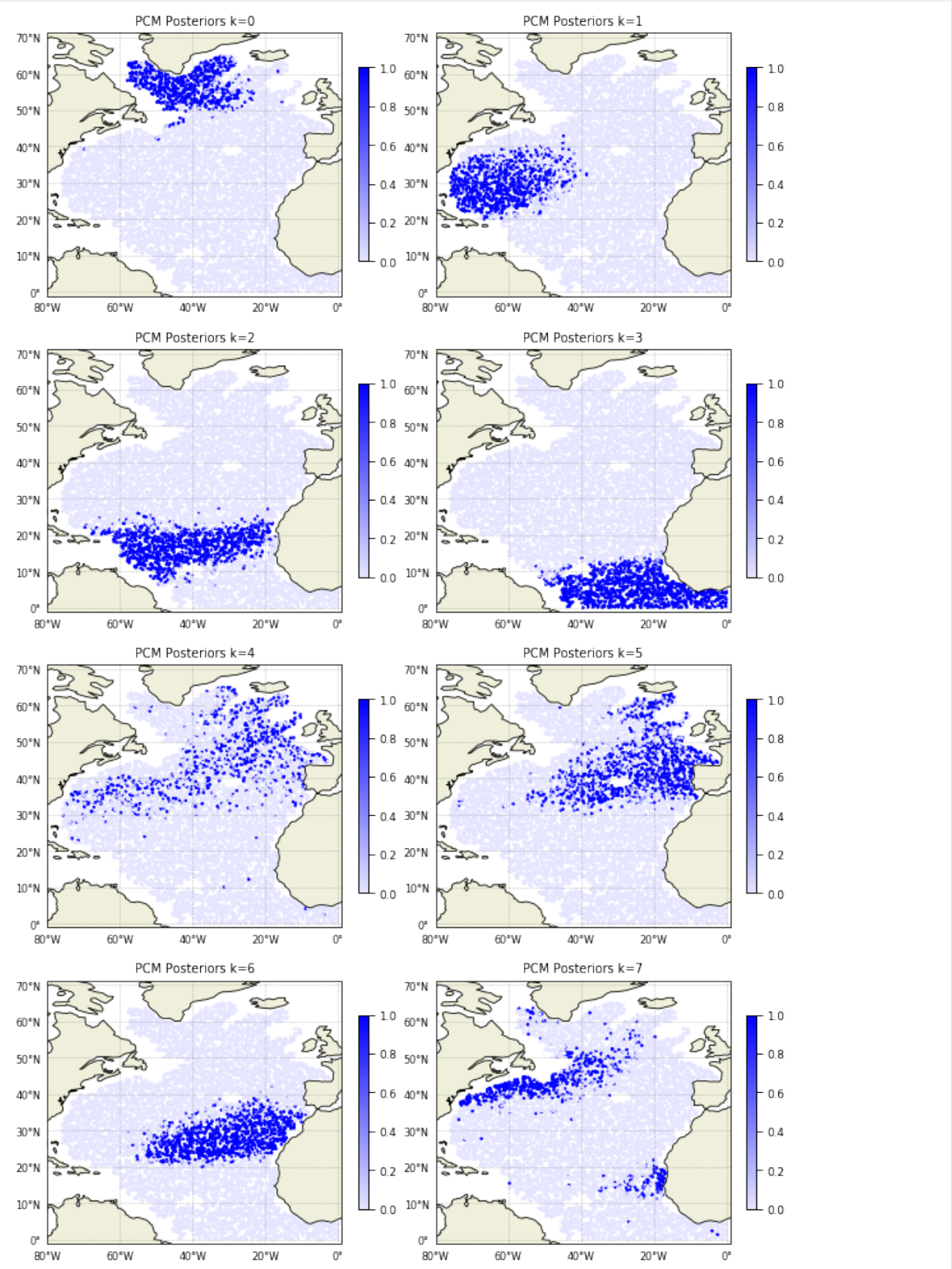
Since we predicted labels for 2 datasets, we can superimpose them

[17]:
```
proj = ccrs.PlateCarree()
subplot_kw={'projection': proj, 'extent': np.array([-75,-35,25,55]) + np.array([-0.1,+0.
↪1,-0.1,+0.1])}
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(5,5), dpi=120, facecolor='w',␣
↪edgecolor='k', subplot_kw=subplot_kw)

kmap = m.plot.cmap()
sc = ax.pcolor(ds_gridded['longitude'], ds_gridded['latitude'], ds_gridded['PCM_LABELS'],
↪ cmap=kmap, transform=proj, vmin=0, vmax=m.K)
sc = ax.scatter(ds['LONGITUDE'], ds['LATITUDE'], s=10, c=ds['PCM_LABELS'], cmap=kmap,␣
↪transform=proj, vmin=0, vmax=m.K, edgecolors=[0.3]*3, linewidths=0.3)
cl = m.plot.colorbar(ax=ax)

gl = m.plot.latlongrid(ax, dx=10)
ax.add_feature(cfeature.LAND)
ax.add_feature(cfeature.COASTLINE)
ax.set_title('LABELS of the training set (dots) and another product (shade)')
plt.show()
```



Posteriors are defined for each data point and give the probability of that point to belong to any of the classes. It can be plotted this way:

[18]:
```
cmap = sns.light_palette("blue", as_cmap=True)
proj = ccrs.PlateCarree()
subplot_kw={'projection': proj, 'extent': np.array([-80,1,-1,66]) + np.array([-0.1,+0.1,-
↪0.1,+0.1])}
fig, ax = m.plot.subplots(figsize=(10,22), maxcols=2, subplot_kw=subplot_kw)

for k in m:
    sc = ax[k].scatter(ds['LONGITUDE'], ds['LATITUDE'], s=3, c=ds['PCM_POST'].sel(pcm_
↪class=k),
                       cmap=cmap, transform=proj, vmin=0, vmax=1)
    cl = plt.colorbar(sc, ax=ax[k], fraction=0.03)
    gl = m.plot.latlongrid(ax[k], fontsize=8, dx=20, dy=10)
    ax[k].add_feature(cfeature.LAND)
    ax[k].add_feature(cfeature.COASTLINE)
    ax[k].set_title('PCM Posteriors k=%i' % k)
```


PCM Posteriors k=0
PCM Posteriors k=1
PCM Posteriors k=2
PCM Posteriors k=3
PCM Posteriors k=4
PCM Posteriors k=5
PCM Posteriors k=6
PCM Posteriors k=7
70°N
60°N
50°N
40°N
30°N
20°N
10°N
0°
80°W
60°W
40°W
20°W
0°
1.0
0.8
0.6
0.4
0.2
0.0

1.5 PCM properties

The PCM class does a lot of data preprocessing under the hood in order to classify profiles.

Here is how to access PCM preprocessed results and data.

Import and set-up

Import the library and toy data

[2]:
```
import pyxpcm
from pyxpcm.models import pcm
```

Let's work with a standard PCM of temperature and salinity, from the surface down to -1000m:

[3]:
```
# Define a vertical axis to work with
z = np.arange(0.,-1000,-10.)

# Define features to use
features_pcm = {'temperature': z, 'salinity': z}

# Instantiate the PCM
m = pcm(K=4, features=features_pcm, maxvar=2)
print(m)
```

```
<pcm 'gmm' (K: 4, F: 2)>
Number of class: 4
Number of feature: 2
Feature names: odict_keys(['temperature', 'salinity'])
Fitted: False
Feature: 'temperature'
         Interpoler: <class 'pyxpcm.utils.Vertical_Interpolator'>
         Scaler: 'normal', <class 'sklearn.preprocessing._data.StandardScaler'>
         Reducer: True, <class 'sklearn.decomposition._pca.PCA'>
Feature: 'salinity'
         Interpoler: <class 'pyxpcm.utils.Vertical_Interpolator'>
         Scaler: 'normal', <class 'sklearn.preprocessing._data.StandardScaler'>
         Reducer: True, <class 'sklearn.decomposition._pca.PCA'>
Classifier: 'gmm', <class 'sklearn.mixture._gaussian_mixture.GaussianMixture'>
```

Note that here we used a strong dimensionality reduction to limit the dimensions and size of the plots to come (`maxvar==2` tell the PCM to use the first 2 PCAs of each variables).

Now we can load a dataset to be used for fitting.

[4]:
```
ds = pyxpcm.tutorial.open_dataset('argo').load()
```

Fit and predict model on data:

[5]:
```
features_in_ds = {'temperature': 'TEMP', 'salinity': 'PSAL'}
ds = ds.pyxpcm.fit_predict(m, features=features_in_ds, inplace=True)
print(ds)
```

```
<xarray.Dataset>
Dimensions:     (DEPTH: 282, N_PROF: 7560)
Coordinates:
```

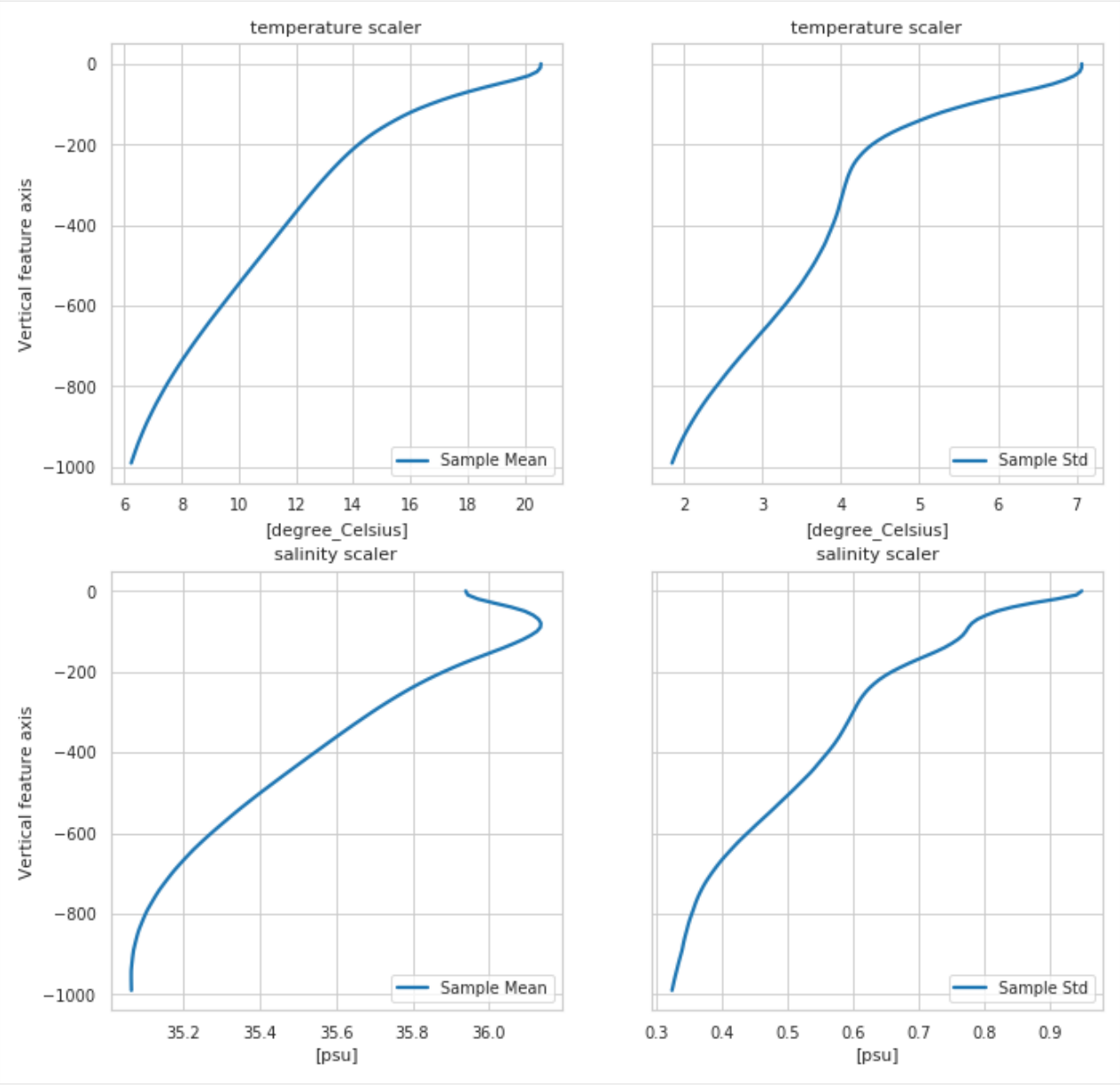
(continues on next page)

(continued from previous page)

```
  * N_PROF       (N_PROF) int64 0 1 2 3 4 5 6 ... 7554 7555 7556 7557 7558 7559
  * DEPTH        (DEPTH) float32 0.0 -5.0 -10.0 -15.0 ... -1395.0 -1400.0 -1405.0
Data variables:
    LATITUDE     (N_PROF) float32 ...
    LONGITUDE    (N_PROF) float32 ...
    TIME         (N_PROF) datetime64[ns] ...
    DBINDEX      (N_PROF) float64 ...
    TEMP         (N_PROF, DEPTH) float32 27.422163 27.422163 ... 4.391791
    PSAL         (N_PROF, DEPTH) float32 36.35267 36.35267 ... 34.910286
    SIG0         (N_PROF, DEPTH) float32 ...
    BRV2         (N_PROF, DEPTH) float32 ...
    PCM_LABELS   (N_PROF) int64 1 1 1 1 1 1 1 1 1 1 1 ... 2 2 2 2 2 2 2 2 2 2 2
Attributes:
    Sample test prepared by:  G. Maze
    Institution:              Ifremer/LOPS
    Data source DOI:          10.17882/42182
```

1.5.1 Scaler properties

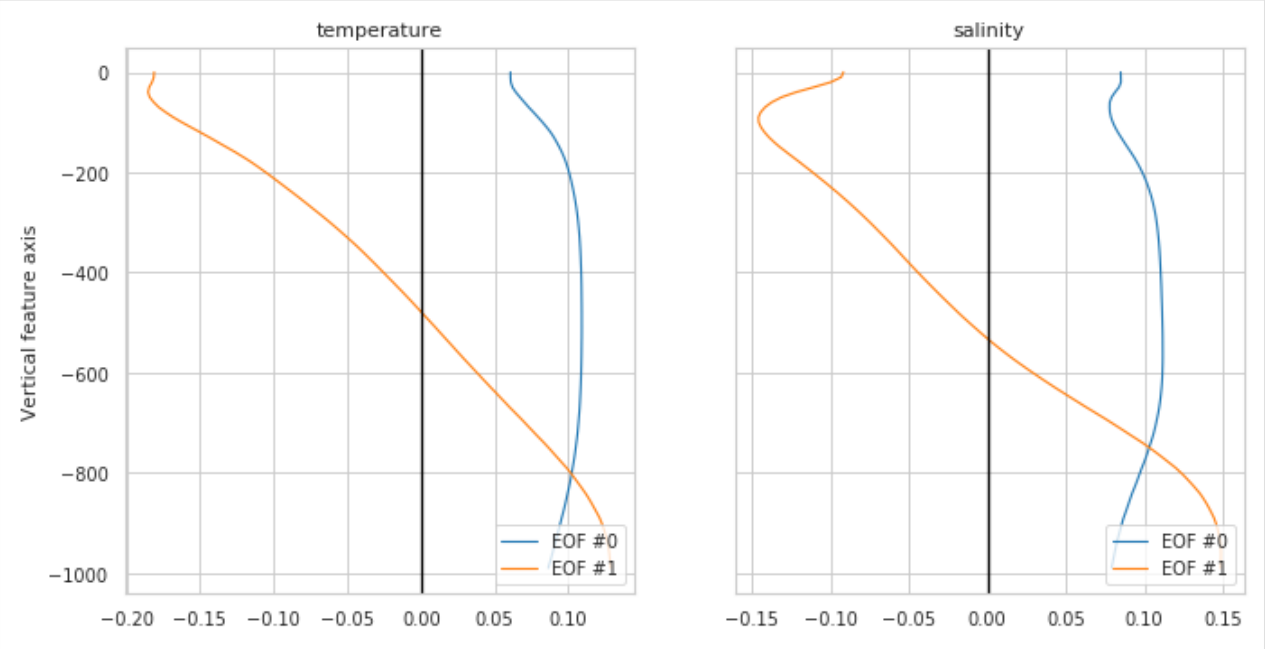
[6]:
```
fig, ax = m.plot.scaler()

# More options:
# m.plot.scaler(style='darkgrid')
# m.plot.scaler(style='darkgrid', subplot_kw={'ylim':[-1000,0]})
```



1.5.2 Reducer properties

Plot eigen vectors for a PCA reducer or nothing if no reduced used

[7]:
```
fig, ax = m.plot.reducer()
# Equivalent to:
# pcmplot.reducer(m)

# More options:
# m.plot.reducer(pcalist = range(0,4));
# m.plot.reducer(pcalist = [0], maxcols=1);
# m.plot.reducer(pcalist = range(0,4), style='darkgrid',  plot_kw={'linewidth':1.5},␣
↪subplot_kw={'ylim':[-1400,0]}, figsize=(12,10));
```



1.5.3 Scatter plot of features, *as seen by the classifier*

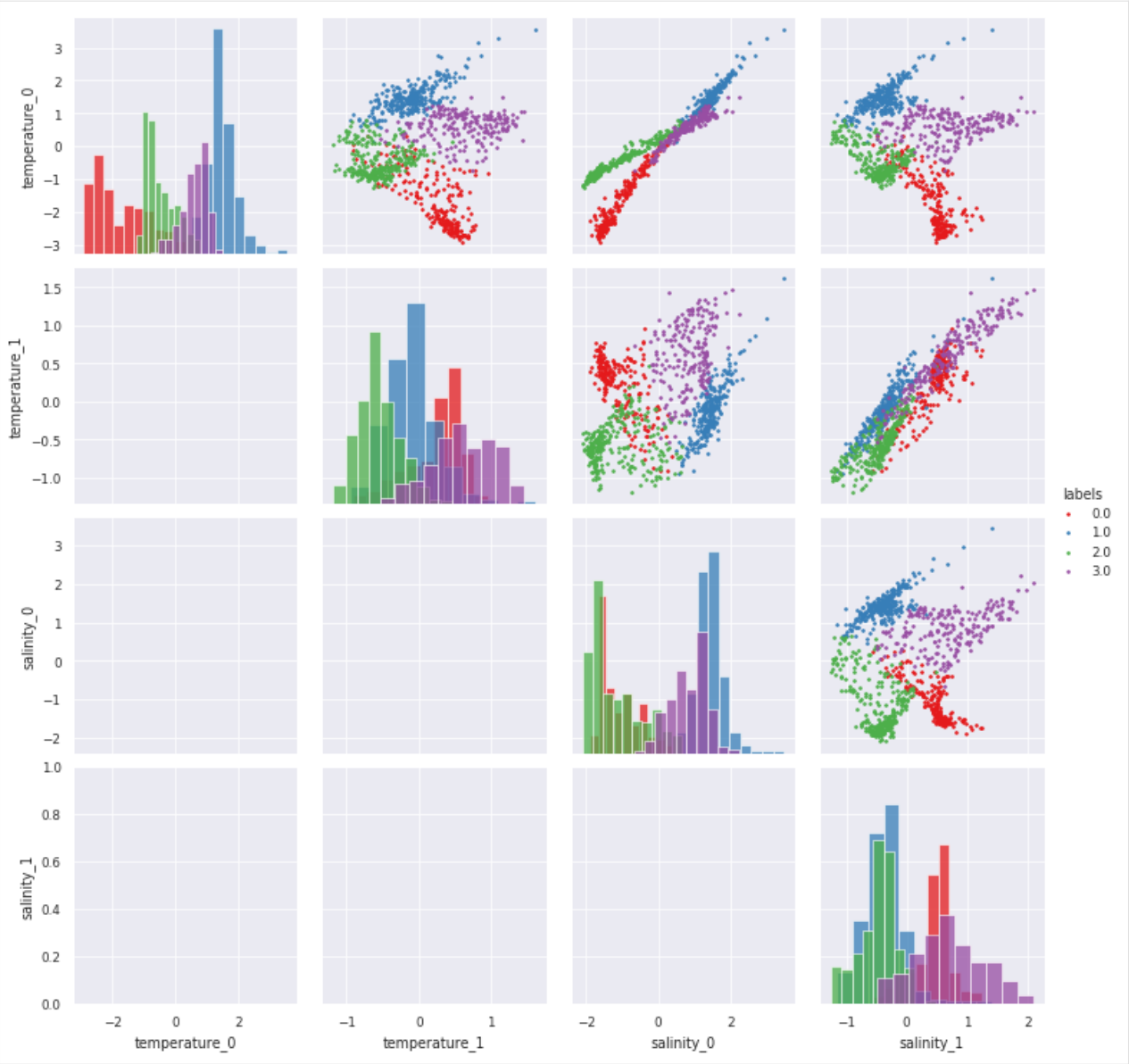
You can have access to pre-processed data for your own plot/analysis through the `pyxpcm.pcm.preprocessing()` method:

[8]:
```
X, sampling_dims = m.preprocessing(ds, features=features_in_ds)
X
```

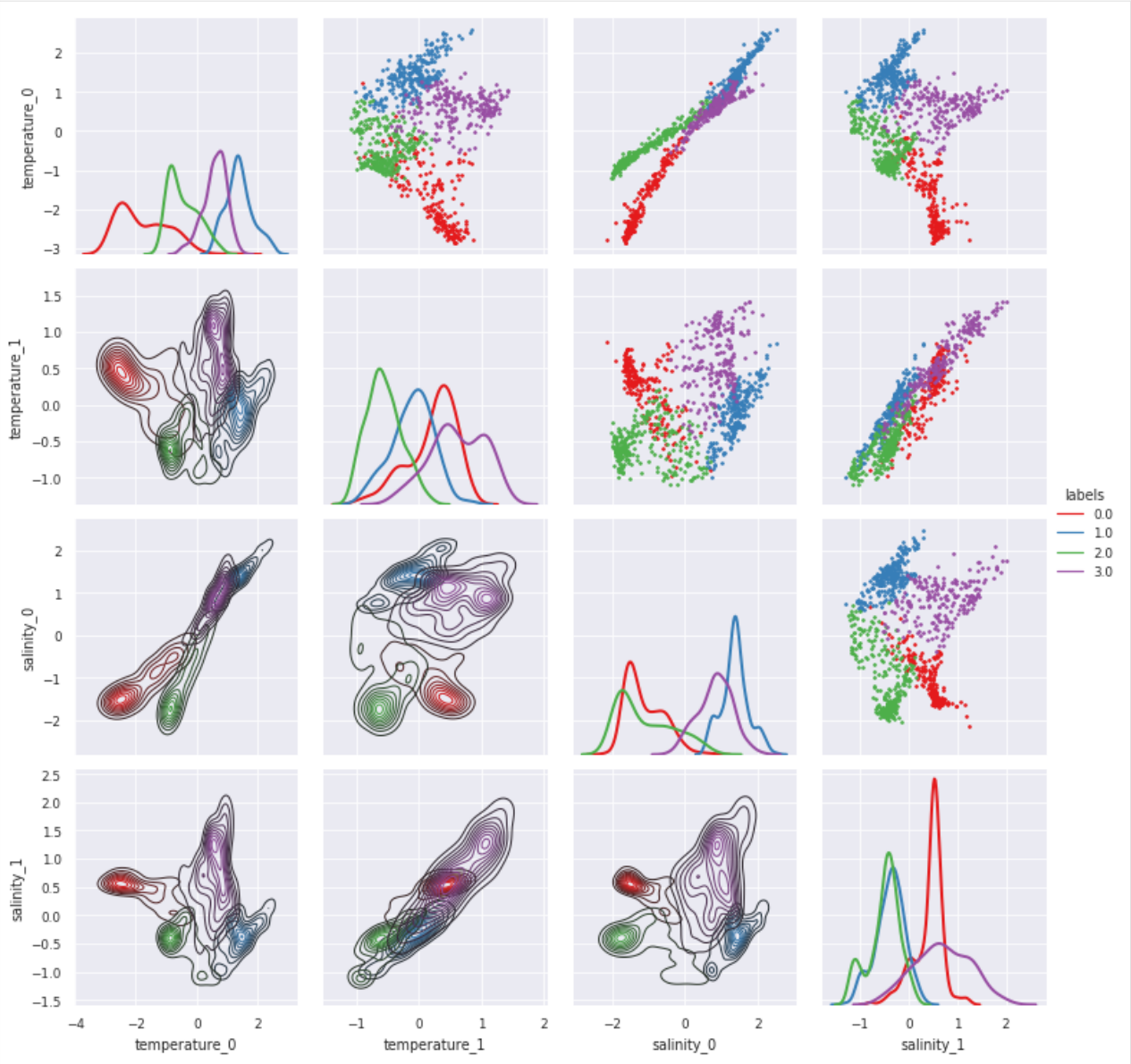
[8]:
```
<xarray.DataArray (n_samples: 7560, n_features: 4)>
array([[ 1.9281656 , -0.09149919,  1.7340997 , -0.27024782],
       [ 2.314077  ,  0.10684185,  2.0836833 , -0.18765019],
       [ 1.6755121 , -0.17313023,  1.5637012 , -0.43244886],
       ...,
       [-0.802601  , -0.5783772 , -1.5761338 , -0.31184074],
       [-0.9552184 , -0.6094395 , -1.8049222 , -0.4273216 ],
       [-0.8925139 , -0.6237318 , -1.7922652 , -0.46551177]],
      dtype=float32)
Coordinates:
  * n_samples   (n_samples) int64 0 1 2 3 4 5 ... 7554 7555 7556 7557 7558 7559
  * n_features  (n_features) <U13 'temperature_0' ... 'salinity_1'
```

pyXpcm return a 2-dimensional `xarray.DataArray` for which pairwise relationship can easily be visualise with the *`pyxpcm.plot.preprocessed()`* method (this requires Seaborn):

[9]:
```
g = m.plot.preprocessed(ds, features=features_in_ds, style='darkgrid')

# A posteriori adjustements:
# g.set(xlim=(-3,3),ylim=(-3,3))
# g.savefig('toto.png')
```



[10]:
```
# Combine KDE with histrograms (very slow plot, so commented here):
g = m.plot.preprocessed(ds, features=features_in_ds, kde=True)
```



1.6 Save and load PCM from local files

PCM instances are light weigth python objects and can easily be saved on and loaded from files. pyXpcm uses the netcdf file format because it is easy to add meta-data to numerical arrays.

Import and set-up

Import the library and toy data

[2]:
```
import pyxpcm
from pyxpcm.models import pcm

# Load tutorial data:
ds = pyxpcm.tutorial.open_dataset('argo').load()
```

1.6.1 Saving a model

Let's first create a PCM and fit it onto the tutorial dataset:

[3]:
```
# Define a vertical axis to work with
z = np.arange(0.,-1000,-10.)

# Define features to use
features_pcm = {'temperature': z, 'salinity': z}

# Instantiate the PCM:
m = pcm(K=4, features=features_pcm)

# Fit:
m.fit(ds, features={'temperature': 'TEMP', 'salinity': 'PSAL'})
```

[3]:
```
<pcm 'gmm' (K: 4, F: 2)>
Number of class: 4
Number of feature: 2
Feature names: odict_keys(['temperature', 'salinity'])
Fitted: True
Feature: 'temperature'
         Interpoler: <class 'pyxpcm.utils.Vertical_Interpolator'>
         Scaler: 'normal', <class 'sklearn.preprocessing._data.StandardScaler'>
         Reducer: True, <class 'sklearn.decomposition._pca.PCA'>
Feature: 'salinity'
         Interpoler: <class 'pyxpcm.utils.Vertical_Interpolator'>
         Scaler: 'normal', <class 'sklearn.preprocessing._data.StandardScaler'>
         Reducer: True, <class 'sklearn.decomposition._pca.PCA'>
Classifier: 'gmm', <class 'sklearn.mixture._gaussian_mixture.GaussianMixture'>
         log likelihood of the training set: 33.335059
```

We can now save the fitted model to a local file:

[4]:
```
m.to_netcdf('my_pcm.nc')
```

1.6.2 Loading a model

To load a PCM from file, use:

[5]:
```
m_loaded = pyxpcm.load_netcdf('my_pcm.nc')
m_loaded
```

[5]:
```
<pcm 'gmm' (K: 4, F: 2)>
Number of class: 4
Number of feature: 2
Feature names: odict_keys(['temperature', 'salinity'])
Fitted: True
Feature: 'temperature'
         Interpoler: <class 'pyxpcm.utils.Vertical_Interpolator'>
         Scaler: 'normal', <class 'sklearn.preprocessing._data.StandardScaler'>
         Reducer: True, <class 'sklearn.decomposition._pca.PCA'>
Feature: 'salinity'
```

(continues on next page)

(continued from previous page)

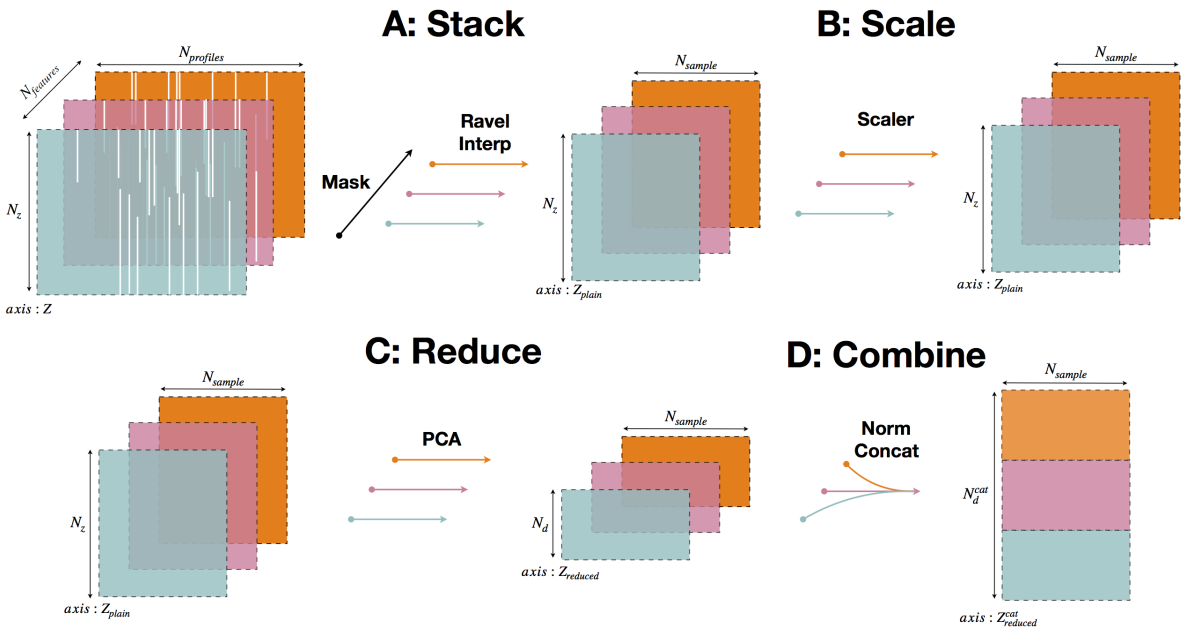
```
        Interpoler: <class 'pyxpcm.utils.Vertical_Interpolator'>
        Scaler: 'normal', <class 'sklearn.preprocessing._data.StandardScaler'>
        Reducer: True, <class 'sklearn.decomposition._pca.PCA'>
Classifier: 'gmm', <class 'sklearn.mixture._gaussian_mixture.GaussianMixture'>
        log likelihood of the training set: 33.335059
```


1.7 Features preprocessing explained

Note: **pyXpcm** performs automatically all the data preprocessing for you. You should not have to manipulate your data before being able to fit or classify. This page is an explanation of the preprocessing steps performed under the hood by **pyXpcm**.

The Profile Classification Model (PCM) requires data to be preprocessed in order to match the model vertical axis, to scale feature dimensions with each others and to reduce the dimensionality of the problem. Preprocessing is done internally by **pyXpcm**. Each step can be parameterised.

The PCM preprocessing operations are organised into 4 steps:



1.7.1 Stack

This step mask, extract, flatten and transform any ND-array set of feature variables (eg: temperature, salinity) into a plain 2D-array collection of vertical profiles usable for machine learning methods.

Mask

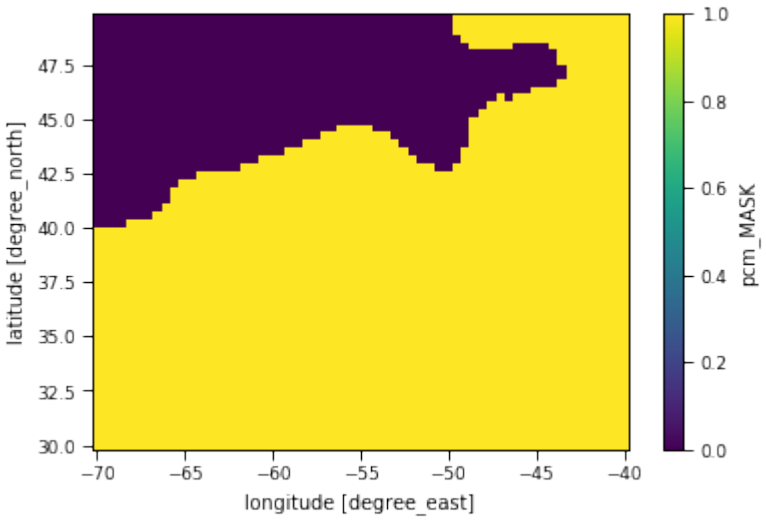
This step computes a mask of the input data that will reject all profiles that are full of nans over the depth range of feature vertical axis. This ensure that all feature variables will be successfully retrieved to fill in the plain 2D-array collection of profiles.

This operation is conducted by *pyxpcm.xarray.pyXpcmDataSetAccessor.mask()*, so that the mask can be computed (and plotted) this way:

[2]:
```
mask = ds.pyxpcm.mask(m)
print(mask)
```

```
<xarray.DataArray 'pcm_MASK' (latitude: 53, longitude: 61)>
dask.array<eq, shape=(53, 61), dtype=bool, chunksize=(53, 61), chunktype=numpy.ndarray>
Coordinates:
  * latitude   (latitude) float32 30.023445 30.455408 ... 49.41288 49.737103
  * longitude  (longitude) float32 -70.0 -69.5 -69.0 -68.5 ... -41.0 -40.5 -40.0
```

[3]:
```
mask.plot();
```



Ravel

For ND-array to be used as a feature, it has to be ravelled, flatten, along the N-1 dimensions that are not the vertical one. This operation will thus transform any ND-array into a 2D-array (`sampling` and `vertical_axis` dimensions) and additionnaly drop profiles according to the PCM mask determined above.

This operation is conducted by *pyxpcm.pcm.ravel()*.

The output 2D-array is a `xarray.DataArray` that can be chunked along the `sampling` dimension with the PCM constructor option `chunk_size`:

[4]:
```
m = pcm(K=3, features=features_pcm, chunk_size=1e3).fit(ds)
```

By default, `chunk_size='auto'`.

[5]:
```
X, z, sampling_dims = m.ravel(ds['TEMP'], dim='depth', feature_name='TEMP')
X
```

[5]:
```
<xarray.DataArray 'TEMP' (sampling: 2289, depth: 152)>
dask.array<rechunk-merge, shape=(2289, 152), dtype=float32, chunksize=(1000, 152),␣
↪chunktype=numpy.ndarray>
Coordinates:
  * depth      (depth) float32 -1.0 -3.0 -5.0 -10.0 ... -1960.0 -1980.0 -2000.0
  * sampling   (sampling) MultiIndex
  - latitude   (sampling) float64 30.02 30.02 30.02 30.02 ... 49.74 49.74 49.74
  - longitude  (sampling) float64 -70.0 -69.5 -69.0 -68.5 ... -41.0 -40.5 -40.0
Attributes:
    long_name:      Temperature
    standard_name:  sea_water_temperature
    units:          degree_Celsius
    valid_min:      -23000
    valid_max:      20000
```

See the chunksize of the `dask.array.Array` for this feature.

Interpolate

Even if input data vertical axis are in the range of the PCM feature axis, they may not be defined on similar level values. In this step, if the input data are not defined on the same vertical axis as the PCM, an interpolation is triggered. The interpolation is conducted following these rules:

- If PCM axis levels are found into the input data vertical axis, then a simple intersection is used.
- If PCM axis starts at the surface (0 value) and not the input data, the 1st non-nan value is replicated to the surface, as a mixed layer.
- If PCM axis levels are not in the input data vertical axis, a linear interpolation through the `xarray.DataArray.interp()` method is triggered for each profiles.

The entire interpolation processed is managed by a `pyxpcm.utils.Vertical_Interpolator` instance that is created at the time of PCM instanciation.

1.7.2 Scale

Each variable can be normalised along a vertical level. This step ensures that structures/patterns located at depth in the profile, will be considered similarly to those close to the surface by the classifier.

Scaling is defined at the PCM creation (`pyxpcm.models.pcm`) with the option `scale`. It is an integer value with the following meaning:

- 0: No scaling
- 1: Center on sample mean and scale by sample std
- 2: Center on sample mean only

1.7.3 Recuce

[TBC]

1.7.4 Combine

[TBC]

1.8 Debugging and performances

Import and set-up

Import the library and toy data

[2]:
```
import pyxpcm
from pyxpcm.models import pcm

# Load a dataset to work with:
ds = pyxpcm.tutorial.open_dataset('argo').load()

# Define vertical axis and features to use:
z = np.arange(0.,-1000.,-10.)
features_pcm = {'temperature': z, 'salinity': z}
features_in_ds = {'temperature': 'TEMP', 'salinity': 'PSAL'}
```

1.8.1 Debugging

Use option debug to print log messages

[3]:
```
# Instantiate a new PCM:
m = pcm(K=8, features=features_pcm, debug=True)

# Fit with log:
m.fit(ds, features=features_in_ds);
```

```
> Start preprocessing for action 'fit'
```

```
       > Preprocessing xarray dataset 'TEMP' as PCM feature 'temperature'
        X RAVELED with success [<class 'xarray.core.dataarray.DataArray'>, <class 'dask.
↪array.core.Array'>, ((7560,), (282,))]
               Output axis is in the input axis, not need to interpolate, simple␣
↪intersection
        X INTERPOLATED with success [<class 'xarray.core.dataarray.DataArray'>, <class
↪'dask.array.core.Array'>, ((7560,), (100,))]
        X SCALED with success) [<class 'xarray.core.dataarray.DataArray'>, <class
↪'numpy.ndarray'>, None]
        X REDUCED with success) [<class 'xarray.core.dataarray.DataArray'>, <class
↪'numpy.ndarray'>, None]
       temperature pre-processed with success,  [<class 'xarray.core.dataarray.DataArray
↪'>, <class 'numpy.ndarray'>, None]
```

(continues on next page)

(continued from previous page)

```
        Homogenisation for fit of temperature

        > Preprocessing xarray dataset 'PSAL' as PCM feature 'salinity'
```

```
         X RAVELED with success [<class 'xarray.core.dataarray.DataArray'>, <class 'dask.
↪array.core.Array'>, ((7560,), (282,))]
                Output axis is in the input axis, not need to interpolate, simple␣
↪intersection
         X INTERPOLATED with success [<class 'xarray.core.dataarray.DataArray'>, <class
↪'dask.array.core.Array'>, ((7560,), (100,))]
         X SCALED with success) [<class 'xarray.core.dataarray.DataArray'>, <class
↪'numpy.ndarray'>, None]
         X REDUCED with success) [<class 'xarray.core.dataarray.DataArray'>, <class
↪'numpy.ndarray'>, None]
        salinity pre-processed with success,  [<class 'xarray.core.dataarray.DataArray'>,
↪ <class 'numpy.ndarray'>, None]
        Homogenisation for fit of salinity
        Features array shape and type for xarray: (7560, 30) <class 'numpy.ndarray'>
↪<class 'memoryview'>
> Preprocessing done, working with final X (<class 'xarray.core.dataarray.DataArray'>)␣
↪array of shape: (7560, 30)  and sampling dimensions: ['N_PROF']
```

1.8.2 Performance / Optimisation

Use `timeit` and `timeit_verb` to compute computation time of PCM operations

Times are accessible as a pandas Dataframe in `timeit` pyXpcm instance property.

The pyXpcm `m.plot.timeit()` plot method allows for a simple visualisation of times.

Time readings during execution

[4]:
```
# Create a PCM and execute methods:
m = pcm(K=8, features=features_pcm, timeit=True, timeit_verb=1)
m.fit(ds, features=features_in_ds);
```

```
  fit.1-preprocess.1-mask: 26 ms
  fit.1-preprocess.2-feature_temperature.1-ravel: 48 ms
  fit.1-preprocess.2-feature_temperature.2-interp: 2 ms
  fit.1-preprocess.2-feature_temperature.3-scale_fit: 14 ms
  fit.1-preprocess.2-feature_temperature.4-scale_transform: 8 ms
  fit.1-preprocess.2-feature_temperature.5-reduce_fit: 22 ms
  fit.1-preprocess.2-feature_temperature.6-reduce_transform: 5 ms
  fit.1-preprocess.2-feature_temperature.total: 103 ms
  fit.1-preprocess: 103 ms
  fit.1-preprocess.3-homogeniser: 1 ms
  fit.1-preprocess.2-feature_salinity.1-ravel: 42 ms
  fit.1-preprocess.2-feature_salinity.2-interp: 1 ms
  fit.1-preprocess.2-feature_salinity.3-scale_fit: 12 ms
  fit.1-preprocess.2-feature_salinity.4-scale_transform: 10 ms
  fit.1-preprocess.2-feature_salinity.5-reduce_fit: 14 ms
  fit.1-preprocess.2-feature_salinity.6-reduce_transform: 3 ms
```

(continues on next page)

(continued from previous page)

```
  fit.1-preprocess.2-feature_salinity.total: 85 ms
  fit.1-preprocess: 85 ms
  fit.1-preprocess.3-homogeniser: 1 ms
  fit.1-preprocess.4-xarray: 1 ms
  fit.1-preprocess: 225 ms
```

```
  fit.fit: 2206 ms
  fit.score: 10 ms
  fit: 2442 ms
```

A posteriori Execution time analysis

[5]:
```
# Create a PCM and execute methods:
m = pcm(K=8, features=features_pcm, timeit=True, timeit_verb=0)
m.fit(ds, features=features_in_ds);
m.predict(ds, features=features_in_ds);
m.fit_predict(ds, features=features_in_ds);
```

Execution times are accessible through a dataframe with the *pyxpcm.pcm.timeit* property

[6]:
```
m.timeit
```

[6]:
```
Method       Sub-method    Sub-sub-method          Sub-sub-sub-method
fit          1-preprocess  1-mask                  total                  19.836187
                           2-feature_temperature   1-ravel                32.550097
                                                   2-interp                0.828981
                                                   3-scale_fit             9.926319
                                                   4-scale_transform       5.232811
                                                                             ...
fit_predict  fit           total                                         737.503052
             score         total                                           8.855104
             predict       total                                           9.073734
             xarray        total                                           8.368969
             total                                                       882.753134
Length: 66, dtype: float64
```

1.8.3 Visualisation help

To facilitate your analysis of execution times, you can use *pyxpcm.plot.timeit()*.

Main steps by method

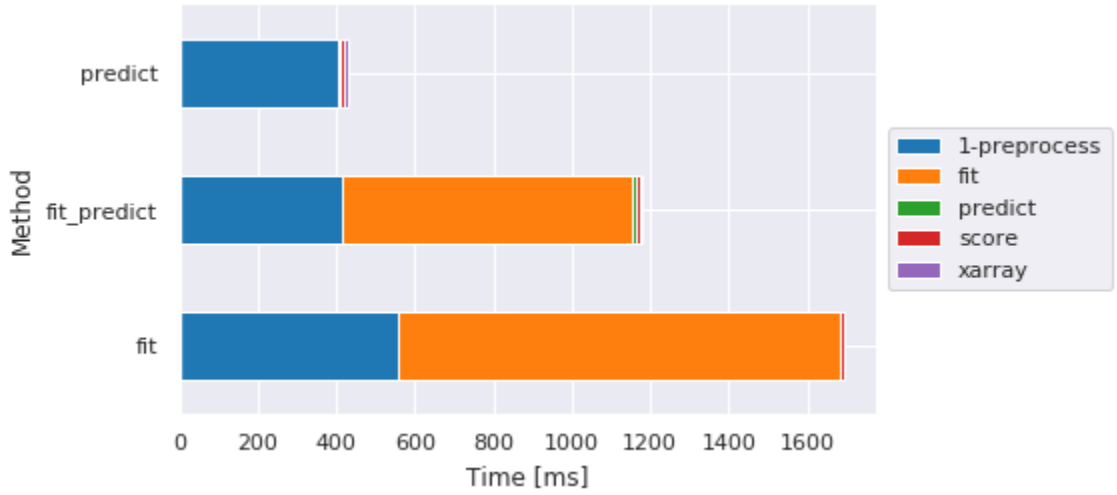
[7]:
```
fig, ax, df = m.plot.timeit(group='Method', split='Sub-method', style='darkgrid') #␣
↪Default group/split
df
```

[7]:
```
Sub-method   1-preprocess          fit   predict      score    xarray
Method
fit            556.826353  1123.903990       NaN  10.521889       NaN
```

(continues on next page)

(continued from previous page)

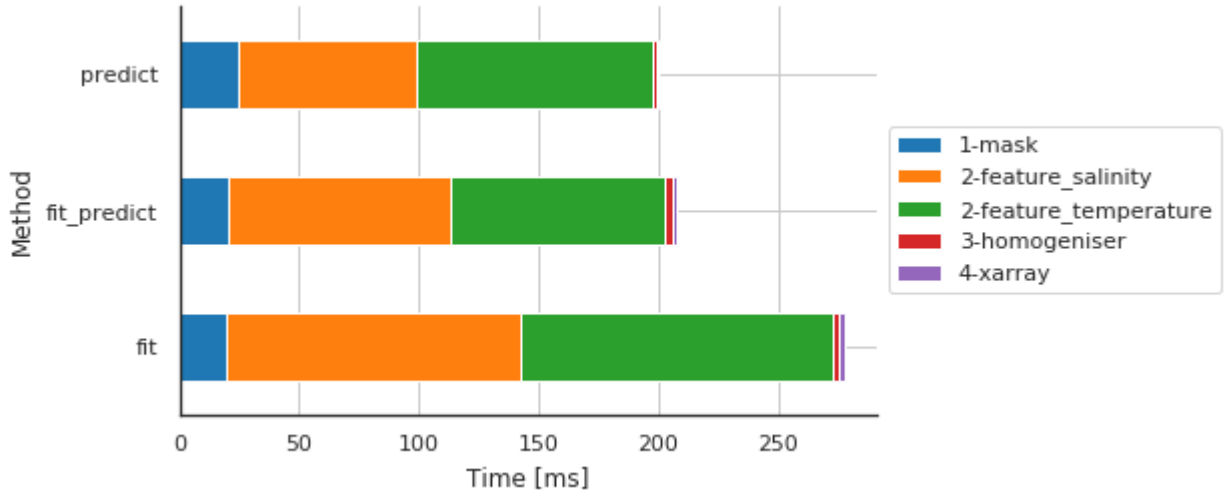
```
fit_predict    416.832924   737.503052  9.073734   8.855104  8.368969
predict        402.269125          NaN  9.886980   9.831905  8.663177
```



Preprocessing main steps by method

[8]:
```
fig, ax, df = m.plot.timeit(group='Method', split='Sub-sub-method')
df
```

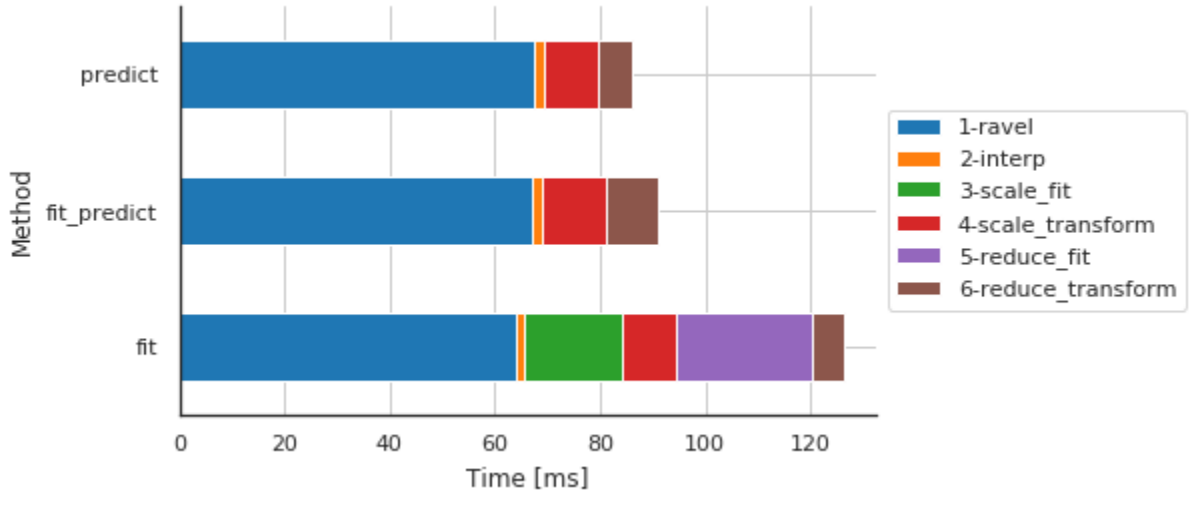
[8]:
```
Sub-sub-method     1-mask  2-feature_salinity  2-feature_temperature  \
Method
fit             19.836187          122.781515             130.059242
fit_predict     20.085096           93.130827              89.278936
predict         24.972916           73.968887              98.288298

Sub-sub-method  3-homogeniser  4-xarray
Method
fit                  2.494097  2.145052
fit_predict          3.419876  1.243114
predict              1.749992  1.240015
```



Preprocessing details by method

[9]:
```
fig, ax, df = m.plot.timeit(group='Method', split='Sub-sub-sub-method')
df
```

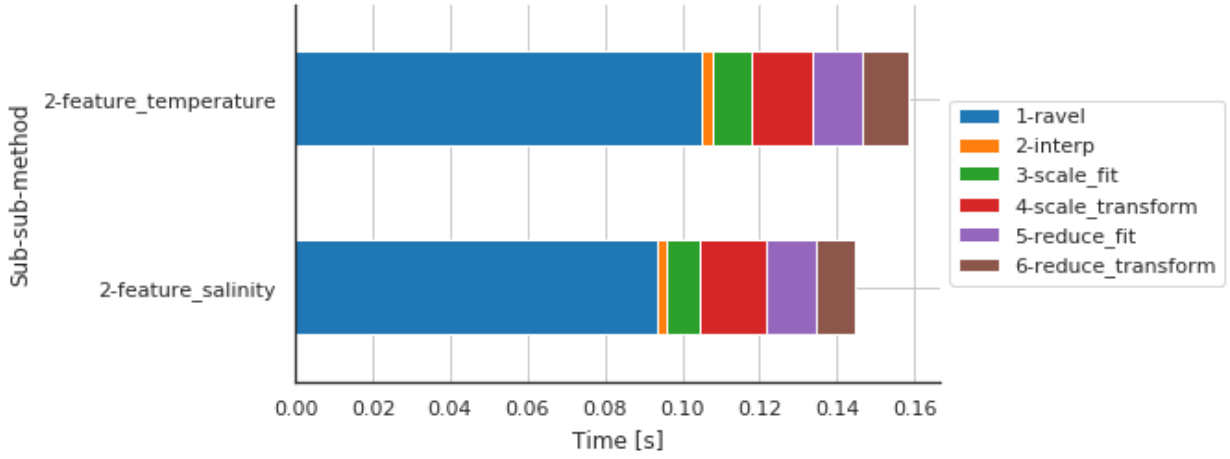
[9]:
```
Sub-sub-sub-method    1-ravel  2-interp  3-scale_fit  4-scale_transform  \
Method
fit                 63.939333  1.705170    18.492460          10.242701
fit_predict         67.147017  1.942873     0.005245          12.181759
predict             67.429066  1.815319     0.004053          10.471821

Sub-sub-sub-method  5-reduce_fit  6-reduce_transform
Method
fit                    25.922775            5.970001
fit_predict             0.003815            9.781122
predict                 0.003099            6.281853
```



Preprocessing details by features

[10]:
```
fig, ax, df = m.plot.timeit(split='Sub-sub-sub-method', group='Sub-sub-method', unit='s')
df
```

[10]:
```
Sub-sub-sub-method      1-ravel  2-interp  3-scale_fit  4-scale_transform  \
Sub-sub-method
2-feature_salinity     0.093312  0.002782     0.008571           0.017127
2-feature_temperature  0.105203  0.002681     0.009930           0.015770

Sub-sub-sub-method     5-reduce_fit  6-reduce_transform
Sub-sub-method
2-feature_salinity         0.012878            0.010059
2-feature_temperature      0.013052            0.011974
```



Help & reference

- *Bibliography*
- *What's New*

1.9 Bibliography

- Maze, Guillaume and Mercier, Herlé and Fablet, Ronan and Tandeo, Pierre and Lopez Radcenco, Manuel and Lenca, Philippe and Feucher, Charlène and Le Goff, Clément. Coherent heat patterns revealed by unsupervised classification of Argo temperature profiles in the North Atlantic Ocean. *Progress in Oceanography* (2017). 10.1016/j.pocean.2016.12.008.
- Maze, Guillaume and Mercier, Herl'e and Cabanes, C'ecile. Profile Classification Models. *Mercator Ocean Journal* (2017). ` <https://archimer.ifremer.fr/doc/00387/49816/>`__.
- Jones, Daniel C. and Holt, Harry J. and Meijers, Andrew J. S. and Shuckburgh, Emily. Unsupervised Clustering of Southern Ocean Argo Float Temperature Profiles. *Journal of Geophysical Research: Oceans* (2019). 10.1029/2018jc014629.
- Rosso, Isabella and Mazloff, Matthew R. and Talley, Lynne D. and Purkey, Sarah G. and Freeman, Natalie M. and Maze, Guillaume. Water Mass and Biogeochemical Variability in the Kerguelen Sector of the Southern Ocean: A Machine Learning Approach for a Mixing Hot Spot. *Journal of Geophysical Research: Oceans* (2020). https://doi.org/10.1029/2019JC015877.

- Boehme, Lars and Rosso, Isabella. Classifying Oceanographic Structures in the Amundsen Sea, Antarctica. *Geophysical Research Letters* (2021). https://doi.org/10.1029/2020GL089412.
- Thomas, S. D. A. and Jones, D. C. and Faul, A. and Mackie, E. and Pauthenet, E.. Defining Southern Ocean fronts using unsupervised classification. *Ocean Science* (2021). 10.5194/os-17-1545-2021.
- Li, Xiaojuan and Mao, Zhihua and Zheng, Hongrui and Zhang, Wei and Yuan, Dapeng and Li, Youzhi and Wang, Zheng and Liu, Yunxin. Process-Oriented Estimation of Chlorophyll-a Vertical Profile in the Mediterranean Sea Using MODIS and Oceanographic Float Products. *Frontiers in Marine Science* (2022). 10.3389/fmars.2022.933680.
- Sambe, Fumika and Suga, Toshio. Unsupervised Clustering of Argo Temperature and Salinity Profiles in the Mid-Latitude Northwest Pacific Ocean and Revealed Influence of the Kuroshio Extension Variability on the Vertical Structure Distribution. *Journal of Geophysical Research: Oceans* (2022). https://doi.org/10.1029/2021JC018138.

1.10 What's New

1.10.1 Upcoming release

- Fix bug with last numpy and integer management **:issue:`42`**. (**:pr:`43`**) by G. Maze

1.10.2 v0.4.1 (21 Feb. 2020)

- Improved documentation
- Improved unit testing
- **Bug fix:**
  - Fix a bug in the preprocessing step using dask_ml bakend that would cause an error for data already in dask arrays

1.10.3 v0.4.0 (1 Nov. 2019)

Warning: The API has changed, break backward compatibility.

- Enhancements:
  - Multiple-features classification
  - ND-Array classification (so that you can classify directly profiles from gridded products, eg: latitude/longitude/time grid, and not only a collection of profiles already in 2D array)
  - pyXpcm methods can be accessed through the `xarray.Dataset` accessor namespace `pyxpcm`
  - Allow to choose statistic backends (sklearn, dask_ml or user-defined)
  - Save/load PCM to/from netcdf files
- pyXpcm now consumes xarray/dask objects all along, not only on the user front-end. This add a small overhead with small dataset but allows for PCM to handle large and more complex datasets.

1.10.4 v0.3 (5 Apr. 2019)

- Removed support for python 2.7
- Added more data input consistency checks
- Fix bug in interpolation and plotting methods
- Added custom colormap and colorbar to plot module

1.10.5 v0.2 (26 Mar. 2019)

- Upgrade to python 3.6 (compatible 2.7)
- Added test for continuous coverage
- Added score and bic methods
- Improved vocabulary consistency in methods

1.10.6 v0.1.3 (12 Nov. 2018)

- Initial release.

1.11 API reference

This page provides an auto-generated summary of pyXpcm's API. For more details and examples, refer to the relevant chapters in the main part of the documentation.

1.11.1 Top-level PCM functions

Creating a PCM

pcm(K, features[, scaling, reduction, ...])	Profile Classification Model class constructor
pyxpcm.load_netcdf(ncfile)	Load a PCM model from netcdf file

pyxpcm.pcm

class pyxpcm.**pcm**(*K: int, features: {}, scaling=1, reduction=1, maxvar=15, classif='gmm', covariance_type='full', verb=False, debug=False, timeit=False, timeit_verb=False, chunk_size='auto', backend='sklearn'*)

Profile Classification Model class constructor

Consume and return `xarray` objects

__init__(*K: int, features: {}, scaling=1, reduction=1, maxvar=15, classif='gmm', covariance_type='full', verb=False, debug=False, timeit=False, timeit_verb=False, chunk_size='auto', backend='sklearn'*)

Create the PCM instance

Parameters

K: int
The number of class, or cluster, in the classification model.

features: dict()
The vertical axis to use for each features. eg: {'temperature':np.arange(-2000,0,1)}

scaling: int (default: 1)
Define the scaling method:

- 0: No scaling
- **1: Center on sample mean and scale by sample std**
- 2: Center on sample mean only

reduction: int (default: 1)
Define the dimensionality reduction method:

- 0: No reduction
- **1: Reduction using :class:`sklearn.decomposition.PCA`**

maxvar: float (default: 99.9)
Maximum feature variance to preserve in the reduced dataset using `sklearn.decomposition.PCA`. In %.

classif: str (default: 'gmm')
Define the classification method. The only method available as of now is a Gaussian Mixture Model. See `sklearn.mixture.GaussianMixture` for more details.

covariance_type: str (default: 'full')
Define the type of covariance matrix shape to be used in the default classifier GMM. It can be 'full' (default), 'tied', 'diag' or 'spherical'.

verb: boolean (default: False)
More verbose output

timeit: boolean (default: False)
Register time of operation for performance evaluation

timeit_verb: boolean (default: False)
Print time of operation during execution

chunk_size: 'auto' or int
Sampling chunk size, (array of features after pre-processing)

backend: str
Statistic library backend, 'sklearn' (default) or 'dask_ml'

Methods

__init__(K, features[, scaling, reduction, ...])	Create the PCM instance
bic(ds[, features, dim])	Compute Bayesian information criterion for the current model on the input dataset
display([deep])	Display detailed parameters of the PCM This is not a get_params because it doesn't return a dictionary Set Boolean option 'deep' to True for all properties display
fit(ds[, features, dim])	Estimate PCM parameters
fit_predict(ds[, features, dim, inplace, name])	Estimate PCM parameters and predict classes.
predict(ds[, features, dim, inplace, name])	Predict labels for profile samples
predict_proba(ds[, features, dim, inplace, ...])	Predict posterior probability of each components given the data
preprocessing(ds[, features, dim, action, mask])	Dataset pre-processing of feature(s)
preprocessing_this(da[, dim, feature_name, ...])	Pre-process data before anything
ravel(da[, dim, feature_name])	Extract from N-d array a X(feature,sample) 2-d array and vertical dimension z
score(ds[, features, dim])	Compute the per-sample average log-likelihood of the given data
to_netcdf(ncfile, **ka)	Save a PCM to a netcdf file
unravel(ds, sampling_dims, X)	Create a DataArray from a numpy array and sampling dimensions

Attributes

F	Return the number of features
K	Return the number of classes
backend	Return the name of the statistic backend
features	Return features definition dictionnary
fitstats	Estimator fit properties
plot	Access plotting functions
stat	Access statistics functions
timeit	Return a pandas.DataFrame with Execution time of method called on this instance

pyxpcm.load_netcdf

pyxpcm.**load_netcdf**(*ncfile*)

Load a PCM model from netcdf file

Parameters

ncfile
[str] File name from which to load a PCM.

Attributes

pcm.K	Return the number of classes
pcm.F	Return the number of features
pcm.features	Return features definition dictionnary

pyxpcm.pcm.K

property pcm.**K**

Return the number of classes

pyxpcm.pcm.F

property pcm.**F**

Return the number of features

pyxpcm.pcm.features

property pcm.**features**

Return features definition dictionnary

Computation

pcm.fit(ds[, features, dim])	Estimate PCM parameters
pcm.fit_predict(ds[, features, dim, ...])	Estimate PCM parameters and predict classes.
pcm.predict(ds[, features, dim, inplace, name])	Predict labels for profile samples
pcm.predict_proba(ds[, features, dim, ...])	Predict posterior probability of each components given the data
pcm.score(ds[, features, dim])	Compute the per-sample average log-likelihood of the given data
pcm.bic(ds[, features, dim])	Compute Bayesian information criterion for the current model on the input dataset

pyxpcm.pcm.fit

pcm.**fit**(*ds*, *features=None*, *dim=None*)

Estimate PCM parameters

For a PCM, the fit method consists in the following operations:

- **pre-processing**
  - interpolation to the `feature_axis` levels of the model
  - scaling
  - reduction

- estimate classifier parameters

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

Returns

self

pyxpcm.pcm.fit_predict

`pcm.fit_predict(`*ds*, *features=None*, *dim=None*, *inplace=False*, *name='PCM_LABELS'*`)`

Estimate PCM parameters and predict classes.

This method add these properties to the PCM object:

- `llh`: The log likelihood of the model with regard to new data

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

inplace: boolean, False by default
If False, return a `xarray.DataArray` with predicted labels If True, return the input `xarray.Dataset` with labels added as a new `xarray.DataArray`

name: string ('PCM_LABELS')
Name of the DataArray holding labels.

Returns

`xarray.DataArray`
Component labels (if option 'inplace' = False)

or
`xarray.Dataset`
Input dataset with component labels as a 'PCM_LABELS' new `xarray.DataArray` (if option 'inplace' = True)

pyxpcm.pcm.predict

pcm.**predict**(*ds*, *features=None*, *dim=None*, *inplace=False*, *name='PCM_LABELS'*)

Predict labels for profile samples

This method add these properties to the PCM object:

- `llh`: The log likelihood of the model with regard to new data

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

inplace: boolean, False by default
If False, return a `xarray.DataArray` with predicted labels If True, return the input `xarray.Dataset` with labels added as a new `xarray.DataArray`

name: str, default is 'PCM_LABELS'
Name of the `xarray.DataArray` with labels

Returns

`xarray.DataArray`
Component labels (if option 'inplace' = False)

or

`xarray.Dataset`
Input dataset with Component labels as a 'PCM_LABELS' new `xarray.DataArray` (if option 'inplace' = True)

pyxpcm.pcm.predict_proba

pcm.**predict_proba**(*ds*, *features=None*, *dim=None*, *inplace=False*, *name='PCM_POST'*, *classdimname='pcm_class'*)

Predict posterior probability of each components given the data

This method adds these properties to the PCM instance:

- `llh`: The log likelihood of the model with regard to new data

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

inplace: boolean, False by default
If False, return a `xarray.DataArray` with predicted probabilities If True, return the input `xarray.Dataset` with probabilities added as a new `xarray.DataArray`

name: str, default is 'PCM_POST'
Name of the DataArray with prediction probability (posteriors)

classdimname: str, default is 'pcm_class'
Name of the dimension holding classes

Returns

`xarray.DataArray`
Probability of each Gaussian (state) in the model given each sample (if option 'inplace' = False)

or

`xarray.Dataset`
Input dataset with Component Probability as a 'PCM_POST' new `xarray.DataArray` (if option 'inplace' = True)

pyxpcm.pcm.score

pcm.**score**(*ds*, *features=None*, *dim=None*)

Compute the per-sample average log-likelihood of the given data

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

Returns

log_likelihood: float
In the case of a GMM classifier, this is the Log likelihood of the Gaussian mixture given data

pyxpcm.pcm.bic

pcm.**bic**(*ds*, *features=None*, *dim=None*)

Compute Bayesian information criterion for the current model on the input dataset

Only for a GMM classifier

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input xarray.Dataset

Returns

bic: float
The lower the better

1.11.2 Low-level PCM properties and functions

pcm.timeit	Return a pandas.DataFrame with Execution time of method called on this instance
pcm.ravel(da[, dim, feature_name])	Extract from N-d array a X(feature,sample) 2-d array and vertical dimension z
pcm.unravel(ds, sampling_dims, X)	Create a DataArray from a numpy array and sampling dimensions

pyxpcm.pcm.timeit

property pcm.**timeit**

Return a pandas.DataFrame with Execution time of method called on this instance

pyxpcm.pcm.ravel

pcm.**ravel**(*da*, *dim=None*, *feature_name=<class 'str'>*)

Extract from N-d array a X(feature,sample) 2-d array and vertical dimension z

Parameters

da: :class:`xarray.DataArray`
The DataArray to process

dim: str
Name of the vertical dimension in the input xarray.DataArray

feature_name: str
Target PCM feature name for the input xarray.DataArray

Returns

X: xarray.DataArray
A new DataArray with dimension ['n_sampling','n_features'] Note that data are always dask.array.Array.

z: numpy.array
The vertical axis of data

sampling_dims: dict()
Dictionary where keys are xarray.Dataset variable names of features and values are another dictionary with the list of sampling dimension in DIM_SAMPLING key and the name of the vertical axis in the DIM_VERTICAL key.

Examples

This function is meant to be used internally only

__author__: gmaze@ifremer.fr

pyxpcm.pcm.unravel

pcm.**unravel**(*ds*, *sampling_dims*, *X*)

Create a DataArray from a numpy array and sampling dimensions

1.11.3 Plotting

pcm.plot	Access plotting functions

pyxpcm.pcm.plot

property pcm.**plot**

Access plotting functions

Plot PCM Contents

plot.quantile(m, da[, xlim, classdimname, ...])	Plot q-th quantiles of a dataArray for each PCM components
plot.scaler(m[, style, plot_kw, subplot_kw])	Plot PCM scalers properties
plot.reducer(m[, pcalist, style, maxcols, ...])	Plot PCM reducers properties
plot.preprocessed(m, ds[, features, dim, n, ...])	Plot preprocessed features as pairwise scatter plots
plot.timeit(m[, group, split, subplot_kw, ...])	Plot PCM registered timing of operations

pyxpcm.plot.quantile

pyxpcm.plot.**quantile**(*m*, *da*, *xlim=None*, *classdimname='pcm_class'*, *quantdimname='quantile'*, *maxcols=3*, *cmap=None*, *ylabel='feature dimension'*, ***kwargs*)

Plot q-th quantiles of a dataArray for each PCM components

Parameters

m
[*pyxpcm.pcm* instance]

da: :class:`xarray.DataArray` with quantiles
xlim
classdimname
quantdimname
maxcols

Returns

fig
[matplotlib.pyplot.figure.Figure]

ax
[matplotlib.axes.Axes object or array of Axes objects.] *ax* can be either a single matplotlib.axes.Axes object or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the squeeze keyword.

pyxpcm.plot.scaler

pyxpcm.plot.**scaler**(*m*, *style='whitegrid'*, *plot_kw=None*, *subplot_kw=None*, ***kwargs*)

Plot PCM scalers properties

Parameters

:class:`pyxpcm.pcm` instance

pyxpcm.plot.reducer

pyxpcm.plot.**reducer**(*m*, *pcalist=None*, *style='whitegrid'*, *maxcols=inf*, *plot_kw=None*, *subplot_kw=None*, ***kwargs*)

Plot PCM reducers properties

pyxpcm.plot.preprocessed

pyxpcm.plot.**preprocessed**(*m*, *ds*, *features=None*, *dim=None*, *n=1000*, *kde=False*, *style='darkgrid'*, ***kargs*)

Plot preprocessed features as pairwise scatter plots

Require seaborn

Parameters

:class:`pyxpcm.pcm` instance
ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input xarray.Dataset. If not specified or set to None, features are identified using xarray.DataArray attributes 'feature_name'.

n
[int] Number of samples to use in scatter plots

Returns

g
[seaborn.axisgrid.PairGrid] Seaborn Pairgrid instance

__author__: gmaze@ifremer.fr

pyxpcm.plot.timeit

pyxpcm.plot.**timeit**(*m*, *group='Method'*, *split='Sub-method'*, *subplot_kw=None*, *style='white'*, *unit='ms'*, ***kwargs*)

Plot PCM registered timing of operations

Parameters

group='Method',
split='Sub-method',
subplot_kw=None, style='white'
unit='s'

Returns

fig, ax, df

Tools

plot.cmap(m, name[, palette, usage])	Return categorical colormaps
plot.colorbar(m[, cmap])	Add a colorbar to the current plot with centered ticks on discrete colors
plot.subplots(m[, maxcols, K, subplot_kw])	Return (figure, axis) with one subplot per cluster
plot.latlongrid(ax[, dx, dy, fontsize])	Add latitude/longitude grid line and labels to a cartopy geoaxes

pyxpcm.plot.cmap

pyxpcm.plot.**cmap**(*m*, *name*, *palette=False*, *usage='class'*)

Return categorical colormaps

Parameters

name
[str] Name of the colormap, ex: 'Set2'

palette
[bool] Whether to return a Seaborn color palette or not.

- False (default): function returns a :class:`matplotlib.colors.LinearSegmentedColormap`
- True: function returns a :func:`seaborn.color_palette`

usage
[str]

The intended usage of the colormap, this can be:

- 'class' (default): one color per class
- 'robustness' : a 5-colors for probability ranges

Returns

:class:`matplotlib.colors.LinearSegmentedColormap` or
:func:`seaborn.color_palette`

pyxpcm.plot.colorbar

pyxpcm.plot.**colorbar**(*m*, *cmap=None*, ***kwargs*)

Add a colorbar to the current plot with centered ticks on discrete colors

Parameters

:class:`pyxpcm.models.pcm`
A PCM instance

cmap
[:class:`matplotlib.colors.LinearSegmentedColormap` or :func:`seaborn.color_palette`]

Returns

:class:`matplotlib.pyplot.colorbar`

pyxpcm.plot.subplots

pyxpcm.plot.**subplots**(*m*, *maxcols=3*, *K=inf*, *subplot_kw=None*, ***kwargs*)

Return (figure, axis) with one subplot per cluster

Parameters

:class:`pyxpcm.models.pcm`
A PCM instance

maxcols
[int] Maximum number of columns to use

K
[int] The number of subplot required (`pyxpcm.models.pcm.K()` by default)

subplot_kw
[dict()] Arguments to be submitted to the `matplotlib.pyplot.subplots` subplot_kw options.

All other **kwargs are forwarded to :class:`matplotlib.pyplot.subplots`

Returns

fig
[`matplotlib.pyplot.figure.Figure`]

ax
[`matplotlib.axes.Axes` object or array of Axes objects.] *ax* can be either a single `matplotlib.axes.Axes` object or an array of Axes objects if more than one subplot was created. The dimensions of the resulting array can be controlled with the squeeze keyword, see above.

Examples

fig, ax = m.plot.subplots(maxcols=4, sharey=True, figsize=(12,6))

__author__: gmaze@ifremer.fr

pyxpcm.plot.latlongrid

pyxpcm.plot.**latlongrid**(*ax*, *dx=5.0*, *dy=5.0*, *fontsize=6*, ***kwargs*)

Add latitude/longitude grid line and labels to a cartopy geoaxes

1.11.4 Statistics

pcm.stat	Access statistics functions
stat.quantile(ds[, q, of, using, outname, ...])	Compute q-th quantile of a xarray.DataArray for each PCM components
stat.robustness(ds[, name, classdimname, ...])	Compute classification robustness
stat.robustness_digit(ds[, name, ...])	Digitize classification robustness

pyxpcm.pcm.stat

property pcm.**stat**

Access statistics functions

pyxpcm.stat.quantile

pyxpcm.stat.**quantile**(*ds*, *q=0.5*, *of=None*, *using='PCM_LABELS'*, *outname='PCM_QUANT'*, *keep_attrs=False*)

Compute q-th quantile of a xarray.DataArray for each PCM components

Parameters

q: float in the range of [0,1] (or sequence of floats)
Quantiles to compute, which must be between 0 and 1 inclusive.

of: str
Name of the xarray.Dataset variable to compute quantiles for.

using: str
Name of the xarray.Dataset variable with classification labels to use. Use 'PCM_LABELS' by default.

outname: 'PCM_QUANT' or str
Name of the xarray.DataArray with quantile

keep_attrs: boolean, False by default
Preserve of xarray.Dataset attributes or not in the new quantile variable.

Returns

xarray.Dataset with shape (K, n_quantiles, N_z=n_features)
or
xarray.DataArray with shape (K, n_quantiles, N_z=n_features)

pyxpcm.stat.robustness

pyxpcm.stat.**robustness**(*ds*, *name='PCM_POST'*, *classdimname='pcm_class'*, *outname='PCM_ROBUSTNESS'*)

Compute classification robustness

Parameters

name: str, default is 'PCM_POST'
Name of the xarray.DataArray with prediction probability (posteriors)

classdimname: str, default is 'pcm_class'
Name of the dimension holding classes

outname: 'PCM_ROBUSTNESS' or str
Name of the xarray.DataArray with robustness

inplace: boolean, False by default
If False, return a xarray.DataArray with robustness If True, return the input xarray.Dataset with robustness added as a new xarray.DataArray

Returns

xarray.Dataset if inplace=True
or
xarray.DataArray if inplace=False

pyxpcm.stat.robustness_digit

pyxpcm.stat.**robustness_digit**(*ds*, *name='PCM_POST'*, *classdimname='pcm_class'*, *outname='PCM_ROBUSTNESS_CAT'*)

Digitize classification robustness

Parameters

ds: :class:`xarray.Dataset`
Input dataset

name: str, default is 'PCM_POST'
Name of the xarray.DataArray with prediction probability (posteriors)

classdimname: str, default is 'pcm_class'
Name of the dimension holding classes

outname: 'PCM_ROBUSTNESS_CAT' or str
Name of the xarray.DataArray with robustness categories

inplace: boolean, False by default
If False, return a xarray.DataArray with robustness If True, return the input xarray.Dataset with robustness categories added as a new xarray.DataArray

Returns

xarray.Dataset if inplace=True
or
xarray.DataArray if inplace=False

1.11.5 Save/load PCM models

pcm.to_netcdf(ncfile, **ka)	Save a PCM to a netcdf file
pyxpcm.load_netcdf(ncfile)	Load a PCM model from netcdf file

pyxpcm.pcm.to_netcdf

pcm.**to_netcdf**(*ncfile*, ***ka*)

Save a PCM to a netcdf file

Any existing file at this location will be overwritten by default. Time logging information are not saved.

Parameters

ncfile
[str] File name where to save this PCM.

global_attributes: dict()
Dictionnary of attributes to add to the Netcdf4 file under the global scope.

mode
[str] Writing mode of the file. mode='w' (default) overwrite any existing file. Anything else will raise an Error if file exists.

1.11.6 Helper

tutorial.open_dataset(name)	Open a dataset from the pyXpcm online data repository (requires internet).

pyxpcm.tutorial.open_dataset

pyxpcm.tutorial.**open_dataset**(*name*)

Open a dataset from the pyXpcm online data repository (requires internet).

If a local copy is found then always use that to avoid network traffic.

Parameters

name
[str] Name of the dataset to load among:

- *dummy* (depth,sample) dummy array
- *argo* (depth,sample) real Argo data sample
- *isas_snapshot* (depth,latitude,longitude) real gridded product
- *isas_series* (depth,latitude,longitude,time) real gridded product time series

Returns

xarray.Dataset

1.11.7 Xarray *pyxpcm* name space

Provide accessor to enhance interoperability between `xarray` and `pyxpcm`.

Provide a scope named `pyxpcm` as accessor to `xarray.Dataset` objects.

class pyxpcm.xarray.**pyXpcmDataSetAccessor**

Class registered under scope `pyxpcm` to access `xarray.Dataset` objects.

add(*da*)

Add a `xarray.DataArray` to this `xarray.Dataset`

bic(*this_pcm*, ***kwargs*)

Compute Bayesian information criterion for the current model on the input dataset

Only for a GMM classifier

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

Returns

bic: float
The lower the better

drop_all()

Remove `xarray.DataArray` created with pyXpcm front this `xarray.Dataset`

feature_dict(*this_pcm*, *features=None*)

Return dictionary of features for this `xarray.Dataset` and a PCM

Parameters

pcm
[`pyxpcm.pcmmodel.pcm`]

features
[dict] Keys are PCM feature name, Values are corresponding `xarray.Dataset` variable names

Returns

dict()
Dictionary where keys are PCM feature names and values the corresponding `xarray.Dataset` variables

fit(*this_pcm*, ***kwargs*)

Estimate PCM parameters

For a PCM, the fit method consists in the following operations:

- **pre-processing**
  - interpolation to the `feature_axis` levels of the model

- scaling
- reduction

- estimate classifier parameters

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

Returns

self

`fit_predict`(*this_pcm*, ***kwargs*)

Estimate PCM parameters and predict classes.

This method add these properties to the PCM object:

- `llh`: The log likelihood of the model with regard to new data

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

inplace: boolean, False by default
If False, return a `xarray.DataArray` with predicted labels If True, return the input `xarray.Dataset` with labels added as a new `xarray.DataArray`

name: string ('PCM_LABELS')
Name of the DataArray holding labels.

Returns

`xarray.DataArray`
Component labels (if option 'inplace' = False)

or
`xarray.Dataset`
Input dataset with component labels as a 'PCM_LABELS' new `xarray.DataArray` (if option 'inplace' = True)

`mask`(*this_pcm*, *features=None*, *dim=None*)

Create a mask where all PCM features are defined

Create a mask where all feature profiles are not null over the PCM feature axis.

Parameters

:class:`pyxpcm.pcmmodel.pcm`
features
[dict()] Definitions of this_pcm features in the `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim
[str] Name of the vertical dimension in the `xarray.Dataset`. If not specified or set to None, dim is identified as the `xarray.DataArray` variables with attributes 'axis' set to 'z'.

Returns

`xarray.DataArray`

predict(*this_pcm*, *inplace=False*, ***kwargs*)

Predict labels for profile samples

This method add these properties to the PCM object:

- `llh`: The log likelihood of the model with regard to new data

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

inplace: boolean, False by default
If False, return a `xarray.DataArray` with predicted labels If True, return the input `xarray.Dataset` with labels added as a new `xarray.DataArray`

name: str, default is 'PCM_LABELS'
Name of the `xarray.DataArray` with labels

Returns

`xarray.DataArray`
Component labels (if option 'inplace' = False)

or
`xarray.Dataset`
Input dataset with Component labels as a 'PCM_LABELS' new `xarray.DataArray` (if option 'inplace' = True)

predict_proba(*this_pcm*, ***kwargs*)

Predict posterior probability of each components given the data

This method adds these properties to the PCM instance:

- `llh`: The log likelihood of the model with regard to new data

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

inplace: boolean, False by default
If False, return a `xarray.DataArray` with predicted probabilities If True, return the input `xarray.Dataset` with probabilities added as a new `xarray.DataArray`

name: str, default is 'PCM_POST'
Name of the DataArray with prediction probability (posteriors)

classdimname: str, default is 'pcm_class'
Name of the dimension holding classes

Returns

`xarray.DataArray`
Probability of each Gaussian (state) in the model given each sample (if option 'inplace' = False)

or
`xarray.Dataset`
Input dataset with Component Probability as a 'PCM_POST' new `xarray.DataArray` (if option 'inplace' = True)

`quantile`(*this_pcm*, *inplace=False*, ***kwargs*)

Compute q-th quantile of a `xarray.DataArray` for each PCM components

Parameters

q: float in the range of [0,1] (or sequence of floats)
Quantiles to compute, which must be between 0 and 1 inclusive.

of: str
Name of the `xarray.Dataset` variable to compute quantiles for.

using: str
Name of the `xarray.Dataset` variable with classification labels to use. Use 'PCM_LABELS' by default.

outname: 'PCM_QUANT' or str
Name of the `xarray.DataArray` with quantile

keep_attrs: boolean, False by default
Preserve `of` `xarray.Dataset` attributes or not in the new quantile variable.

Returns

`xarray.Dataset` **with shape (K, n_quantiles, N_z=n_features)**
or
`xarray.DataArray` **with shape (K, n_quantiles, N_z=n_features)**

`robustness`(*this_pcm*, *inplace=False*, ***kwargs*)

Compute classification robustness

Parameters

name: str, default is 'PCM_POST'
Name of the `xarray.DataArray` with prediction probability (posteriors)

classdimname: str, default is 'pcm_class'
Name of the dimension holding classes

outname: 'PCM_ROBUSTNESS' or str
Name of the `xarray.DataArray` with robustness

inplace: boolean, False by default
If False, return a `xarray.DataArray` with robustness If True, return the input `xarray.Dataset` with robustness added as a new `xarray.DataArray`

Returns

`xarray.Dataset` if inplace=True
or
`xarray.DataArray` if inplace=False

`robustness_digit`(*this_pcm*, *inplace=False*, ***kwargs*)

Digitize classification robustness

Parameters

ds: :class:`xarray.Dataset`
Input dataset

name: str, default is 'PCM_POST'
Name of the `xarray.DataArray` with prediction probability (posteriors)

classdimname: str, default is 'pcm_class'
Name of the dimension holding classes

outname: 'PCM_ROBUSTNESS_CAT' or str
Name of the `xarray.DataArray` with robustness categories

inplace: boolean, False by default
If False, return a `xarray.DataArray` with robustness If True, return the input `xarray.Dataset` with robustness categories added as a new `xarray.DataArray`

Returns

`xarray.Dataset` if inplace=True
or
`xarray.DataArray` if inplace=False

`sampling_dim`(*this_pcm*, *features=None*, *dim=None*)

Return the list of dimensions to be stacked for sampling

Parameters

pcm
[*pyxpcm.pcm*]

features
[None (default) or dict()] Keys are PCM feature name, Values are corresponding `xarray.Dataset` variable names. It set to None, all PCM features are used.

dim
[None (default) or str()] The `xarray.Dataset` dimension to use as vertical axis in all features. If set to None, it is automatically set to the dimension with an attribute `axis` set to Z.

Returns

dict()
Dictionary where keys are `xarray.Dataset` variable names of features and values are another dictionary with the list of sampling dimension in DIM_SAMPLING key and the name of the vertical axis in the DIM_VERTICAL key.

`score`(*this_pcm*, ***kwargs*)

Compute the per-sample average log-likelihood of the given data

Parameters

ds: :class:`xarray.Dataset`
The dataset to work with

features: dict()
Definitions of PCM features in the input `xarray.Dataset`. If not specified or set to None, features are identified using `xarray.DataArray` attributes 'feature_name'.

dim: str
Name of the vertical dimension in the input `xarray.Dataset`

Returns

log_likelihood: float
In the case of a GMM classifier, this is the Log likelihood of the Gaussian mixture given data

`split`()

Split pyXpcm variables from the original `xarray.Dataset`

Returns

list of `xarray.Dataset`, `xarray.Dataset`
Two DataSest: one with pyXpcm variables, one with the original DataSet

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

F

K

L

M

O

P

Q

R

S

T

U