OpenMORe

model order reduction for reacting flows

Software documentation

Contents

1 Introduction **1**

1.1 General purpose . . . 1

1.2 Copyright and warranty . . . 1

1.2.1 Copyright . . . 1

1.2.2 Warranty . . . 1

1.2.3 Contacts . . . 1

1.3 Folders' organization . . . 1

2 Installation **3**

2.1 Prerequisites . . . 3

2.2 Installation via command line . . . 3

2.3 Testing . . . 3

3 Modules, classes, properties and methods **5**

3.1 `model_order_reduction.py` . . . 5

3.1.1 class: PCA . . . 5

3.1.2 class: LPCA . . . 8

3.1.3 class: KPCA . . . 9

3.1.4 class: variables_selection . . . 10

3.1.5 class: SamplePopulation . . . 11

3.1.6 class: NMF . . . 11

3.2 `clustering.py` . . . 14

3.2.1 class: lpca . . . 14

3.2.2 class: fpca . . . 15

3.2.3 class: KMeans . . . 16

3.2.4 class: multistageLPCA . . . 16

3.2.5 class: spectralClustering . . . 17

3.3 `classification.py` . . . 18

3.3.1 class: VQPCA . . . 18

3.4 `utilities.py` . . . 19

3.4.1 function: evaluate_clustering_DB . . . 19

3.4.2 function: evaluate_clustering_PHC . . . 19

4 Examples **20**

4.1 Dimensionality reduction and Feature Extraction via PCA . . . 20

4.2 Clustering via Local Principal Component Analysis . . . 24

4.3 Spectral Clustering . . . 27

4.4 Outlier identification and removal via PCA . . . 28

4.5 Set automatically the optimal number of clusters via Davies-Bouldin index . . . 30

1 Introduction

1.1 General purpose

OpenMORe is a python software for model-order-reduction, clustering and classification tasks. Different techniques are implemented i.e., Principal Component Analysis (PCA), Local Principal Component Analysis (LPCA), Kernel Principal Component Analysis (KPCA), Non-negative Matrix Factorization (NMF) for model order reduction, and Local Principal Component Analysis (LPCA), K-means (KM), Spectral Clustering (SC) for clustering and classification. Moreover, a wide range of useful functions for other machine-learning tasks, such as data scaling, matrix sampling, clustering evaluation, multivariate outliers removal and feature selection are implemented.

1.2 Copyright and warranty

1.2.1 Copyright

1.2.2 Warranty

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

1.2.3 Contacts

- Giuseppe D'Alessio: `giuseppe.dalessio@ulb.ac.be`
- Alberto Cuoci: `alberto.cuoci@polimi.it`
- Alessandro Parente: `alessandro.parente@ulb.be`

1.3 Folders' organization

OpenMORe consists of several folders, each of them having a different purpose, and being organized as follows:

- *OpenMORe:* this folder contains the source code, i.e., four python modules specifically conceived to accomplish different tasks:

    1. `model_order_reduction.py`;
    2. `clustering.py`;
    3. `classification.py`;
    4. `utilities.py`.

- *Tests:* this folder contains the files to test the different code functionalities, as well as the correct installation of the libraries.

- *Data:* the aim of this folder is to provide data in *.csv* format to the user, to test the different code functionalities. On its turn, it contains two additional folders: **dummy_data/** and **reactive_flow/**. In the first one, four simple dataset are contained of which three accounting for 300 samples each, and previously generated by means of the *sklearn.dataset* functions: `make_moons`, `make_blobs`, `make_circles`, and a last one containing 2D data with synthetic outliers. In the **reactive_flow/** folder, instead, it is possible to find data obtained from a RANS simulation of Sandia flameD, using a detailed kinetic mechanisms accounting for 36 chemical species. The file *flameD.csv* consist of the thermochemical space, i.e., temperature (first column) and the 36 species (ordered in the same order of the mechanism, as in the file *labels.csv*). The file *fluid_var.csv*, instead, consist of the fluid-dynamics variables, i.e., three velocity components (u_x, u_y, u_z), turbulent kinetic energy (k) and rate of dissipation of turbulent kinetic energy (ϵ). The file *mesh.csv* consist of the grid points used for the corresponding CFD simulation. This file is useful to visualize, for example, the clustering solution.

- *Examples:* in this folder, several examples of the different code's functionalities are contained.

2 Installation

2.1 Prerequisites

In order to successfully install the OpenMORe libraries, the following prerequisites must be satisfied:

- Python version $\geq$ 3.6 (*mandatory*)
- Numpy version $\geq$ 1.15.1 (*mandatory*)
- Scipy version $\geq$ 1.1.0 (*mandatory*)
- Matplotlib version $\geq$ 2.2.3 (*mandatory*)
- Scikit-learn version $\geq$ 0.19.1 (*only needed for Kernel-PCA*)

2.2 Installation via command line

If the prerequisites are satisfied, it is possible to proceed to the software installation as follows:

- Open the terminal, and clone the Github repository on your PC:

```
$ git clone https://github.com/gdalessi/OpenMORe.git
```

- Go to the OpenMORe folder where the file 'setup.py' is contained, and then install the software:

```
$ cd OpenMORe/
$ python setup.py install
```

If no error messages are displayed, the OpenMORe libraries were correctly installed and are now ready to be used.

2.3 Testing

To check if all the libraries' functionalities are correctly working, it is possible to use the tests in the **tests/** folder. The latter are based on the python built-in (at least for versions $\geq$ 3.6) *unittest* library.

To run the tests, the user must proceed as follows:
- From the **OpenMORe/** folder, go to the **tests/** folder:

```
$ cd tests
```

- After that, run (separately) each of the python files inside the folder calling the *unittest* library:

```
$ python -m unittest test_PCA.py
$ python -m unittest test_NMF.py
$ python -m unittest test_sampling.py
$ python -m unittest test_clustering.py
```

If all the tests have positive response, the user should get, for each of them, a message from the terminal such as the following one:

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.113s

OK
```

3 Modules, classes, properties and methods

In this section, for each module, a detailed description of all the classes functionalities, as well as their methods, are given to the user. Moreover, the classes' properties to set are also explained in detail.

3.1 `model_order_reduction.py`

3.1.1 class: PCA

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size(**X**) = [n_{obs} $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

dictionary: a dictionary containing all the instructions to be given to the properties (*optional*, see Section 4 or the source code for more information).

Properties:

@eigens:
Input type: *int.*
Description: Set the number of Principal Components (PCs) to retain, i.e., the final dimensionality of the problem.
Dictionary entry: "number_of_eigenvectors".

@to_center:
Input type: *boolean.*
Description: Enable the preprocessing method to center the matrix' **X** observations. In this way, all the observations can be seen as fluctuations from a given value. It is recommended to center if the data are multivariate.
Dictionary entry: "to_center"

@centering:
Input type: *string.*
Description: Set the centering method. Available choices are "mean" or "min": in the first case, the mean value of each variable is used as centering factor, in the second case, instead, its minimum value.
Dictionary entry: "centering_method"

@to_scale:
Input type: *boolean.*
Description: Enable the preprocessing method to scale the matrix' **X** observations. This is a mandatory operation if the dataset is multivariate, i.e., the variables have different units and ranges, to get reliable results.
Dictionary entry: "to_scale"

@scaling:
Input type: *string.*

Description: Set the scaling method. Available choices are "auto", "vast", "range" or "pareto". For additional info about the aforementioned scaling criteria: *Parente, Alessandro, and James C. Sutherland. Combustion and flame 160.2 (2013): 340-350.*
Dictionary entry: "scaling_method"

@plot_explained_variance:
Input type: *boolean.*
Description: Enable the methods to plot cumulative curve of the explained variance by the PCs, and the original data variance being explained by the selected dimensionality.
Dictionary entry: "enable_plot_variance"

@set_PCs_method:
Input type: *string.*
Description: This property is required by the method which is in charge of automatically select the number of PCs on the basis of the explained variance, or by means of the Normalized Root Mean Square Error (NRMSE). In the first case, it has to be set as "var", otherwise as "nrmse".
Dictionary entry: "set_criterion_autoPCs"

@set_explained_variance_perc:
Input type: *float.*
Description: Set the desired amount of data variance which has to be explained by the PCs, in case the method for the unsupervised choice is activated. A value $\geq 95\%$ is recommended.
Dictionary entry: "variance_to_explain"

@set_num_to_plot:
Input type: *int.*
Description: Select the PC number whose weights have to be visualized on a bar plot.
Dictionary entry: "variable_to_plot"

Methods:

fit()
Description: Perform the dimensionality reduction via PCA on the centered and scaled training data matrix **X_tilde**, with first "q" PCs being retained. The eigenvectors are already returned in decreasing order of importance, i.e., the magnitude of the associated eigenvalues decreases.

Returns: evecs: eigenvectors from the covariance matrix decomposition (PCs), evals: eigenvalues from the covariance matrix decomposition (lambda).

recover()
Description: Reconstruct the original matrix from the reduced PCA-manifold, using the prescribed number "q" of Principal Components.

Returns: **X_rec**: uncentered and unscaled reconstructed matrix with the selected number of PCA eigenvectors.

get_explained()

Description: Assess the variance explained by the first "q" retained Principal Components. This information is important to know if the percentage of explained variance is enough, or additional PCs must be retained. For many applications it is recommended a percentage of explained variance $\geq 95\%$.

Returns: explained: percentage of explained variance.

get_scores()

Description: Project the centered and scaled data matrix on the low-dimensional manifold spanned by the first "q" Principal Components, and obtain the scores matrix **Z**.

Returns: **Z**: scores matrix.

set_PCs()

Description: Automatically assess the number of PCs to be retained. This can be done in two different ways, the first one is the explained variance criterion: retain "q" PCs such that the 95% of the original data variance is explained; the second one is the nrmse criterion: retain "q" PCs such that the difference between the original and the reconstructed matrix is lower than the 10%.

Returns: optimalPCs: number of PCs required to satisfy the chosen criterion.

plot_PCs()

Description: Plot the variables' weights on a selected Principal Component (PC). The PCs are linear combination of the original variables: examining the weights, especially for the PCs associated with the largest eigenvalues, can be important for feature extraction and data-analysis purposes because it is possible to associate a PC to a particular physical phenomenon.

Returns: Plot.

plot_parity()

Description: Print the parity plot of the reconstructed variable profile from the PCA manifold. The more the scatter (i.e., the black dots) is in line with the red line, the better the reconstruction is.

Returns: Plot.

outlier_removal_leverage()

Description: Identify and remove leverage multivariate outliers eventually contained in the training dataset via PCA, examining the data projection on the PCA manifold (i.e., the scores), and measuring the score distance from the manifold center. This kind of outliers is characterized by a very high distance from the projected center of mass, and once detected they can be easily removed constructing a Cumulative Density Function (CDF). Additional info on outlier identification and removal can be found here: *Jolliffe, I. T. "Principal component analysis."; pag 237 — formula (10.1.2):*

Returns: **X_tilde′**: the matrix without leverage outliers; bin: the "clusters" of the CDF; mask: the vector containing the ID of the non-outlier observations.

outlier_removal_orthogonal()

Description: Identify and remove orthogonal multivariate outliers eventually contained in the training dataset via PCA, examining the reconstruction error, ϵ. This kind of outliers is characterized by very high distance from the manifold (i.e., large ϵ), and once detected they can be easily removed constructing a Cumulative Density Function (CDF). Additional info on outlier identification and removal can be found here: *Hubert, Mia, Peter Rousseeuw, and Tim Verdonck. Computational StatisticsData Analysis 53.6 (2009):2264-2274.*

Returns: **X_tilde′**: the matrix without orthogonal outliers; bin: the "clusters" of the CDF; mask: the vector containing the ID of the non-outlier observations.

outlier_removal_multistep()

Description: Removes outliers via PCA. Firstly, the input matrix is trimmed by means of the calculation of the Mahalanobis distance: a very small percentage (0.01 - 0.1%) of the observations are discarded. After that, an iterative algorithm is started for outlier removal based on PCA, and the convergence is reached when the data kurtosis' change is below a fixed threshold between two consecutive iterations. Additional info on the iterative algorithm and its validation can be found here: *Parente, Alessandro, and James C. Sutherland. Combustion and flame 160.2 (2013): 340-350.*

Returns: **X_tilde′**: the matrix without outliers.

3.1.2 class: LPCA

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size(**X**) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

dictionary: a dictionary containing all the instructions to be given to the properties (*optional*, see Section 4 or the source code for more information).

Properties:

This class inherits all the properties from the class:PCA, whose detailed description can be found in Section 3.1.1. In addition to those, it has:

@path_to_idx:

Input type: *string.*
Description: Specify the path where the file "idx.txt", containing the partitioning solution, is located. Warning: it is mandatory to call the partitioning file to be loaded "idx.txt".
Dictionary entry: "path_to_idx"

@clust_to_plot:

Input type: *int.*
Description: Set the number of the cluster whose PC has to be plotted.
Dictionary entry: "clust_to_plot"

Methods:
This class inherits all the methods from the class:PCA, whose detailed description can be found in Section 3.1.1. In addition to those, it has:

fit()

Description: Compute the LPCs (Local Principal Components), the u_scores (the projection of the training data on the local manifold), and the eigenvalues in each cluster, given a previous clustering solution.

Returns: LPCs: LPCs in each cluster; u_scores: scores in each cluster; Leigen: eigenvalues in each cluster; centroids: centroids of each cluster.

recover()

Description: Reconstruct the original matrix from the "k" local reduced PCA-manifolds. Given the idx vector, for each cluster the points are reconstructed from the local manifolds spanned by the LPCs.

Returns: $\mathbf{X_{rec}}$: uncentered/unscaled matrix reconstructed by means of the LPCs.

3.1.3 class: KPCA

Warning (i): in light of the affinity matrix' dimensions, this algorithm can be very slow and its application could not be feasible for data consisting of a large number of observations.
Warning (ii): for the computation and the centering of the Kernel the scikit-learn libraries are required.
Warning (iii): to speed-up the decomposition, a "fast-SVD" algorithm has been adopted[1]. Thus, a minimal loss of accuracy in the PCs computation can be observed.

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size($\mathbf{X}$) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

dictionary: a dictionary containing all the instructions to be given to the properties (*optional*, see Section 4 or the source code for more information).

Properties:
This class inherits all the setters from the class:PCA, whose detailed description can be found in Section 3.1.1. In addition to those, it has:

@kernel_type:
Input type: *string.*
Description: Set the function to be used for the computation of the affinity matrix. The available choices are: "rbf" (radial basis function) and "polynomial" (3rd order).
Dictionary entry: "selected_kernel"

[1] Halko, Nathan, et al. SIAM Journal on Scientific computing 33.5 (2011): 2580-2594.

Methods:
This class inherits all the methods from the class:PCA, whose detailed description can be found in Section 3.1.1. In addition to those, it has:

fit()

Description: Compute the KPCs (Kernel Principal Components) obtained from the decomposition of the affinity matrix.

Returns: $\underline{\mathbf{Z}}$: Kscores; $\underline{\mathbf{A}}$: KPCs; $\underline{Sigma}$: Singular values (approximated) from the decomposition of the affinity matrix.

3.1.4 class: variables_selection

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size(**X**) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

dictionary: a dictionary containing all the instructions to be given to the properties (*optional*, see Section 4 or the source code for more information).

Properties:
This class inherits all the properties from the class:PCA, whose detailed description can be found in Section 3.1.1. In addition to those, it has:

@retained:
Input type: *int.*
Description: Set the number of original variables to retain, i.e., the dimensionality of the final matrix (m).
Dictionary entry: "number_of_variables"

@path_to_labels:
Input type: *string.*
Description: Set the path to the variables name file, which has to be in *.csv* format.
Dictionary entry: "path_to_labels"

@method:
Input type: *string.*
Description: Set the method to be used for the variables selection task. The implemented options are: "b2", "procustes", "procustes_rotation".
Dictionary entry: "method"

Methods:
This class inherits all the methods from the class:PCA, whose detailed description can be found in Section 3.1.1. In addition to those, it has:

fit()

Description: Apply the aforementioned algorithms for the variables selection.

Returns: <u>PVs</u>: a list with the variables names, or numbers if the variables' name is not available.

3.1.5 class: SamplePopulation

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size($\mathbf{X}$) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

dictionary: a dictionary containing all the instructions to be given to the properties (*optional*, see Section 4 or the source code for more information).

Properties:

@sampling_strategy:
Input type: *string*.
Description: Select the method to be used for the data sampling. Available choices are: "random", "kmeans", "lpca", "stratified" or "multistage".
Dictionary entry: "method"

@set_size:
Input type: *int*.
Description: Set the final number of observations of the sampled matrix.
Dictionary entry: "final_size"

@set_conditioning:
Input type: *int*.
Description: Select the number of the variable (of the matrix **X**) to be used for the conditioning in the stratified sampling strategy.
Dictionary entry: not implemented, a setter is required.

Methods:

fit()
Description: Apply the chosen strategy to sample the input matrix **X**.

Returns: $\underline{\mathbf{X}'}$: the matrix with the chosen number of observations.

3.1.6 class: NMF

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size($\mathbf{X}$) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

dictionary: a dictionary containing all the instructions to be given to the properties (*optional*, see Section 4 or the source code for more information).

Properties:

@encoding:

Input type: *int.*
Description: Set the dimensionality of the reduced space, i.e., the column of the matrix **H**.
Dictionary entry: ”number_of_features”

@to_center:

Input type: *boolean.*
Description: Enable the preprocessing method to center the matrix' **X** observations. In this way, all the observations can be seen as fluctuations. It is recommended to center if the data are multivariate.
Dictionary entry: ”center”

@centering:

Input type: *string.*
Description: Set the centering method. Available choices are ”mean” or ”min”: in the first case, the mean value of each variable is used as centering factor, in the second case, instead, its minimum value.
Dictionary entry: ”centering_method”

@to_scale:

Input type: *boolean.*
Description: Enable the preprocessing method to scale the matrix' **X** observations. This is a mandatory operation to do if the dataset is multivariate, i.e., the variables have different units and ranges.
Dictionary entry: ”scale”

@scaling:

Input type: *string.*
Description: Set the scaling method. Available choices are ”auto”, ”vast”, ”range” or ”pareto”. For additional info about the aforementioned scaling criteria: *Parente, Alessandro, and James C. Sutherland. Combustion and flame 160.2 (2013): 340-350.*
Dictionary entry: ”scaling_method”

@algorithm:

Input type: *string.*
Description: Select the algorithm to compute the matrices **H** and **W**. Two choices are available: ”als” or ”sparse”.
Dictionary entry: ”optimization_algorithm”

@method:

Input type: *string.*
Description: In case of ALS algorithm, select if it has to be ”standard”, or ”sparse”.
Dictionary entry: ”als_method”

@metric:

Input type: *float.*
Description: Select the metric which has to be used to measure the convergence criteria. Two metrics are available: ”Frobenius” and ”KLD”. If first one is chosen, the Frobenius distance between the original data matrix and the reconstructed from the low-dimensional

space is used as a metric. Otherwise, if "KLD" is chosen, the Kullback-Leibler divergence is used as a metric.
Dictionary entry: "optimization_metric"

@beta:
Input type: *float*.
Description: Set the value of beta, a parameter to control the degree of sparsity in case ALS with sparsity is selected (*optional*).
Dictionary entry: "sparsity_beta"

@eta:
Input type: *float*.
Description: Set the value of eta, a parameter to control the degree of sparsity in case ALS + sparsity is selected (*optional*).
Dictionary entry: "sparsity_eta"

Methods:

fit()
Description: Apply the chosen NMF algorithm to reduce the order of the input matrix, **X**.

Returns: $\underline{\mathbf{W}}$: the matrix matrix containing the NMF reduced basis; $\underline{\mathbf{H}}$: the matrix containing the NMF scores.

cluster()
Description: Group the observations of the input matrix **X**, depending on their NMF scores.

Returns: $\underline{\text{idx}}$: the vector containing the cluster assignment, whose shape is $[n_{obs} \times 1]$.

3.2 clustering.py

3.2.1 class: lpca

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size(**X**) = [$n_{obs} \times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

dictionary: a dictionary containing all the instructions to be given to the properties (*optional*, see Section 4 or the source code for more information).

Properties:

@clusters:
Input type: *int.*
Description: Set the number of clusters "k" to be used for the partitioning.
Dictionary entry: "number_of_clusters"

@eigens:
Input type: *int.*
Description: Set the number of Principal Components to retain in each cluster (LPCs).
Dictionary entry: "number_of_eigenvectors"

@to_center:
Input type: *boolean.*
Description: Enable the preprocessing method to center the matrix' **X** observations. In this way, all the observations can be seen as fluctuations. It is recommended to center if the data are multivariate.
Dictionary entry: "center"

@centering:
Input type: *string.*
Description: Set the centering method. Available choices are "mean" or "min": in the first case, the mean value of each variable is used as centering factor, in the second case, instead, its minimum value.
Dictionary entry: "centering_method"

@to_scale:
Input type: *boolean.*
Description: Enable the preprocessing method to scale the matrix' **X** observations with their mean value. This is a mandatory operation to do if the dataset is multivariate, i.e., the variables have different units and ranges.
Dictionary entry: "scale"

@scaling:
Input type: *string.*
Description: Set the scaling method. Available choices are "auto", "vast", "range" or "pareto". For additional info about the aforementioned scaling criteria: *Parente, Alessandro, and James C. Sutherland. Combustion and flame 160.2 (2013): 340-350.*
Dictionary entry: "scaling_method"

@initialization:

Input type: *string.*

Description: Initialize the clustering solution, i.e., the labels vector idx. The available initializations are: ”random” (assign randomly an integer between 0 and k to each observation); ”kmeans” (start the algorithm from a K-means solution); ”observations” (pick randomly ”k” observations from the dataset and use them as initial centroids); ”uniform” (divide uniformly the observations in each cluster and use it as starting solution); ”pkcia” (initialize the centroids with the method described in[2]).

Dictionary entry: ”initialization_method”

@writeFolder:

Input type: *boolean.*

Description: Allow the algorithm to create a folder where the clustering stats are collected (number of iteration, final reconstruction error etc.).

Dictionary entry: ”write_stats”

@correction:

Input type: *string.*

Description: Use a multiplicative correction to the reconstruction error for the clusters assignment. [Work in progress]

Dictionary entry: ”correction_factor”

Methods:

fit()

Description: Group the observations of the input matrix **X** on the basis of their reconstruction error.

Returns: idx: the vector containing the cluster assignment, whose shape is $(n_{obs} \times 1)$.

3.2.2 class: fpca

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size(**X**) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

conditioning: vector representing the variable which has to be used to condition the data matrix **X**.

dictionary: a dictionary containing all the instructions to be given to the properties (*optional*, see Section 4 or the source code for more information).

Methods:

[2] Manochandar, S., M. Punniyamoorthy, and R. K. Jeyachitra. Computers Industrial Engineering (2020): 106290.

condition()

Description: This method partitions the data matrix in 'k' different bins, depending on the conditioning vector interval.

Returns: idx: the vector containing the cluster assignment, whose shape is $(n_{obs} \times 1)$.

fit()

Description: Performs PCA in each conditioned bin, and then it returns the LPCs, the local eigenvalues, the local scores and the centroids.

Returns: LPCs: list containing the PCs in each cluster; u_scores: list containing the scores in each cluster; Leigen: list containing the eigenvalues in each cluster; centroids: list containing the centroids vectors in each cluster/

3.2.3 class: KMeans

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size(**X**) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

dictionary: a dictionary containing all the instructions to be given to the properties (*optional*, see Section 4 or the source code for more information).

Properties:

This class inherits all the setters from the class:PCA, whose detailed description can be found in Section 3.1.1. In addition to those, it has:

@initMode:

Input type: *bool.*
Description: Choose if the algorithm has to run with a higher tolerance for the convergence criterion. It is recommended to activate this option if the algorithm is used to initialize lpca, and to keep this property set as False otherwise.
Dictionary entry: not implemented, a setter is required.

Methods:

fit()

Description: Start the K-means clustering to partition the input data matrix in "k" bins.

Returns: idx: the vector containing the cluster assignment, whose shape is $(n_{obs} \times 1)$.

3.2.4 class: multistageLPCA

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size(**X**) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

conditioning: vector representing the variable which has to be used to condition the data matrix **X**.

Methods:

partition()

Description: Group the observations by means of the coupled fpca-vqpca algorithm.

Returns: idx: the vector containing the cluster assignment, whose shape is $(n_{obs} \times 1)$.

3.2.5 class: spectralClustering

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size(**X**) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

Properties:

@clusters:
Input type: *int.*
Description: Set the number of clusters ”k” to be used for the matrix partitioning.
Dictionary entry: ”number_of_clusters”

@to_center:
Input type: *boolean.*
Description: Enable the preprocessing method to center the matrix' **X** observations. In this way, all the observations can be seen as fluctuations. It is recommended to center if the data are multivariate.
Dictionary entry: ”center”

@centering:
Input type: *string.*
Description: Set the centering method. Available choices are ”mean” or ”min”: in the first case, the mean value of each variable is used as centering factor, in the second case, instead, its minimum value.
Dictionary entry: ”centering_method”

@to_scale:
Input type: *boolean.*
Description: Enable the preprocessing method to scale the matrix' **X** observations with their mean value. This is a mandatory operation to do if the dataset is multivariate, i.e., the variables have different units and ranges.
Dictionary entry: ”scale”

@scaling:
Input type: *string.*
Description: Set the scaling method. Available choices are ”auto”, ”vast”, ”range” or ”pareto”. For additional info about the aforementioned scaling criteria: *Parente, Alessandro, and James C. Sutherland. Combustion and flame 160.2 (2013): 340-350.*
Dictionary entry: ”scaling_method”

@sigma:

Input type: *float.*
Description: Set the value of sigma to be used in the formula of the affinity matrix computation.
Dictionary entry: ”sigma”

Methods:

fit()

Description: Start spectral clustering partitioning algorithm to partition the observations of the input matrix.

Returns: idx: the vector containing the cluster assignment, whose shape is $(n_{obs} \times 1)$.

3.3 `classification.py`

3.3.1 class: VQPCA

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size($\mathbf{X}$) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

idx: vector representing the training matrix **X** partitioning, whose dimensions are: $(n_{obs} \times 1)$.

Y: raw test data matrix, uncentered and unscaled. It must be organized with a structure: size($\mathbf{X}$) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

Methods:

fit()

Description: Classify the unobserved matrix by means of the reconstruction error.

Returns: idx: the vector containing the cluster assignment, whose shape is $(n_{obs} \times 1)$.

3.4 `utilities.py`

3.4.1 function: evaluate_clustering_DB

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size(**X**) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

idx: vector representing the training matrix **X** partitioning, whose dimensions are: $[n_{obs} \times 1]$.

Outputs:

DB: Davies-Bouldin index (type: float)

3.4.2 function: evaluate_clustering_PHC

Inputs:

X: raw training data matrix, uncentered and unscaled. It must be organized with a structure: size(**X**) = [n$_{obs}$ $\times n_{var}$], i.e., the matrix' rows must consist of n statistical observations of the p variables.

idx: vector representing the training matrix **X** partitioning, whose dimensions are: $(n_{obs} \times 1)$.

Outputs:

PHC_coeff: Physical Homogeneity Coefficient, averaged for all clusters (type: float)

PHC_dev: standard deviation of the PHC coefficient in the different clusters (type: float)

4 Examples

In this Section, several examples of typical machine-learning tasks for reacting flows applications are shown and commented. The data used for the examples can be found in the **data/reactive_flows/** folder, the file is: *flameD.csv*. They consist of a matrix accounting for 10,613 observations of 37 variables, i.e., temperature and 36 chemical species. Only for the examples regarding the spectral clustering algorithm and the outliers removal are used two different datasets, which can be both found in the **data/dummy_data/** folder. For the spectral clustering example, the file *moon.csv* is used, consisting of 300 observations of 2D non-linear points. For the outlier removal example, instead, the *outlier_data.csv* folder is used, consisting of 2,200 observations of 2 variables, which contain synthetical outliers.

4.1 Dimensionality reduction and Feature Extraction via PCA

One of the simplest tasks to accomplish by means of the OpenMORe libraries is to reduce the data dimensionality via PCA, and consequently perform Feature Extraction.

The first step to use the software is always to import the required modules, which in this case are the model order reduction and utilities modules:

```
import OpenMORe.model_order_reduction as model_order_reduction
from OpenMORe.utilities import *
import os
```

After that, it is necessary to set the path to the data matrix **X** whose dimensionality has to be reduced, and then to call the function to load the matrix. For clarity, the file options' settings and the algorithm's settings will be always written as dictionaries.

```
file_options = {
#it automatically goes to the reactive flow folder, by means of the os library
    "path_to_file"              : os.path.abspath(os.path.join(__file__ ,"
                                            ../../../data/reactive_flow/")),
    "input_file_name"           : "flameD.csv",
}

X = readCSV(file_options["path_to_file"], file_options["input_file_name"])
```

Now, another dictionary must be written with the instructions to be given to the PCA algorithm, i.e., centering and scaling options, number of Principal Components etc:

```
settings ={
    #centering and scaling options
    "center"                    : True,
    "centering_method"          : "mean",
    "scale"                     : True,
    "scaling_method"            : "auto",

    #set the final dimensionality
    "number_of_eigenvectors"    : 7,

    #enable to plot the cumulative explained variance
    "enable_plot_variance"      : True,
    #set the number of the variable whose reconstruction must be plotted
    "variable_to_plot"          : 0,

}
```

The number of PCs is set as 7 because, in this way, it is possible to explain the 95% of the original data variance. After having completed the dictionary settings, it is possible to call the PCA class, giving in input the training matrix **X** and the instructions to the algorithm's properties via the settings dictionary:

```
model = model_order_reduction.PCA(X, settings)
```

Thus, it is now possible to apply the dimensionality reduction by means of the **fit()** method, and to retrieve the reduced matrix of the PCA scores, **Z**, by means of the **get_scores()** method:

```
PCs = model.fit()
Z = model.get_scores()
```

To assess the percentage of explained data variance and the quality of the reconstruction, the method **get_explained()** provides a plot as the one in Figure 1, with the curve of cumulative variance, while the method **plot_parity()** provides a parity plot for the reconstruction of the chosen variable, as shown in Figure 2. In this example, the variable number 0 is chosen to assess the quality of the reconstruction, which corresponds to the temperature.

```
model.get_explained()
model.plot_parity()
```

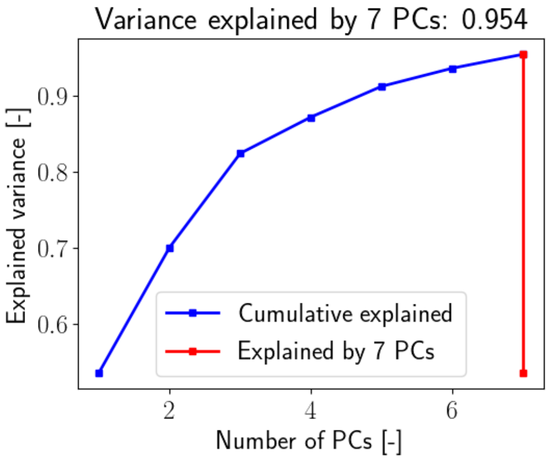

Figure 1: Curve representing the cumulative explained variance.

To perform feature extraction via PCA, the variables' weights on the PCs must be inspected, as well the contours of the first scores (corresponding to the most powerful eigenvectors). To visualize the variables' weights on the PCs it is sufficient to call the method **plot_PCs()**, which returns the

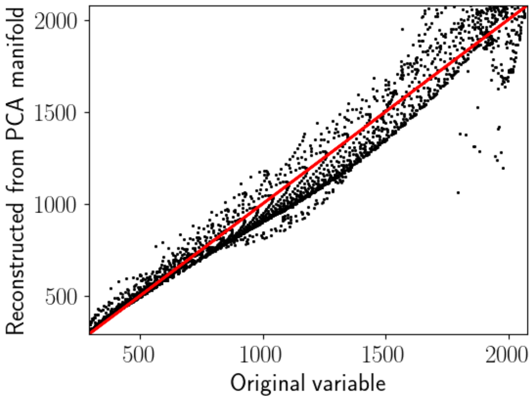

Figure 2: Parity plot for the reconstruction of temperature, using the prescribed number of PCs (selected in an unsupervised fashion) to explain at least the 95% of the original data variance.

bar plot of the selected PC, as in Figure 3:

```
model.plot_PCs()
```

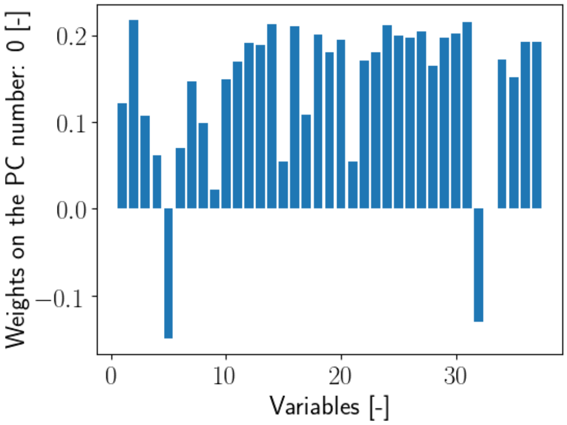

Figure 3: Bar plot representing the variables' weights on the first Principal Component.

To plot the scores contour, instead, it is first necessary to load the mesh from the path: **data/reactive_flow/** as follows:

```
import numpy as np

mesh_options = {
    "path_to_file"                : os.path.abspath(os.path.join(__file__ ,"
                                            ../../../data/reactive_flow/")),
    "mesh_file_name"              : "mesh.csv",
}
```

```
plt.rcParams.update({’font.size’ : 12, ’text.usetex’ : True})
mesh = np.genfromtxt(mesh_options["path_to_file"] + "/" + mesh_options["
                                                    mesh_file_name"], delimiter= ’,’)
```

Then, the matplotlib libraries must be imported to plot the grid, coloured on the basis of the scores, as in Figure 4:

```
import matplotlib.pyplot as plt

fig = plt.figure()
axes = fig.add_axes([0.2,0.15,0.7,0.7], frameon=True)
axes.scatter(mesh[:,0], mesh[:,1], c=Z[:,0],alpha=0.5, cmap=’gnuplot’)
axes.set_xlabel(’X [m]’)
axes.set_ylabel(’Y [m]’)
plt.show()

fig = plt.figure()
axes = fig.add_axes([0.2,0.15,0.7,0.7], frameon=True)
axes.scatter(mesh[:,0], mesh[:,1], c=Z[:,1],alpha=0.5, cmap=’gnuplot’)
axes.set_xlabel(’X [m]’)
axes.set_ylabel(’Y [m]’)
plt.show()
```

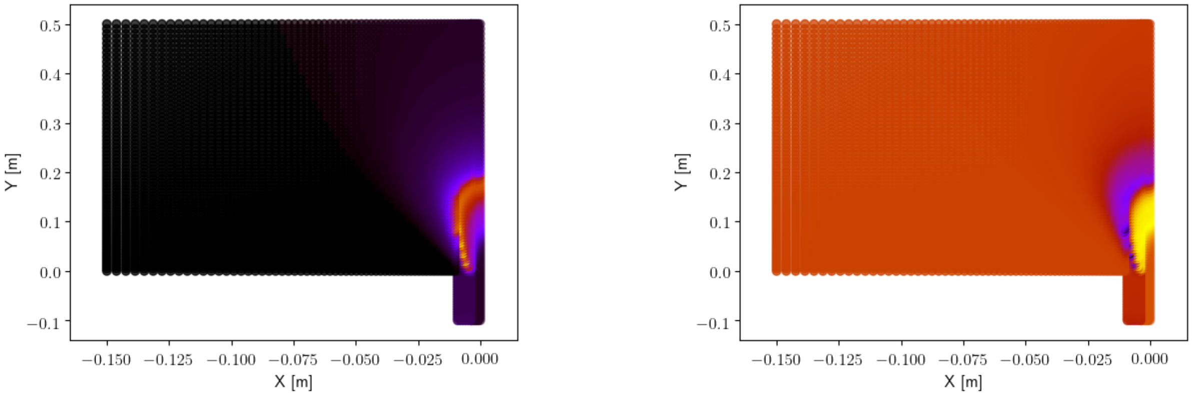

Figure 4: Left: first extracted Principal Component score, plotted on the mesh; Right: second extracted Principal Component score, plotted on the mesh.

4.2 Clustering via Local Principal Component Analysis

The following example shows how to perform clustering by means of the Local Principal Component Analysis algorithm. First of all, it is necessary to import the required modules. In addition to the modules for clustering and utilities, it is also necessary to import numpy and matplotlib if the user wants to plot the clustering solution on the mesh.

```
import OpenMORe.clustering as clustering
from OpenMORe.utilities import *

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import os
```

Now, three different dictionaries must be defined: one to load the data matrix, one to load the mesh and the last one containing the settings to give in input to the LPCA class:

```
file_options = {
    #set the training matrix file options
    "path_to_file"              : os.path.abspath(os.path.join(__file__ ,"
                                            ../../../data/reactive_flow/")),
    "input_file_name"           : "flameD.csv",
}

mesh_options = {
    #set the mesh file options
    "path_to_file"              : "/Users/OpenMORe/data/reactive_flow",
    "mesh_file_name"            : "mesh.csv",

    #eventually enable the clustering solution plot on the mesh
    "plot_on_mesh"              : True,
}

settings = {
    #centering and scaling options
    "center"                    : True,
    "centering_method"          : "mean",
    "scale"                     : True,
    "scaling_method"            : "auto",

    #set the initialization method:
    "initialization_method"     : "uniform",

    #set the number of clusters and PCs in each cluster
    "number_of_clusters"        : 8,
    "number_of_eigenvectors"    : 12,

    #enable additional options:

    #option to write the idx vector containing the labels
    "write_on_txt"              : True,
    #option to use indeces to evaluate the clustering solution
    "evaluate_clustering"       : True,
}
```

To retrieve the clustering solution it is sufficient to give in input to the lpca clustering class the training matrix **X** and the dictionary containing the algorithm's settings, and then to call the **fit()** method:

```
X = readCSV(file_options["path_to_file"], file_options["input_file_name"])

model = clustering.lpca(X, settings)
index = model.fit()
```

After that, if the options `settings["evaluate_clustering]` and `settings["plot_on_mesh"]` are activated, it is possible to evaluate the goodness of the clustering solution by means of the Physical Homogeneity Coefficient and the Davies-Bouldin index implemented in the `utilities.py` module, as well as to visualize the clustering solution plotting the contours on the mesh, as in Figure 5.

```
#eventually evaluate the goodness of the clustering solution
if settings["evaluate_clustering"]:

    #evaluate the clustering solution via PHC
    PHC_coeff, PHC_deviations = evaluate_clustering_PHC(X, index)
    print(PHC_coeff)

    #evaluate the clustering solution by means of the Davies-Bouldin index.
    #the input matrix must be preprocessed first
    X_tilde = center_scale(X, center(X, method=settings["centering_method"]),
                                        scale(X, method=settings["
                                        scaling_method"]))
    DB = evaluate_clustering_DB(X_tilde, index)

    #save the stats in a txt file for a recap
    text_file = open("stats_clustering_solution.txt", "wt")
    DB_index = text_file.write("DB index equal to: {} \n".format(DB))
    PHC_coeff = text_file.write("Average PHC is: {} \n".format(np.mean(
                                        PHC_coeff)))
    text_file.close()


#eventually plot the clustering solution on the mesh
if mesh_options["plot_on_mesh"]:
    matplotlib.rcParams.update({’font.size’ : 12, ’text.usetex’ : True})
    mesh = np.genfromtxt(mesh_options["path_to_file"] + "/" + mesh_options["
                                        mesh_file_name"], delimiter= ’,’)

    fig = plt.figure()
    axes = fig.add_axes([0.2,0.15,0.7,0.7], frameon=True)
    axes.scatter(mesh[:,0], mesh[:,1], c=index,alpha=0.5, cmap=’gnuplot’)
    axes.set_xlabel(’X [m]’)
    axes.set_ylabel(’Y [m]’)
    plt.show()
```

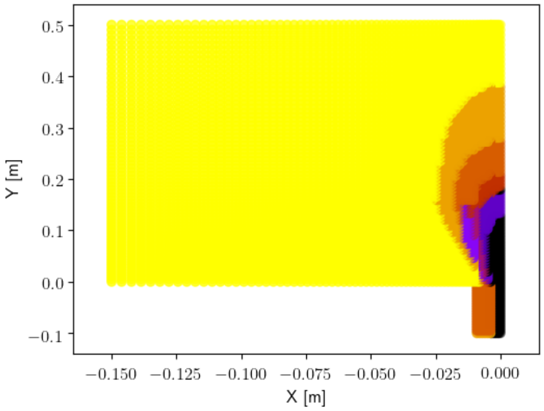

Figure 5: Clustering solution for a RANS simulation of the Sandia Flame D by means of a 36 species mechanism, obtained via Local Principal Component Analysis, using 8 clusters and retaining 12 PCs in each cluster.

4.3 Spectral Clustering

The following example shows how to group similar points by means of the Spectral Clustering algorithm. As usually, it is necessary to import the required modules, i.e., the clustering and utilities modules from OpenMORe, as well as matplotlib to visualize the solution:

```
import OpenMORe.clustering as clustering
from OpenMORe.utilities import *

import matplotlib
import matplotlib.pyplot as plt
import os
```

At this point, it is possible to set the dictionaries with the path to the data and the clustering algorithm's settings. For this example, the non-linear dataset ”moons” will be loaded from the path **data/dummy_data**.

```
file_options = {
    "path_to_file"              : os.path.abspath(os.path.join(__file__ ,"
                                             ../../../data/dummy_data/")),
    "input_file_name"           : "moons.csv",
}


settings = {
    #centering and scaling options
    "center"                    : False,
    "centering_method"          : "mean",
    "scale"                     : False,
    "scaling_method"            : "auto",

    #clustering options: choose the number of clusters
    "number_of_clusters"        : 2,
    "sigma"                     : 0.2,

    #write clustering solution on txt
    "write_on_txt"              : False,
    "evaluate_clustering"       : False,
}
```

In this case, it is not necessary to preprocess the matrix, i.e. center and scale the data, because they are univariate.

```
X = readCSV(file_options["path_to_file"], file_options["input_file_name"])

model = clustering.spectralClustering(X, settings)
idx = model.fit()
```

After the algorithm's convergence, it is possible to plot the scatter with the data coloured with the clustering solution vector (idx), as shown in Figure 6, as follows:

```
matplotlib.rcParams.update({'font.size' : 12, 'text.usetex' : True})

fig = plt.figure()
```

```
axes = fig.add_axes([0.2,0.15,0.7,0.7], frameon=True)

cmap = matplotlib.colors.ListedColormap(['darkred', 'midnightblue'])
sc = axes.scatter(X[:,0], X[:,1], c=idx,alpha=0.5, cmap=cmap, edgecolor ='none'
                                        )
bounds = [0, 1]
axes.set_title("Spectral clustering solution")
axes.set_xlabel('X [-]')
axes.set_ylabel('Y [-]')
cb = plt.colorbar(sc, spacing='uniform', ticks=bounds)
cb.set_ticks(ticks=range(2))
plt.show()
```

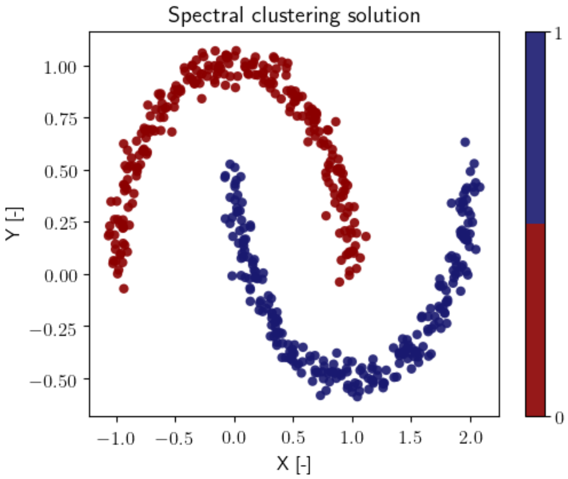

Figure 6: Clustering solution for the moon dataset via Spectral clustering, using $\sigma = 0.2$.

4.4 Outlier identification and removal via PCA

The following example shows how to identify and remove outliers from data via Principal Component Analysis. The dataset which will be used is available in: **OpenMORe/data/dummy_data**, and the associated file name is: *outlier_data.csv*. As usual, the first step is the modules import, as well as the definition of the dictionary with the path and the name of the matrix to be loaded:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import os

import OpenMORe.model_order_reduction as model_order_reduction
from OpenMORe.utilities import *

file_options = {
    "path_to_file"              : os.path.abspath(os.path.join(__file__ ,"
                                            ../../../data/dummy_data/")),
    "input_file_name"           : "outlier_data.csv",
```

As it is possible to see from Figure 7, the loaded data matrix consists of a large gaussian distribution of points centered around zero, and other additional four, which are the synthetic outliers.

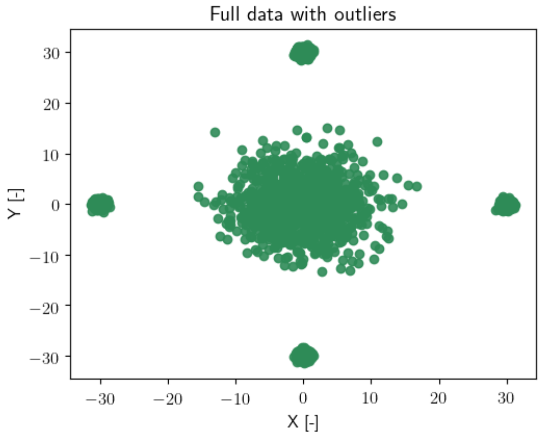

Figure 7: Scatter plot representation of the 2D dataset with synthetic outliers, before their identification and removal via PCA.

At this point it is possible to define the dictionary containing the algorithm's settings:

```
settings ={
    #centering and scaling options
    "center"                    : True,
    "centering"                 : "mean",
    "scale"                     : True,
    "scaling"                   : "auto",

    #set the number of PCs: it can be done automatically, or it can be
    #decided by the user.
    "number_of_PCs"             : 1,

    #set the method for the outlier removal procedure
    "method"                    : "orthogonal",
}
```

The number of PCs is set to one, because the original dimensionality of the data is two, so it is the only choice for the reduced manifold dimensionality. Now, the PCA class is called, the required properties defined in the dictionary are also given in input, and the method to remove the orthogonal outliers is called:

```
model = model_order_reduction.PCA(X, settings)
X_cleaned, bins, idx = model.outlier_removal_orthogonal()
```

After that, two figures are plotted, one containing the outliers identified in red (Figure 8) and one after the outliers removal, i.e., the cleaned matrix (Figure 9):

```
fig = plt.figure()
axes = fig.add_axes([0.2,0.15,0.7,0.7], frameon=True)
axes.scatter(X[:,0], X[:,1], c=’red’,alpha=0.9)
axes.scatter(X_cleaned[:,0], X_cleaned[:,1], c=’darkblue’,alpha=0.9)
axes.set_title("Outliers identification (in red)")
axes.set_xlabel('X [-]')
axes.set_ylabel('Y [-]')
plt.show()


fig = plt.figure()
axes = fig.add_axes([0.2,0.15,0.7,0.7], frameon=True)

axes.scatter(X_cleaned[:,0], X_cleaned[:,1], c='darkblue',alpha=0.9)
axes.set_title("Cleaned data without outliers")
axes.set_xlabel('X [-]')
axes.set_ylabel('Y [-]')
plt.show()
```

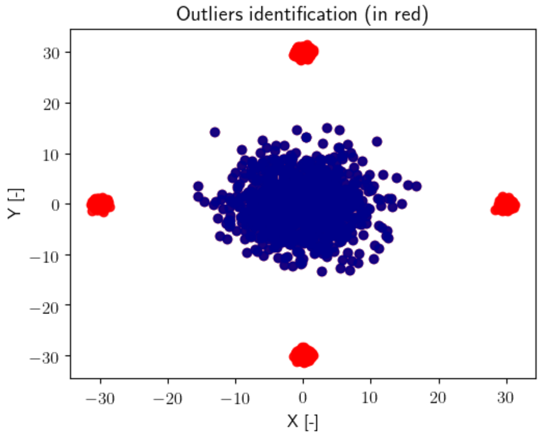

Figure 8: Scatter plot representation of the 2D dataset with synthetic outliers, highlighted in red after their identification via PCA.

4.5 Set automatically the optimal number of clusters via Davies-Bouldin index

The following example shows how to automatically set the optimal number of clusters via Davies-Bouldin (DB) index. Several clustering solutions are computed via LPCA for different values of k, and for each of them the DB is calculated: the optimal k value is chosen as the solution which minimizes the DB value.

As a first step, it is necessary, as usual, to import the modules and to define the file dictionary:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import os

import OpenMORe.clustering as clustering
from OpenMORe.utilities import *
```

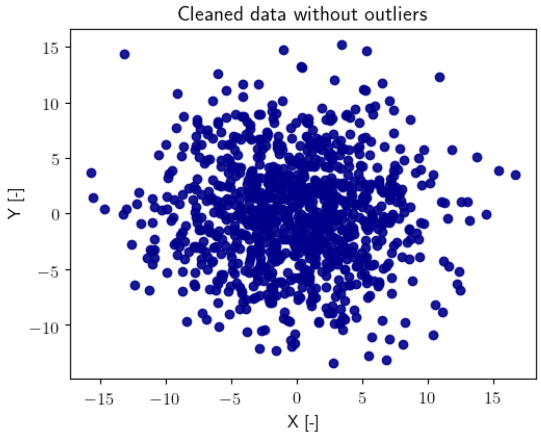

Figure 9: Scatter plot representation of the 2D dataset after the removal of the synthetic outliers via PCA.

```
file_options = {
    "path_to_file"                : os.path.abspath(os.path.join(__file__ ,"
                                            ../../../data/reactive_flow/")),
    "input_file_name"             : "flameD.csv",
}
```

After that, the matrix is loaded and also preprocessed, because the function to compute the DB index needs in input a matrix which is already centered and scaled:

```
X = readCSV(file_options["path_to_file"], file_options["input_file_name"])
X_tilde = center_scale(X, center(X, method=settings["centering_method"]), scale
                                        (X, method=settings["scaling_method"]))
```

Now it is possible to define the dictionary with the settings for the class and its methods:

```
settings = {
    #centering and scaling options
    "center"                      : True,
    "centering_method"            : "mean",
    "scale"                       : True,
    "scaling_method"              : "auto",

    #set the initialization method (random, observations, kmeans, pkcia,
                                            uniform)
    "initialization_method"       : "uniform",

    #set the number of PCs in each cluster
    "number_of_eigenvectors"      : 12,

    #enable additional options:
    "initial_k"                   : 2,
    "final_k"                     : 10,
}
```

All the settings are identical to the LPCA dictionary, with the addition of the two entries ”initial_k” and ”final_k”. These two values are the boundaries of the interval which is chosen to test the clustering solutions. Before to start the clustering, the vector containing the number of clusters to test must be defined, as well as two lists to store the DB score, as well as the clustering solution for a given k:

```
num_of_k = np.linspace(settings["initial_k"], settings["final_k"], settings["
                                          final_k"]-settings["initial_k"]+1)
DB_scores = [None]*len(num_of_k)
idxs = [None]*len(num_of_k)
```

It is now possible to call the class in a for loop: at the end of each loop iteration the number of clusters is updated. Moreover, each time the algorithm converges, the associated solution vector and DB score are stored in the corresponding lists.

```
for ii in range(0,len(num_of_k)):
    model = clustering.lpca(X, settings)
    model.clusters = int(num_of_k[ii])
    index = model.fit()

    idxs[ii] = index
    DB_scores[ii] = evaluate_clustering_DB(X_tilde, index)
```

Right after the for loop, it is possible to save the optimal clustering solution in a text file, as well as to plot the DB scores results, as shown in Figure 10:

```
best_solution_idx = idxs[np.argmin(DB_scores)]
np.savetxt("best_idx.txt", best_solution_idx)


fig = plt.figure()
axes = fig.add_axes([0.2,0.15,0.7,0.7], frameon=True)

axes.plot(num_of_k, DB_scores, c=’darkblue’,alpha=0.9, marker=’o’, linestyle=’
                                          dashed’, linewidth=2, markersize=12)
axes.set_xlabel(’Number of clusters [-]’)
axes.set_ylabel(’Davies-Bouldin index [-]’)
plt.show()
```

As it is possible to observe from Figure 10, the clustering solutions from $k = 2$ to $k = 10$ have been successfully computed, and the value of k which minimizes the DB index, i.e., the optimal number of clusters, is $k = 7$.

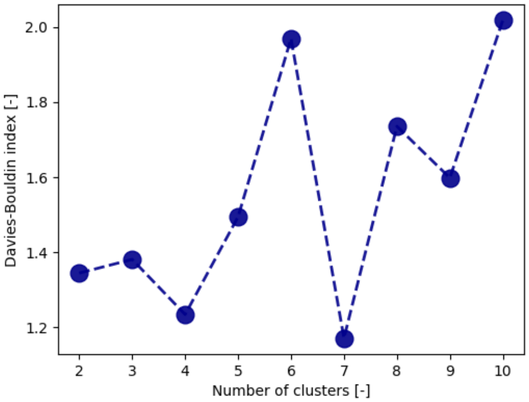

Figure 10: Variation of the Davies-Bouldin (DB) index changing the number of clusters in input. The optimal number of cluster is set as $k = 7$ because it minimizes the DB index' value.