ITMO$_{FS}$

Release 0.3.2

Aug 13, 2020

Getting Started

CHAPTER 1

Install and contribution

1.1 Prerequisites

The feature selection library requires the following dependencies:

- python (>=3.6)
- numpy (>=1.13.3)
- scipy (>=0.19.1)
- scikit-learn (>=0.22)
- imblearn (>=0.0)
- qpsolvers (>=1.0.1)

1.2 Install

ITMO_FS is currently available on the PyPi's repositories and you can install it via *pip*:

```
pip install -U ITMO_FS
```

If you prefer, you can clone it and run the setup.py file. Use the following commands to get a copy from Github and install all dependencies:

```
git clone https://github.com/LastShekel/ITMO_FS.git
cd ITMO_FS
pip install .
```

Or install using pip and GitHub:

```
pip install -U git+https://github.com/LastShekel/ITMO_FS.git
```

1.3 Test and coverage

You want to test the code before to install:

```
$ make test
```

You wish to test the coverage of your version:

```
$ make coverage
```

You can also use *pytest*:

```
$ pytest ITMO_FS -v
```

1.4 Contribute

You can contribute to this code through Pull Request on GitHub. Please, make sure that your code is coming with unit tests to ensure full coverage and continuous integration in the API.

CHAPTER 2

User Guide

2.1 Introduction

2.1.1 API's of feature selectors

Available selectors follow the scikit-learn API using the base estimator and selector mixin:

Transformer The base object, implements a `fit` method to learn from data, either:

```
selector.fit(data, targets)
```

To select features from a data set after learning, each selector implements:

```
data_selected = selector.transform(data)
```

To learn from data and select features from the same data set at once, each selector implements:

```
data_selected = selector.fir_transform(data, targets)
```

To reverse the selection operation, each selector implements:

```
data_reversed = selector.fir_transform(data)
```

Feature selectors accept the same inputs that in scikit-learn:

- `data`: array-like (2-D list, pandas.Dataframe, numpy.array) or sparse matrices;
- `targets`: array-like (1-D list, pandas.Series, numpy.array).

The output will be of the following type:

- **`data_selected`: array-like (2-D list, pandas.Dataframe, numpy.array) or** sparse matrices;
- **`data_reversed`: array-like (2-D list, pandas.Dataframe, numpy.array) or** sparse matrices;

Sparse input

For sparse input the data is **converted to the Compressed Sparse Rows representation** (see `scipy.sparse.csr_matrix`) before being fed to the sampler. To avoid unnecessary memory copies, it is recommended to choose the CSR representation upstream.

2.1.2 Problem statement regarding data sets with redundant features

Feature selection methods can be used to identify and remove unneeded, irrelevant and redundant attributes from data that do not contribute to the accuracy of a predictive model or may in fact decrease the accuracy of the model. Fewer attributes is desirable because it reduces the complexity of the model, and a simpler model is simpler to understand and explain.

Here is one of examples of feature selection improving the classification quality:

```
>>> from sklearn.datasets import make_classification
>>> from sklearn.linear_model import SGDClassifier
>>> from ITMO_FS.embedded import MOS

>>> X, y = make_classification(n_samples=300, n_features=10, random_state=0, n_
↪informative=2)
>>> sel = MOS()
>>> trX = sel.fit_transform(X, y, smote=False)

>>> cl1 = SGDClassifier()
>>> cl1.fit(X, y)
>>> cl1.score(X, y)
0.9033333333333333

>>> cl2 = SGDClassifier()
>>> cl2.fit(trX, y)
>>> cl2.score(trX, y)
0.9433333333333334
```

As expected, the quality of the SVGClassifier's results is impacted by the presence of redundant features in data set. We can see that after using of feature selection the mean accuracy increases from 0.903 to 0.943.

CHAPTER 3

ITMO_FS API

This is the full API documentation of the *ITMO_FS* toolbox.

3.1 ITMO_FS.filters: Filter methods

3.1.1 ITMO_FS.filters.univariate: Univariate filter methods

filters.univariate.VDM([weighted])	Creates Value Difference Metric builder http://aura.abdn.ac.uk/bitstream/handle/2164/10951/payne_ecai_98.pdf?sequence=1 https://www.jair.org/index.php/jair/article/view/10182
filters.univariate.UnivariateFilter(measure)	Basic interface for using univariate measures for feature selection.

ITMO_FS.filters.univariate.VDM

class ITMO_FS.filters.univariate.**VDM**(*weighted=True*)

Creates Value Difference Metric builder http://aura.abdn.ac.uk/bitstream/handle/2164/10951/payne_ecai_98.pdf?sequence=1 https://www.jair.org/index.php/jair/article/view/10182

Parameters **weighted** (*bool*) – If weighted = False, modified version of metric which omits the weights is used

Notes

For more details see papers about Improved Heterogeneous Distance Functions and Implicit Future Selection with the VDM.

Examples

```
>>> x = np.array([[0, 0, 0, 0],
...               [1, 0, 1, 1],
...               [1, 0, 0, 2]])
>>> y = np.array([0,
...               1,
...               1])
>>> vdm = VDM()
>>> vdm.run(x, y)
array([[0.         4.35355339 4.        ]
       [4.5        0.         0.5       ]
       [4.         0.35355339 0.        ]])
```

__init__(*weighted=True*)
Initialize self. See help(type(self)) for accurate signature.

run(*x*, *y*)

Generates metric for the data Complexity: O(n_features * n_samples^3) worst case, should be faster on a real data

x: array-like, shape (n_features, n_samples) Input samples' parameters. Parameters among every class must be sequential integers.

y: array-like, shape (n_samples) Input samples' class labels. Class labels must be sequential integers.

result: numpy.ndarray, shape=(n_samples, n_samples), dtype=np.double with selected version of metrics

feature_scores = {} def run(self, x, y, weighted=True):

ITMO_FS.filters.univariate.UnivariateFilter

class ITMO_FS.filters.univariate.**UnivariateFilter**(*measure*, *cutting_rule=('Best by percentage', 0.2)*)

Basic interface for using univariate measures for feature selection. List of available measures is in ITMO_FS.filters.univariate.measures, also you can provide your own measure but it should suit the argument scheme for measures, i.e. take two arguments x,y and return scores for all the features in dataset x. Same applies to cutting rules.

Parameters

- **measure** (*string or callable*) – A metric name defined in GLOB_MEASURE or a callable with signature measure (sample dataset, labels of dataset samples) which should return a list of metric values for each feature in the dataset.
- **cutting_rule** (*string or callables*) – A cutting rule name defined in GLOB_CR or a callable with signature cutting_rule (features), which should return a list features ranked by some rule.

Examples

```
>>> from sklearn.datasets import make_classification
>>> from ITMO_FS.filters.univariate import select_k_best
>>> from ITMO_FS.filters.univariate import UnivariateFilter
>>> from ITMO_FS.filters.univariate import f_ratio_measure
>>> x, y = make_classification(1000, 100, n_informative = 10, n_redundant = 30, n_
↪repeated = 10, shuffle = False)
>>> ufilter = UnivariateFilter(f_ratio_measure, select_k_best(10))
>>> ufilter.fit(x, y)
>>> print(ufilter.selected_features)
```

__init__ (*measure*, *cutting_rule=('Best by percentage', 0.2)*)
Initialize self. See help(type(self)) for accurate signature.

fit (*X*, *y*, *feature_names=None*, *store_scores=True*)
Fits the filter.

Parameters

- **X** (*array-like, shape (n_features, n_samples)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names
- **store_scores** (*boolean, optional (by default False)*) – In case you want to store the scores of features for future calls to Univariate filter

Returns

Return type None

fit_transform (*X*, *y=None*, *feature_names=None*, *store_scores=False*, ***fit_params*)
Fits the filter and transforms given dataset X.

Parameters

- **X** (*array-like, shape (n_features, n_samples)*) – The training input samples.
- **y** (*array-like, shape (n_samples,), optional*) – The target values.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names
- **store_scores** (*boolean, optional (by default False)*) – In case you want to store the scores of features for future calls to Univariate filter
- ****fit_params** – dictonary of measure parameter if needed.

Returns

Return type X dataset sliced with features selected by the filter

get_scores (*X*, *y*, *feature_names*)
Counts feature scores on given data.

Parameters

- **X** (*array-like, shape (n_features, n_samples)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.

- **feature_names** (*list of strings*) – In case you want to define feature names

Returns **dictionary of format**

Return type key - feature_names, values - feature scores

transform(*X*)

Slices given dataset by previously selected features.

Parameters **X** (*array-like, shape (n_features, n_samples)*) – The training input samples.

Returns

Return type X dataset sliced with features selected by the filter

Measures for univariate filters

filters.univariate.fit_criterion_measure(X, y)	
filters.univariate.f_ratio_measure(X, y)	Calculates Fisher score for features.
filters.univariate.gini_index(X, y)	Gini index is a measure of statistical dispersion.
filters.univariate.su_measure(X, y)	SU is a correlation measure between the features and the class calculated, via formula SU(X,Y) = 2 * I(X|Y) / (H(X) + H(Y))
filters.univariate.spearman_corr(X, y)	Calculates spearman correlation for each feature.
filters.univariate.pearson_corr(X, y)	Calculates pearson correlation for each feature.
filters.univariate.fechner_corr(X, y)	Calculates Sample sign correlation (Fechner correlation) for each feature.
filters.univariate.kendall_corr(X, y)	Calculates Sample sign correlation (Kendall correlation) for each feature.
filters.univariate.reliefF_measure(X, y[, ...])	Counts ReliefF measure for each feature
filters.univariate.chi2_measure(X, y)	Calculates score for the test chi-squared statistic from X.
filters.univariate.information_gain(X, y)	Calculates mutual information for each feature by formula, I(X,Y) = H(X) - H(X|Y)

ITMO_FS.filters.univariate.fit_criterion_measure

ITMO_FS.filters.univariate.**fit_criterion_measure**(*X*, *y*)

ITMO_FS.filters.univariate.f_ratio_measure

ITMO_FS.filters.univariate.**f_ratio_measure**(*X*, *y*)

Calculates Fisher score for features.

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The input samples.
- **y** (*numpy array, shape (n_samples,)*) – The classes for the samples.

Returns

Return type Score for each feature as a numpy array, shape (n_features,)

See also:

https() //papers.nips.cc/paper/2909-laplacian-score-for-feature-selection.pdf

Examples

```
>>> import sklearn.datasets as datasets
>>> from ITMO_FS.filters.univariate import f_ratio_measure
>>> X, y = datasets.make_classification(n_samples=200, n_features=7,␣
↪shuffle=False)
>>> scores = f_ratio_measure(X, y)
>>> print(scores)
```

ITMO_FS.filters.univariate.gini_index

ITMO_FS.filters.univariate.**gini_index**(*X*, *y*)

Gini index is a measure of statistical dispersion. Note: before counting gini index data is normalized with MinMaxScaler

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The input samples.
- **y** (*numpy array, shape (n_samples,)*) – The classes for the samples.

Returns

Return type Score for each feature as a numpy array, shape (n_features,)

See also:

https() //en.wikipedia.org/wiki/Gini_coefficient

Examples

import sklearn.datasets as datasets from ITMO_FS.filters.univariate import gini_index

X, y = datasets.make_classification(n_samples=200, n_features=7, shuffle=False) scores = gini_index(X, y) print(scores)

```
>>> import sklearn.datasets as datasets
>>> from ITMO_FS.filters.univariate import gini_index
>>> X, y = datasets.make_classification(n_samples=200, n_features=7,␣
↪shuffle=False)
>>> scores = gini_index(X, y)
>>> print(scores)
```

ITMO_FS.filters.univariate.su_measure

ITMO_FS.filters.univariate.**su_measure**(*X*, *y*)

SU is a correlation measure between the features and the class calculated, via formula SU(X,Y) = 2 * I(X|Y) / (H(X) + H(Y))

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The input samples.
- **y** (*numpy array, shape (n_samples,)*) – The classes for the samples.

Returns

Return type Score for each feature as a numpy array, shape (n_features,)

See also:

https() //www.matec-conferences.org/articles/matecconf/pdf/2016/05/matecconf_iccma2016_06002.pdf

Examples

```
>>> import sklearn.datasets as datasets
>>> from ITMO_FS.filters.univariate import su_measure
>>> X = np.array([[1, 2, 3, 3, 1],[2, 2, 3, 3, 2], [1, 3, 3, 1, 3],[3, 1, 3, 1,␣
↪4],[4, 4, 3, 1, 5]], dtype = np.integer)
>>> y = np.array([1, 2, 3, 4, 5], dtype=np.integer)
>>> scores = su_measure(X, y)
>>> print(scores)
```

ITMO_FS.filters.univariate.spearman_corr

ITMO_FS.filters.univariate.**spearman_corr**(*X*, *y*)

Calculates spearman correlation for each feature. Spearman's correlation assesses monotonic relationships (whether linear or not). If there are no repeated data values, a perfect Spearman correlation of +1 or 1 occurs when each of the variables is a perfect monotone function of the other.

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The input samples.
- **y** (*numpy array, shape (n_samples,)*) – The classes for the samples.

Returns

Return type Score for each feature as a numpy array, shape (n_features,)

See also:

https() //en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

Examples

```
>>> import sklearn.datasets as datasets
>>> from ITMO_FS.filters.univariate import spearman_corr
>>> X, y = datasets.make_classification(n_samples=200, n_features=7,␣
↪shuffle=False)
>>> scores = spearman_corr(X, y)
>>> print(scores)
```

ITMO_FS.filters.univariate.pearson_corr

ITMO_FS.filters.univariate.**pearson_corr**(*X*, *y*)

Calculates pearson correlation for each feature. Pearson correleation coeficient is a statistic that measures linear correlation between two variables X and Y. It has a value in interval [-1, +1], where 1 is total positive linear correlation, 0 is no linear correlation, and 1 is total negative linear correlation

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The input samples.
- **y** (*numpy array, shape (n_samples,)*) – The classes for the samples.

Returns

Return type Score for each feature as a numpy array, shape (n_features,)

See also:

https() //en.wikipedia.org/wiki/Pearson_correlation_coefficient

Examples

```
>>> import sklearn.datasets as datasets
>>> from ITMO_FS.filters.univariate import pearson_corr
>>> X, y = datasets.make_classification(n_samples=200, n_features=7,␣
↪shuffle=False)
>>> scores = pearson_corr(X, y)
>>> print(scores)
```

ITMO_FS.filters.univariate.fechner_corr

ITMO_FS.filters.univariate.**fechner_corr**(*X*, *y*)

Calculates Sample sign correlation (Fechner correlation) for each feature.

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The input samples.
- **y** (*numpy array, shape (n_samples,)*) – The classes for the samples.

Returns

Return type Score for each feature as a numpy array, shape (n_features,)

Examples

```
>>> import sklearn.datasets as datasets
>>> from ITMO_FS.filters.univariate import fechner_corr
>>> X, y = datasets.make_classification(n_samples=200, n_features=7,␣
↪shuffle=False)
>>> scores = fechner_corr(X, y)
>>> print(scores)
```

ITMO_FS.filters.univariate.kendall_corr

ITMO_FS.filters.univariate.**kendall_corr**(*X*, *y*)

Calculates Sample sign correlation (Kendall correlation) for each feature.

Parameters

- **X** (*numpy array, shape (n_samples, n_features) or (n_samples,)*) – The input samples.
- **y** (*numpy array, shape (n_samples,)*) – The classes for the samples.

Returns

Return type Score for each feature as a numpy array, shape (n_features,)

See also:

https() //en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient

Examples

```
>>> import sklearn.datasets as datasets
>>> from ITMO_FS.filters.univariate import kendall_corr
>>> X, y = datasets.make_classification(n_samples=200, n_features=7,␣
↪shuffle=False)
>>> scores = kendall_corr(X, y)
>>> print(scores)
```

ITMO_FS.filters.univariate.reliefF_measure

ITMO_FS.filters.univariate.**reliefF_measure**(*X*, *y*, *k_neighbors=1*)

Counts ReliefF measure for each feature

Note: Only for complete X Rather than repeating the algorithm m(TODO Ask Nikita about user defined) times, implement it exhaustively (i.e. n times, once for each instance) for relatively small n (up to one thousand).

Calculates spearman correlation for each feature. Spearman's correlation assesses monotonic relationships (whether linear or not). If there are no repeated data values, a perfect Spearman correlation of +1 or 1 occurs when each of the variables is a perfect monotone function of the other.

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The input samples.
- **y** (*numpy array, shape (n_samples,)*) – The classes for the samples.
- **k_neighbors** (*int, optional = 1,*) – The number of neighbors to consider when assigning feature importance scores. More neighbors results in more accurate scores, but takes longer. Selection of k hits and misses is the basic difference to Relief and ensures greater robustness of the algorithm concerning noise.

Returns

Return type Score for each feature as a numpy array, shape (n_features,)

See also:

R.J.(), Journal()

Examples

```
>>> import sklearn.datasets as datasets
>>> from ITMO_FS.filters.univariate import reliefF_measure
>>> X, y = datasets.make_classification(n_samples=200, n_features=7,␣
↪shuffle=False)
>>> scores = reliefF_measure(X, y)
>>> print(scores)
```

ITMO_FS.filters.univariate.chi2_measure

ITMO_FS.filters.univariate.**chi2_measure**(*X*, *y*)

Calculates score for the test chi-squared statistic from X. Chi-squared test is a statistical hypothesis test that is valid to perform when the test statistic is chi-squared distributed under the null hypothesis

Note: Input data must contain only non-negative features such as booleans or frequencies (e.g., term counts in document classification), relative to the classes.

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The input samples.
- **y** (*numpy array, shape (n_samples,)*) – The classes for the samples.

Returns

Return type Score for each feature as a numpy array, shape (n_features,)

See also:

https() //en.wikipedia.org/wiki/Chi-squared_test

Examples

ITMO_FS.filters.univariate.information_gain

ITMO_FS.filters.univariate.**information_gain**(*X*, *y*)

Calculates mutual information for each feature by formula, I(X,Y) = H(X) - H(X|Y)

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The input samples.
- **y** (*numpy array, shape (n_samples,)*) – The classes for the samples.

Returns

Return type Score for each feature as a numpy array, shape (n_features,)

Examples

```
>>> import sklearn.datasets as datasets
>>> from ITMO_FS.filters.univariate import information_gain
>>> X = np.array([[1, 2, 3, 3, 1],[2, 2, 3, 3, 2], [1, 3, 3, 1, 3],[3, 1, 3, 1,␣
↪4],[4, 4, 3, 1, 5]], dtype = np.integer)
>>> y = np.array([1, 2, 3, 4, 5], dtype=np.integer)
```

(continues on next page)

(continued from previous page)

```
>>> scores = information_gain(X, y)
>>> print(scores)
```

Cutting rules for univariate filters

filters.univariate.select_best_by_value(value)
filters.univariate.select_worst_by_value(value)
filters.univariate.select_k_best(k)
filters.univariate.select_k_worst(k)
filters.univariate.select_best_percentage(percent)
filters.univariate.select_worst_percentage(percent)

ITMO_FS.filters.univariate.select_best_by_value

ITMO_FS.filters.univariate.**select_best_by_value**(*value*)

ITMO_FS.filters.univariate.select_worst_by_value

ITMO_FS.filters.univariate.**select_worst_by_value**(*value*)

ITMO_FS.filters.univariate.select_k_best

ITMO_FS.filters.univariate.**select_k_best**(*k*)

ITMO_FS.filters.univariate.select_k_worst

ITMO_FS.filters.univariate.**select_k_worst**(*k*)

ITMO_FS.filters.univariate.select_best_percentage

ITMO_FS.filters.univariate.**select_best_percentage**(*percent*)

ITMO_FS.filters.univariate.select_worst_percentage

ITMO_FS.filters.univariate.**select_worst_percentage**(*percent*)

3.1.2 ITMO_FS.filters.multivariate: Multivariate filter methods

filters.multivariate.DISRWithMassive([...])	Creates DISR (Double Input Symmetric Relevance) feature selection filter based on kASSI criterin for feature selection which aims at maximizing the mutual information avoiding, meanwhile, large multivariate density estimation.
filters.multivariate.FCBFDiscreteFilter()	Creates FCBF (Fast Correlation Based filter) feature selection filter based on mutual information criteria for data with discrete features This filter finds best set of features by searching for a feature, which provides the most information about classification problem on given dataset at each step and then eliminating features which are less relevant than redundant
filters.multivariate.MultivariateFilter(...)	Provides basic functionality for multivariate filters.
filters.multivariate.STIR([n_features_to_keep])	Feature selection using STIR algorithm.
filters.multivariate.TraceRatioFisher(...)	Creates TraceRatio(similarity based) feature selection filter performed in supervised way, i.e fisher version
filters.multivariate.MIMAGA(mim_size, ...)	

ITMO_FS.filters.multivariate.DISRWithMassive

class ITMO_FS.filters.multivariate.**DISRWithMassive**(*expected_size=None*)

Creates DISR (Double Input Symmetric Relevance) feature selection filter based on kASSI criterin for feature selection which aims at maximizing the mutual information avoiding, meanwhile, large multivariate density estimation. Its a kASSI criterion with approximation of the information of a set of variables by counting average information of subset on combination of two features. This formulation thus deals with feature complementarity up to order two by preserving the same computational complexity of the MRMR and CMIM criteria The DISR calculation is done using graph based solution.

Parameters **expected_size** (*int*) – Expected size of subset of features.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import DISRWithMassive
>>> import numpy as np
>>> X = np.array([[1, 2, 3, 3, 1],[2, 2, 3, 3, 2], [1, 3, 3, 1, 3],[3, 1, 3, 1,␣
↪4],[4, 4, 3, 1, 5]], dtype = np.integer)
>>> y = np.array([1, 2, 3, 4, 5], dtype=np.integer)
>>> disr = DISRWithMassive(3)
>>> print(disr.fit_transform(X, y))
```

__init__(*expected_size=None*)

Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y*, *feature_names=None*)

Fits filter

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names

Returns

Return type None

fit_transform(*X*, *y*, *feature_names=None*)

Fits the filter and transforms given dataset X.

Parameters

- **X** (*array-like, shape (n_features, n_samples)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names

Returns

Return type X dataset sliced with features selected by the filter

transform(*X*)

Transform given data by slicing it with selected features.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.

Returns

Return type Transformed 2D numpy array

ITMO_FS.filters.multivariate.FCBFDiscreteFilter

class ITMO_FS.filters.multivariate.**FCBFDiscreteFilter**

Creates FCBF (Fast Correlation Based filter) feature selection filter based on mutual information criteria for data with discrete features This filter finds best set of features by searching for a feature, which provides the most information about classification problem on given dataset at each step and then eliminating features which are less relevant than redundant

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import FCBFDiscreteFilter
>>> import numpy as np
>>> X = np.array([[1, 2, 3, 3, 1],[2, 2, 3, 3, 2], [1, 3, 3, 1, 3],[3, 1, 3, 1,␣
↪4],[4, 4, 3, 1, 5]], dtype = np.integer)
```

(continues on next page)

(continued from previous page)

```
>>> y = np.array([1, 2, 3, 4, 5], dtype=np.integer)
>>> fcbf = FCBFDiscreteFilter()
>>> print(fcbf.fit_transform(X, y))
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y*, *feature_names=None*)
Fits filter

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names

Returns

Return type None

fit_transform(*X*, *y*, *feature_names=None*)
Fits the filter and transforms given dataset X.

Parameters

- **X** (*array-like, shape (n_features, n_samples)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names

Returns

Return type X dataset sliced with features selected by the filter

transform(*X*)
Transform given data by slicing it with selected features.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.

Returns

Return type Transformed 2D numpy array

ITMO_FS.filters.multivariate.MultivariateFilter

class ITMO_FS.filters.multivariate.**MultivariateFilter**(*measure*, *n_features*, *beta=None*, *gamma=None*)

Provides basic functionality for multivariate filters.

Parameters

- **measure** (*string or callable*) – A metric name defined in GLOB_MEASURE or a callable with signature measure(selected_features, free_features, dataset, labels) which should return a list of metric values for each feature in the dataset.

- **n_features** (*int*) – Number of features to select.
- **beta** (*float, optional*) – Initialize only in case you run MIFS or generalizedCriteria metrics.
- **gamma** (*float, optional*) – Initialize only in case you run generalizedCriteria metric.

Examples

```
>>> from ITMO_FS.filters.multivariate import MultivariateFilter
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> model = MultivariateFilter('MRMR', 8)
>>> model.fit(data, target)
>>> print(model.selected_features)
```

__init__ (*measure*, *n_features*, *beta=None*, *gamma=None*)
Initialize self. See help(type(self)) for accurate signature.

fit (*X*, *y*, *feature_names=None*)
Fits the filter.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names

Returns

Return type None

fit_transform (*X*, *y=None*, *feature_names=None*, ***fit_params*)
Fits the filter and transforms given dataset X.

Parameters

- **X** (*array-like, shape (n_features, n_samples)*) – The training input samples.
- **y** (*array-like, shape (n_samples,), optional*) – The target values.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names
- ****fit_params** – dictonary of measure parameter if needed.

Returns

Return type X dataset sliced with features selected by the filter

transform(*X*)
Transform given data by slicing it with selected features.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.

Returns

Return type Transformed 2D numpy array

ITMO_FS.filters.multivariate.STIR

class ITMO_FS.filters.multivariate.**STIR**(*n_features_to_keep=10*)
Feature selection using STIR algorithm.

Algorithm taken from paper:

STatistical Inference Relief (STIR) feature selection (https://academic.oup.com/bioinformatics/article/35/8/1358/5100883).

__init__(*n_features_to_keep=10*)
Sets up STIR to perform feature selection.

distance_matrix(*X*)
Computes the distance matrix.

Before calculating distance we center matrix and normalize it.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – matrix to compute column difference of.

Returns **X_distances** – distance matrix.

Return type array-like, shape (n_samples, n_samples)

find_neighbors(*X*, *y*, *k=1*)
Finds the nearest hit/miss matrices.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – matrix to compute neighbors of.
- **y** (*array-like, shape (n_samples,)*) – vector of binary class status (usually -1/1).
- **k** (*int, optional*) – number of constant nearest hits/misses.

Returns **hitmiss** – hitmiss[1] (hits) and hitmiss[2] (misses). Each list has two columns: index is the first column (instances) in both lists. The second column is hit_index (nearest hits for the first column instance) for list [1] and miss_index (nearest misses) for list [2].

Return type array-like, shape (2,)

fit(*X*, *y*, *feature_names=None*, *k=1*)
Computes the feature importance scores from the training data.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training instances to compute the feature importance scores from.
- **y** (*array-like, shape (n_samples,)*) – Training labels.

- **feature_names** (*list of strings, optional*) – In case you want to define feature names
- **k** (*int, optional*) – number of constant nearest hits/misses.

Returns

Return type None

fit_transform(*X*, *y*, *feature_names=None*, *k=1*)

Fits and transforms data.

Computes the feature importance scores from the training data, then reduces the feature set down to the top 'n_features_to_keep' features.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training instances to compute the feature importance scores from.
- **y** (*array-like, shape (n_samples,)*) – Training labels.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names
- **k** (*int, optional*) – number of constant nearest hits/misses.

Returns

Return type Transformed 2D numpy array

max_diff(*X*)

Computes max difference in each column.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – matrix to compute column difference of.

Returns **diff_vector** – column difference vector.

Return type array-like, shape (n_features)

transform(*X*)

Reduces the feature set down to the top *n_features_to_keep* features.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – Feature matrix to perform feature selection on.

Returns

Return type Transformed 2D numpy array

ITMO_FS.filters.multivariate.TraceRatioFisher

class ITMO_FS.filters.multivariate.**TraceRatioFisher**(*n_selected_features*)

Creates TraceRatio(similarity based) feature selection filter performed in supervised way, i.e fisher version

Parameters **n_selected_features** (*int*) – Amount of features to filter

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import TraceRatioFisher
>>> from sklearn.datasets import make_classification
>>> x, y = make_classification(1000, 100, n_informative = 10,n_redundant = 30, n_
↪repeated = 10, shuffle = False)
>>> tracer = TraceRatioFisher(10)
>>> print(tracer.fit_transform(x, y))
```

__init__ (*n_selected_features*)
Initialize self. See help(type(self)) for accurate signature.

fit (*X*, *y*, *feature_names=None*)
Fits filter

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The training input samples
- **y** (*numpy array, shape (n_samples,)*) – The target values
- **feature_names** (*list of strings, optional*) – In case you want to define feature names

Returns

Return type None

Examples

fit_transform (*X*, *y*, *feature_names=None*)
Fits the filter and transforms given dataset X.

Parameters

- **X** (*array-like, shape (n_features, n_samples)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names

Returns

Return type X dataset sliced with features selected by the filter

transform (*X*)
Transform given data by slicing it with selected features.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.

Returns

Return type Transformed 2D numpy array

ITMO_FS.filters.multivariate.MIMAGA

class ITMO_FS.filters.multivariate.**MIMAGA**(*mim_size*, *pop_size*, *max_iter*, *f_target*, *k1*, *k2*, *k3*, *k4*)

__init__(*mim_size*, *pop_size*, *max_iter*, *f_target*, *k1*, *k2*, *k3*, *k4*)

Parameters

- **mim_size** – desirable number of filtered features after MIM
- **pop_size** – initial population size
- **max_iter** – maximum number of iterations in algorithm
- **f_target** – desirable fitness value
- **k1** – consts to determine crossover probability
- **k2** – consts to determine crossover probability
- **k3** – consts to determine mutation probability
- **k4** – consts to determine mutation probability

mimaga_filter(*genes*, *classes*)

The main function to run algorithm :param genes: initial dataset in format: features are rows, samples are columns :param classes: distribution pf initial dataset :return: filtered with MIMAGA dataset, fitness value

Measures for multivariate filters

filters.multivariate.MIM(selected_features, ...)	Mutual Information Maximization feature scoring criterion.
filters.multivariate.MRMR(selected_features, ...)	Minimum-Redundancy Maximum-Relevance feature scoring criterion.
filters.multivariate.JMI(selected_features, ...)	Joint Mutual Information feature scoring criterion.
filters.multivariate.CIFE(selected_features, ...)	Conditional Infomax Feature Extraction feature scoring criterion.
filters.multivariate.MIFS(selected_features, ...)	Mutual Information Feature Selection feature scoring criterion.
filters.multivariate.CMIM(selected_features, ...)	Conditional Mutual Info Maximisation feature scoring criterion.
filters.multivariate.ICAP(selected_features, ...)	Interaction Capping feature scoring criterion.
filters.multivariate.DCSF(selected_features, ...)	Dynamic change of selected feature with the class scoring criterion.
filters.multivariate.CFR(selected_features, ...)	The criterion of CFR maximizes the correlation and minimizes the redundancy.
filters.multivariate.MRI(selected_features, ...)	Max-Relevance and Max-Independence feature scoring criteria.
filters.multivariate.IWFS(selected_features, ...)	Interaction Weight base feature scoring criteria.

Continued on next page

Table 5 – continued from previous page

filters.multivariate.generalizedCriteria(...)	This feature scoring criteria is a linear combination of all relevance, redundancy, conditional dependency Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

ITMO_FS.filters.multivariate.MIM

ITMO_FS.filters.multivariate.**MIM**(*selected_features*, *free_features*, *X*, *y*)

Mutual Information Maximization feature scoring criterion. This criterion focuses only on increase of relevance. Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import MIM
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(MIM(np.array(selected_features), np.array(other_features), data,␣
↪target))
```

ITMO_FS.filters.multivariate.MRMR

ITMO_FS.filters.multivariate.**MRMR**(*selected_features*, *free_features*, *X*, *y*)

Minimum-Redundancy Maximum-Relevance feature scoring criterion. Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features

- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import MRMR
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(MRMR(np.array(selected_features), np.array(other_features), data,␣
↪target))
```

ITMO_FS.filters.multivariate.JMI

ITMO_FS.filters.multivariate.**JMI**(*selected_features*, *free_features*, *X*, *y*)

Joint Mutual Information feature scoring criterion. Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import JMI
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(JMI(np.array(selected_features), np.array(other_features), data,␣
↪target))
```

ITMO_FS.filters.multivariate.CIFE

ITMO_FS.filters.multivariate.**CIFE**(*selected_features*, *free_features*, *X*, *y*)

Conditional Infomax Feature Extraction feature scoring criterion. Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.

  Notes

- **-----** –
- **more details see `this paper <http**(*For*) –

Examples

```
>>> from ITMO_FS.filters.multivariate import CIFE
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(CIFE(np.array(selected_features), np.array(other_features), data,␣
↪target))
```

ITMO_FS.filters.multivariate.MIFS

ITMO_FS.filters.multivariate.**MIFS**(*selected_features*, *free_features*, *X*, *y*, *beta*)

Mutual Information Feature Selection feature scoring criterion. This criterion includes the I(X;Y) term to ensure feature relevance, but introduces a penalty to enforce low correlations with features already selected in set. Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.
- **beta** (*float,*) – coeficient for redundancy term

  Notes

- **-----** –
- **more details see `this paper <http** (*For*) –

Examples

```
>>> from ITMO_FS.filters.multivariate import MIFS
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(MIFS(np.array(selected_features), np.array(other_features), data,␣
↪target, 0.4))
```

ITMO_FS.filters.multivariate.CMIM

ITMO_FS.filters.multivariate.**CMIM**(*selected_features*, *free_features*, *X*, *y*)

Conditional Mutual Info Maximisation feature scoring criterion. Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.

- **y** (*array-like, shape (n_samples,)*) – The target values.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import CMIM
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(CMIM(np.array(selected_features), np.array(other_features), data,␣
↪target))
```

ITMO_FS.filters.multivariate.ICAP

ITMO_FS.filters.multivariate.**ICAP**(*selected_features*, *free_features*, *X*, *y*)

Interaction Capping feature scoring criterion. Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import ICAP
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
```

(continues on next page)

(continued from previous page)

```
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(ICAP(np.array(selected_features), np.array(other_features), data,␣
↪target))
```

ITMO_FS.filters.multivariate.DCSF

ITMO_FS.filters.multivariate.**DCSF**(*selected_features*, *free_features*, *X*, *y*)

Dynamic change of selected feature with the class scoring criterion. DCSF employs both mutual information and conditional mutual information to find an optimal subset of features. Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import DCSF
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(DCSF(np.array(selected_features), np.array(other_features), data,␣
↪target))
```

ITMO_FS.filters.multivariate.CFR

ITMO_FS.filters.multivariate.**CFR**(*selected_features*, *free_features*, *X*, *y*)

The criterion of CFR maximizes the correlation and minimizes the redundancy. Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import CFR
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(CFR(np.array(selected_features), np.array(other_features), data,␣
↪target))
```

ITMO_FS.filters.multivariate.MRI

ITMO_FS.filters.multivariate.**MRI**(*selected_features*, *free_features*, *X*, *y*)

Max-Relevance and Max-Independence feature scoring criteria. Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import MRI
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(MRI(np.array(selected_features), np.array(other_features), data,␣
↪target))
```

ITMO_FS.filters.multivariate.IWFS

ITMO_FS.filters.multivariate.**IWFS**(*selected_features*, *free_features*, *X*, *y*)

Interaction Weight base feature scoring criteria. IWFS is good at identifyng Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.multivariate import IWFS
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
```

(continues on next page)

(continued from previous page)

```
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(IWFS(np.array(selected_features), np.array(other_features), data,␣
↪target))
```

ITMO_FS.filters.multivariate.generalizedCriteria

ITMO_FS.filters.multivariate.**generalizedCriteria**(*selected_features*, *free_features*, *X*, *y*, *beta*, *gamma*)

This feature scoring criteria is a linear combination of all relevance, redundancy, conditional dependency Given set of already selected features and set of remaining features on dataset X with labels y selects next feature.

Parameters

- **selected_features** (*list of ints,*) – already selected features
- **free_features** (*list of ints*) – free features
- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.
- **beta** (*float,*) – coeficient for redundancy term
- **gamma** (*float,*) – coeficient for conditional dependancy term

Notes

See the original paper[1] for more details.

References

Theoretic Feature Selection." JMLR 2012.

Examples

```
>>> from ITMO_FS.filters.multivariate import CFR
>>> from sklearn.datasets import make_classification
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> est = KBinsDiscretizer(n_bins=10, encode='ordinal')
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> est.fit(data)
>>> data = est.transform(data)
>>> selected_features = [1, 2]
```

(continues on next page)

[1] Brown, Gavin et al. "Conditional Likelihood Maximisation: A Unifying Framework for Information

(continued from previous page)

```
>>> other_features = [i for i in range(0, data.shape[1]) if i not in selected_
↪features]
>>> print(generalizedCriteria(np.array(selected_features), np.array(other_
↪features), data, target, 0.4, 0.3))
```

3.1.3 ITMO_FS.filters.unsupervised: Unsupervised filter methods

filters.unsupervised.TraceRatioLaplacian(...)	Creates TraceRatio(similarity based) feature selection filter performed in unsupervised way, i.e laplacian version

ITMO_FS.filters.unsupervised.TraceRatioLaplacian

class ITMO_FS.filters.unsupervised.**TraceRatioLaplacian**(*n_selected_features*, *k=5*, *t=1*)

Creates TraceRatio(similarity based) feature selection filter performed in unsupervised way, i.e laplacian version

Parameters

- **n_selected_features** (*int*) – Amount of features to filter
- **k** (*int*) – number of neighbours to use for knn
- **t** (*int*) – constant for kernel function calculation
  - Note: in laplacian case only. In fisher it uses label similarity, i.e if both samples belong to same class

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.unsupervised.trace_ratio_laplacian import␣
↪TraceRatioLaplacian
>>> from sklearn.datasets import make_classification
>>> x, y = make_classification(1000, 100, n_informative = 10, n_redundant = 30, n_
↪repeated = 10, shuffle = False)
>>> tracer = TraceRatioLaplacian(10)
>>> print(tracer.run(x, y)[0])
```

__init__(*n_selected_features*, *k=5*, *t=1*)

Initialize self. See help(type(self)) for accurate signature.

run(*X*, *y*)

Fits filter

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The training input samples
- **y** (*numpy array, shape (n_samples,)*) – The target values

Returns **feature_indices** – array of feature indices in X

Return type numpy array

Examples

3.1.4 ITMO_FS.filters.sparse: Sparse filter methods

filters.sparse.MCFS(d[, k, p, scheme, sigma])	Performs the Unsupervised Feature Selection for Multi-Cluster Data algorithm.
filters.sparse.NDFS(p[, c, k, alpha, beta, ...])	Performs the Nonnegative Discriminative Feature Selection algorithm.
filters.sparse.RFS(p[, gamma, ...])	Performs the Robust Feature Selection via Joint L2,1-Norms Minimization algorithm.
filters.sparse.SPEC(p[, k, gamma, sigma, ...])	Performs the Spectral Feature Selection algorithm.
filters.sparse.UDFS(p[, c, k, gamma, l, ...])	Performs the Unsupervised Discriminative Feature Selection algorithm.

ITMO_FS.filters.sparse.MCFS

class ITMO_FS.filters.sparse.**MCFS**(*d*, *k=5*, *p=5*, *scheme='dot'*, *sigma=1*)

Performs the Unsupervised Feature Selection for Multi-Cluster Data algorithm.

Parameters

- **d** (*int*) – Number of features to select.
- **k** (*int, optional*) – Amount of clusters to find.
- **p** (*int, optional*) – Amount of nearest neighbors to use while building the graph.
- **scheme** (*str, either '0-1', 'heat' or 'dot', optional*) – Weighting scheme to use while building the graph.
- **sigma** (*float, optional*) – Parameter for heat weighting scheme. Ignored if scheme is not 'heat'.

Notes

For more details see this paper.

Examples

__init__(*d*, *k=5*, *p=5*, *scheme='dot'*, *sigma=1*)

Initialize self. See help(type(self)) for accurate signature.

feature_ranking(*W*)

Calculate the MCFS score for a feature weight matrix.

Parameters **W** (*array-like, shape (n_features, k)*) – Feature weight matrix.

Returns **indices** – Indices of d selected features.

Return type array-like, shape (d)

run (*X*, *y=None*)
Fits filter

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The training input samples.
- **y** (*numpy array, optional*) – The target values (ignored).

Returns **W** – Feature weight matrix.

Return type array-like, shape (n_features, k)

Examples

from ITMO_FS.filters.sparse import MCFS from sklearn.datasets import make_classification import numpy as np

dataset = make_classification(n_samples=100, n_features=20, n_informative=4, n_redundant=0, shuffle=False) data, target = np.array(dataset[0]), np.array(dataset[1]) model = MCFS(d=5, k=2, scheme='heat') weights = model.run(data, target) print(model.feature_ranking(weights))

ITMO_FS.filters.sparse.NDFS

class ITMO_FS.filters.sparse.**NDFS**(*p*, *c=5*, *k=5*, *alpha=1*, *beta=1*, *gamma=1000000000.0*, *sigma=1*, *max_iterations=1000*, *epsilon=1e-05*)
Performs the Nonnegative Discriminative Feature Selection algorithm.

Parameters

- **p** (*int*) – Number of features to select.
- **c** (*int, optional*) – Amount of clusters to find.
- **k** (*int, optional*) – Amount of nearest neighbors to use while building the graph.
- **alpha** (*float, optional*) – Parameter in the objective function.
- **beta** (*float, optional*) – Regularization parameter in the objective function.
- **gamma** (*float, optional*) – Parameter in the objective function that controls the orthogonality condition.
- **sigma** (*float, optional*) – Parameter for the weighting scheme.
- **max_iterations** (*int, optional*) – Maximum amount of iterations to perform.
- **epsilon** (*positive float, optional*) – Specifies the needed residual between the target functions from consecutive iterations. If the residual is smaller than epsilon, the algorithm is considered to have converged.

See also:

http //www.nlpr.ia.ac.cn/2012papers/gjhy/gh27.pdf

Examples

__init__(*p*, *c=5*, *k=5*, *alpha=1*, *beta=1*, *gamma=1000000000.0*, *sigma=1*, *max_iterations=1000*, *epsilon=1e-05*)
Initialize self. See help(type(self)) for accurate signature.

feature_ranking(*W*)
Calculate the NDFS score for a feature weight matrix.

Parameters **W** (*array-like, shape (n_features, c)*) – Feature weight matrix.

Returns **indices** – Indices of p selected features.

Return type array-like, shape(p)

run(*X*, *y=None*)
Fits filter

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The training input samples.
- **y** (*numpy array, shape (n_samples) or (n_samples, n_classes), optional*) – The target values or their one-hot encoding that are used to compute F. If not present, a k-means clusterization algorithm is used. If present, n_classes should be equal to c.

Returns **W** – Feature weight matrix.

Return type array-like, shape (n_features, c)

Examples

```
>>> from ITMO_FS.filters.sparse import NDFS
>>> from sklearn.datasets import make_classification
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_
↪informative=4, n_redundant=0, shuffle=False)
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> model = NDFS(p=5, c=2)
>>> weights = model.run(data)
>>> print(model.feature_ranking(weights))
```

ITMO_FS.filters.sparse.RFS

class ITMO_FS.filters.sparse.**RFS**(*p*, *gamma=1*, *max_iterations=1000*, *epsilon=1e-05*)
Performs the Robust Feature Selection via Joint L2,1-Norms Minimization algorithm.

Parameters

- **p** (*int*) – Number of features to select.
- **gamma** (*float, optional*) – Regularization parameter.
- **max_iterations** (*int, optional*) – Maximum amount of iterations to perform.
- **epsilon** (*positive float, optional*) – Specifies the needed residual between the target functions from consecutive iterations. If the residual is smaller than epsilon, the algorithm is considered to have converged.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.filters.sparse import RFS
>>> from sklearn.datasets import make_classification
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_informative=4,␣
↪n_redundant=0, shuffle=False)
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> model = RFS(gamma=15, epsilon=1e-12)
>>> print(model.run(data, target))
```

__init__ (*p*, *gamma=1*, *max_iterations=1000*, *epsilon=1e-05*)
Initialize self. See help(type(self)) for accurate signature.

feature_ranking (*W*)
Calculate the RFS score for a feature weight matrix.

Parameters **W** (*array-like, shape (n_features, c)*) – Feature weight matrix.

Returns **indices** – Indices of p selected features.

Return type array-like, shape(p)

run (*X*, *y*)
Fits the algorithm.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples) or (n_samples, n_classes)*) – The target values or their one-hot encoding.

Returns **W** – Feature weight matrix.

Return type array-like, shape (n_features, n_classes)

Examples

ITMO_FS.filters.sparse.SPEC

class ITMO_FS.filters.sparse.**SPEC** (*p*, *k=5*, *gamma=<function SPEC.<lambda>>*, *sigma=0.5*, *phi_type=1*)
Performs the Spectral Feature Selection algorithm.

Parameters

- **p** (*int*) – Number of features to select.
- **k** (*int, optional*) – Amount of clusters to find.
- **gamma** (*callable, optional*) – An "increasing function that penalizes high frequency components". Default is gamma(x) = x^2.
- **sigma** (*float, optional*) – Parameter for the weighting scheme.
- **phi_type** (*int (1, 2 or 3), optional*) – Type of feature ranking function to use.

Notes

For more details see this paper.

Examples

__init__ (*p*, *k=5*, *gamma=<function SPEC.<lambda>>*, *sigma=0.5*, *phi_type=1*)
Initialize self. See help(type(self)) for accurate signature.

feature_ranking (*W*)
Calculate the SPEC score for a feature weight matrix.

Parameters **W** (*array-like, shape (n_features)*) – Feature weight matrix.

Returns **indices** – Indices of p selected features.

Return type array-like, shape(p)

run (*X*, *y=None*)
Fits filter

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The training input samples.
- **y** (*numpy array, optional*) – The target values. If present, label values are used to construct the similarity graph and the amount of classes overrides k.

Returns **W** – Feature weight matrix.

Return type array-like, shape (n_features)

Examples

```
>>> from ITMO_FS.filters.sparse import SPEC
>>> from sklearn.datasets import make_classification
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_
↪informative=4, n_redundant=0, shuffle=False)
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> model = SPEC(p=5, k=2)
>>> weights = model.run(data, target)
>>> print(model.feature_ranking(weights))
```

ITMO_FS.filters.sparse.UDFS

class ITMO_FS.filters.sparse.**UDFS** (*p*, *c=5*, *k=5*, *gamma=1*, *l=1e-06*, *max_iterations=1000*, *epsilon=1e-05*)
Performs the Unsupervised Discriminative Feature Selection algorithm.

Parameters

- **p** (*int*) – Number of features to select.
- **c** (*int, optional*) – Amount of clusters to find.
- **k** (*int, optional*) – Amount of nearest neighbors to use while building the graph.

- **gamma** (*float, optional*) – Regularization term in the target function.
- **l** (*float, optional*) – Parameter that controls the invertibility of the matrix used in computing of B.
- **max_iterations** (*int, optional*) – Maximum amount of iterations to perform.
- **epsilon** (*positive float, optional*) – Specifies the needed residual between the target functions from consecutive iterations. If the residual is smaller than epsilon, the algorithm is considered to have converged.

Notes

For more details see this paper.

Examples

__init__ (*p*, *c=5*, *k=5*, *gamma=1*, *l=1e-06*, *max_iterations=1000*, *epsilon=1e-05*)
Initialize self. See help(type(self)) for accurate signature.

feature_ranking (*W*)
Calculate the UDFS score for a feature weight matrix.

Parameters **W** (*array-like, shape (n_features, c)*) – Feature weight matrix.

Returns **indices** – Indices of p selected features.

Return type array-like, shape(p)

run (*X*, *y=None*)
Fits filter

Parameters

- **X** (*numpy array, shape (n_samples, n_features)*) – The training input samples.
- **y** (*numpy array, optional*) – The target values (ignored).

Returns **W** – Feature weight matrix.

Return type array-like, shape (n_features, c)

Examples

```
>>> from ITMO_FS.filters.sparse import UDFS
>>> from sklearn.datasets import make_classification
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_
↪informative=4, n_redundant=0, shuffle=False)
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> model = UDFS(p=5, c=2)
>>> weights = model.run(data)
>>> print(model.feature_ranking(weights))
```

3.2 ITMO_FS.ensembles: Ensemble methods

3.2.1 ITMO_FS.ensembles.measure_based: Measure based ensemble methods

ensembles.measure_based.WeightBased(filters)	

ITMO_FS.ensembles.measure_based.WeightBased

class ITMO_FS.ensembles.measure_based.**WeightBased**(*filters*)

__init__(*filters*)
TODO comments :param filters:

fit(*X*, *y*, *feature_names=None*)
TODO comments :param X: :param y: :param feature_names: :return:

fit_transform(*X*, *y=None*, ***fit_params*)
TODO comments :param X: :param y: :param fit_params: :return:

transform(*x*, *cutting_rule*, *fusion_function=<function weight_fusion>*, *weights=None*)
Transfrom dataset :param x: :param cutting_rule: :param fusion_function: :param weights: :return:

3.2.2 ITMO_FS.ensembles.model_based: Model based ensemble methods

ensembles.model_based.BestSum(models, ...)	

ITMO_FS.ensembles.model_based.BestSum

class ITMO_FS.ensembles.model_based.**BestSum**(*models*, *cutting_rule*)

__init__(*models*, *cutting_rule*)
Initialize self. See help(type(self)) for accurate signature.

3.2.3 ITMO_FS.ensembles.ranking_based: Ranking based ensemble methods

ensembles.ranking_based.Mixed(filters)	Performs feature selection based on several filters, selecting features this way: Get ranks from every filter from input.

ITMO_FS.ensembles.ranking_based.Mixed

class ITMO_FS.ensembles.ranking_based.**Mixed**(*filters*)
Performs feature selection based on several filters, selecting features this way: Get ranks from every filter from input. Then loops through, on every iteration=i selects features on i position on every filter then shuffles them, then adds to result list without duplication, continues until specified number of features

Parameters **filters** (*list of filter functions*) –

Examples

```
>>> from ITMO_FS.filters.univariate.measures import spearman_corr,pearson_corr
>>> from ITMO_FS.ensembles.ranking_based.Mixed import Mixed
>>> from sklearn.datasets import make_classification
>>> x, y = make_classification(1000, 50, n_informative = 5, n_redundant = 3, n_
↪repeated = 2, shuffle = True)
>>> mixed = Mixed([spearman_corr, pearson_corr])
>>> mixed.fit(x, y)
>>> print(mixed.transform(x, 20))
```

__init__ (*filters*)
Initialize self. See help(type(self)) for accurate signature.

3.3 ITMO_FS.embedded: Embedded methods

embedded.MOS([model, loss, seed])	Performs Minimizing Overlapping Selection under SMOTE (MOSS) or under No-Sampling (MOSNS) algorithm.

3.3.1 ITMO_FS.embedded.MOS

class ITMO_FS.embedded.**MOS**(*model=<class 'sklearn.linear_model._stochastic_gradient.SGDClassifier'>*, *loss='log'*, *seed=42*)
Performs Minimizing Overlapping Selection under SMOTE (MOSS) or under No-Sampling (MOSNS) algorithm.

Parameters

- **model** (*constructor*) – The constructor of the model that will be used. Currently only SGDClassifier should be passed, other models would not work.
- **loss** (*str, 'log' or 'hinge'*) – Loss function to use in the algorithm. 'log' gives a logistic regression, while 'hinge' gives a support vector machine.
- **seed** (*int*) – Seed for python random.

Notes

For more details see this paper.

Examples

```
>>> from ITMO_FS.embedded import MOS
>>> import numpy as np
>>> from sklearn.datasets import make_classification
>>> dataset = make_classification(n_samples=100, n_features=20)
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> for i in range(50):  # create imbalance between classes
...     target[i] = 0
>>> print(MOS().fit_transform(data, target))
```

__init__(*model=<class 'sklearn.linear_model._stochastic_gradient.SGDClassifier'>*, *loss='log'*, *seed=42*)
Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y*, *l1_ratio=0.5*, *threshold=0.001*, *epochs=1000*, *alphas=array([0.0002*, *0.0004*, *0.0006*, *0.0008*, *0.001*, *0.0012*, *0.0014*, *0.0016*, *0.0018*, *0.002*, *0.0022*, *0.0024*, *0.0026*, *0.0028*, *0.003*, *0.0032*, *0.0034*, *0.0036*, *0.0038*, *0.004*, *0.0042*, *0.0044*, *0.0046*, *0.0048*, *0.005*, *0.0052*, *0.0054*, *0.0056*, *0.0058*, *0.006*, *0.0062*, *0.0064*, *0.0066*, *0.0068*, *0.007*, *0.0072*, *0.0074*, *0.0076*, *0.0078*, *0.008*, *0.0082*, *0.0084*, *0.0086*, *0.0088*, *0.009*, *0.0092*, *0.0094*, *0.0096*, *0.0098*, *0.01*, *0.0102*, *0.0104*, *0.0106*, *0.0108*, *0.011*, *0.0112*, *0.0114*, *0.0116*, *0.0118*, *0.012*, *0.0122*, *0.0124*, *0.0126*, *0.0128*, *0.013*, *0.0132*, *0.0134*, *0.0136*, *0.0138*, *0.014*, *0.0142*, *0.0144*, *0.0146*, *0.0148*, *0.015*, *0.0152*, *0.0154*, *0.0156*, *0.0158*, *0.016*, *0.0162*, *0.0164*, *0.0166*, *0.0168*, *0.017*, *0.0172*, *0.0174*, *0.0176*, *0.0178*, *0.018*, *0.0182*, *0.0184*, *0.0186*, *0.0188*, *0.019*, *0.0192*, *0.0194*, *0.0196*, *0.0198])*, *sampling=True*, *feature_names=None*)
Runs the MOS algorithm on the specified dataset.

Parameters

- **X** (*array-like, shape (n_samples,n_features)*) – The input samples.
- **y** (*array-like, shape (n_samples)*) – The classes for the samples.
- **l1_ratio** (*float, optional*) – The value used to balance the L1 and L2 penalties in elastic-net.
- **threshold** (*float, optional*) – The threshold value for feature dropout. Instead of comparing them to zero, they are normalized and values with absolute value lower than the threshold are dropped out.
- **epochs** (*int, optional*) – The number of epochs to perform in the algorithm.
- **alphas** (*array-like, shape (n_alphas), optional*) – The range of lambdas that should form the regularization path.
- **sampling** (*bool, optional*) – Bool value that control whether MOSS (True) or MOSNS (False) should be executed.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names

Returns

Return type None

fit_transform(*X*, *y*, *l1_ratio=0.5*, *threshold=0.001*, *epochs=1000*, *alphas=array([0.0002*, *0.0004*, *0.0006*, *0.0008*, *0.001*, *0.0012*, *0.0014*, *0.0016*, *0.0018*, *0.002*, *0.0022*, *0.0024*, *0.0026*, *0.0028*, *0.003*, *0.0032*, *0.0034*, *0.0036*, *0.0038*, *0.004*, *0.0042*, *0.0044*, *0.0046*, *0.0048*, *0.005*, *0.0052*, *0.0054*, *0.0056*, *0.0058*, *0.006*, *0.0062*, *0.0064*, *0.0066*, *0.0068*, *0.007*, *0.0072*, *0.0074*, *0.0076*, *0.0078*, *0.008*, *0.0082*, *0.0084*, *0.0086*, *0.0088*, *0.009*, *0.0092*, *0.0094*, *0.0096*, *0.0098*, *0.01*, *0.0102*, *0.0104*, *0.0106*, *0.0108*, *0.011*, *0.0112*, *0.0114*, *0.0116*, *0.0118*, *0.012*, *0.0122*, *0.0124*, *0.0126*, *0.0128*, *0.013*, *0.0132*, *0.0134*, *0.0136*, *0.0138*, *0.014*, *0.0142*, *0.0144*, *0.0146*, *0.0148*, *0.015*, *0.0152*, *0.0154*, *0.0156*, *0.0158*, *0.016*, *0.0162*, *0.0164*, *0.0166*, *0.0168*, *0.017*, *0.0172*, *0.0174*, *0.0176*, *0.0178*, *0.018*, *0.0182*, *0.0184*, *0.0186*, *0.0188*, *0.019*, *0.0192*, *0.0194*, *0.0196*, *0.0198])*, *sampling=True*, *feature_names=None*)
Fits the algorithm and transforms given dataset X.

Parameters

- **X** (*array-like, shape (n_features, n_samples)*) – The training input samples.

- **y** (*array-like, shape (n_samples,)*) – The target values.
- **l1_ratio** (*float, optional*) – The value used to balance the L1 and L2 penalties in elastic-net.
- **threshold** (*float, optional*) – The threshold value for feature dropout. Instead of comparing them to zero, they are normalized and values with absolute value lower than the threshold are dropped out.
- **epochs** (*int, optional*) – The number of epochs to perform in gradient descent.
- **alphas** (*array-like, shape (n_alphas), optional*) – The range of lambdas that should form the regularization path.
- **sampling** (*bool, optional*) – Bool value that control whether MOSS (True) or MOSNS (False) should be executed.
- **feature_names** (*list of strings, optional*) – In case you want to define feature names

Returns

Return type X dataset sliced with features selected by the algorithm

transform(*X*)

Transform given data by slicing it with selected features.

Parameters **X** (*array-like, shape (n_samples, n_features)*) – The training input samples.

Returns

Return type Transformed 2D numpy array

3.4 ITMO_FS.hybrid: Hybrid methods

hybrid.FilterWrapperHybrid(filter_, wrapper)	
hybrid.Melif(filter_ensemble[, scorer, verbose])	

3.4.1 ITMO_FS.hybrid.FilterWrapperHybrid

class ITMO_FS.hybrid.**FilterWrapperHybrid**(*filter_*, *wrapper*)

__init__(*filter_*, *wrapper*)

Initialize self. See help(type(self)) for accurate signature.

3.4.2 ITMO_FS.hybrid.Melif

class ITMO_FS.hybrid.**Melif**(*filter_ensemble*, *scorer=None*, *verbose=False*)

__init__(*filter_ensemble*, *scorer=None*, *verbose=False*)

Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y*, *estimator*, *cutting_rule*, *test_size=0.3*, *delta=0.5*, *feature_names=None*, *points=None*)

Parameters

- **X** –
- **y** –
- **estimator** –
- **cutting_rule** –
- **test_size** –
- **delta** –
- **feature_names** –
- **points** –

Returns

3.5 ITMO_FS.wrappers: Wrapper methods

3.5.1 ITMO_FS.wrappers.deterministic: Deterministic wrapper methods

wrappers.deterministic.AddDelWrapper(...[, ...])	Creates add-del feature wrapper
wrappers.deterministic.BackwardSelection(...)	Backward Selection removes one feature at a time until the number of features to be removed is reached.
wrappers.deterministic.RecursiveElimination(...)	Performs a recursive feature elimination until the required number of features is reached.
wrappers.deterministic.SequentialForwardSelection(...)	Sequentially Adds Features that Maximises the Classifying function when combined with the features already used TODO add theory about this method

ITMO_FS.wrappers.deterministic.AddDelWrapper

class ITMO_FS.wrappers.deterministic.**AddDelWrapper**(*estimator*, *score*, *maximize=True*, *seed=42*)

Creates add-del feature wrapper

Parameters

- **estimator** (*object*) – A supervised learning estimator with a fit method
- **score** (*boolean*) – A callable function which will be used to estimate score
- **score** – maximize = True if bigger values are better for score function
- **seed** (*int*) – Seed for python random
- **best_score** (*float*) – The best score of given metric on the feature combination after add-del procedure

See also:

Lecture, p.13

Examples

```
>>> from sklearn.metrics import accuracy_score
>>> from sklearn import datasets,linear_model
>>> data = datasets.make_classification(n_samples=1000, n_features=20)
>>> X = np.array(data[0])
>>> y = np.array(data[1])
>>> lg = linear_model.LogisticRegression(solver='lbfgs')
>>> add_del = AddDelWrapper(lg, accuracy_score)
>>> add_del.fit(X, y)
```

```
>>> from sklearn.metrics import mean_absolute_error
>>> boston = datasets.load_boston()
>>> X = boston['data']
>>> y = boston['target']
>>> lasso = linear_model.Lasso()
>>> add_del = AddDelWrapper(lasso, mean_absolute_error, maximize=False)
>>> add_del.fit(X, y)
```

__init__ (*estimator*, *score*, *maximize=True*, *seed=42*)
Initialize self. See help(type(self)) for accurate signature.

fit (*X*, *y*, *cv=3*, *silent=True*)
Fits wrapper.

Parameters

- **X** (*numpy array or pandas DataFrame, shape (n_samples, n_features)*) – The training input samples.
- **y** (*numpy array of pandas Series, shape (n_samples,)*) – The target values.
- **cv=3** (*int*) – Number of splits in cross-validation
- **silent=True** (*boolean*) – If silent=False then prints all the scores during add-del procedure
- **Returns** –
- **----------** –
- **features** (*list*) – List of feature after add-del procedure

Examples

Parameters

- **silent** –
- **y** –
- **X** –
- **cv** –

ITMO_FS.wrappers.deterministic.BackwardSelection

class ITMO_FS.wrappers.deterministic.**BackwardSelection**(*estimator*, *n_features*, *measure*)

Backward Selection removes one feature at a time until the number of features to be removed is reached. On each step, the best n-1 features out of n are chosen (according to some estimator metric) and the last one is removed.

Parameters

- **estimator** (*object*) – A supervised learning estimator with a fit method.
- **n_features** (*int*) – Number of features to be removed.
- **measure** (*string or callable*) – A standard estimator metric (e.g. 'f1' or 'roc_auc') or a callable object / function with signature measure(estimator, X, y) which should return only a single value.

Examples

__init__(*estimator*, *n_features*, *measure*)

Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y*, *cv=3*)

Fits wrapper.

Parameters

- **X** (*array-like, shape (n_samples,n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values.
- **cv** (*int*) – Number of folds in cross-validation.

Returns

Return type None

Examples

```
>>> from ITMO_FS.wrappers import BackwardSelection
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.datasets import make_classification
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_
↪informative=4, n_redundant=0, shuffle=False)
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> model = BackwardSelection(LogisticRegression(), 15, 'f1_macro')
>>> model.fit(data, target)
>>> print(model.selected_features)
```

ITMO_FS.wrappers.deterministic.RecursiveElimination

class ITMO_FS.wrappers.deterministic.**RecursiveElimination**(*estimator*, *n_features*)

Performs a recursive feature elimination until the required number of features is reached.

Parameters

- **estimator** (*object*) – A supervised learning estimator with a fit method that provides information about feature importance either through a **coef_** attribute or through a **feature_importances_** attribute.
- **n_features** (*int*) – Number of features to leave.

Examples

__init__ (*estimator*, *n_features*)
Initialize self. See help(type(self)) for accurate signature.

fit (*X*, *y*)
Fits wrapper.

Parameters

- **X** (*array-like, shape (n_samples,n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – the target values.

Returns

Return type None

See also:

Guyon, I., Weston, J., Barnhill, S., & Vapnik, V., "Gene selection for cancer classification using support vector machines", Mach. Learn., 46(1-3), 389–422, 2002. https://link.springer.com/article/10.1023/A:1012487302797

Examples

```
>>> from sklearn.datasets import make_classification
>>> from ITMO_FS.wrappers import RecursiveElimination
>>> from sklearn.svm import SVC
>>> import numpy as np
>>> dataset = make_classification(n_samples=1000, n_features=20)
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> model = SVC(kernel='linear')
>>> rfe = RecursiveElimination(model, 5)
>>> rfe.fit(data, target)
>>> print("Resulting features: ", rfe.__features__)
```

ITMO_FS.wrappers.deterministic.SequentialForwardSelection

class ITMO_FS.wrappers.deterministic.**SequentialForwardSelection** (*estimator*, *n_features*, *measure*)
Sequentially Adds Features that Maximises the Classifying function when combined with the features already used TODO add theory about this method

Parameters

- **estimator** (*object*) – A supervised learning estimator with a fit method that provides information about feature importance either through a **coef_** attribute or through a **feature_importances_** attribute.

- **n_features** (*int*) – Number of features to select.
- **measure** (*string or callable*) – A standard estimator metric (e.g. 'f1' or 'roc_auc') or a callable object / function with signature measure(estimator, X, y) which should return only a single value.

Examples

__init__(*estimator*, *n_features*, *measure*)
Initialize self. See help(type(self)) for accurate signature.

fit(*X*, *y*, *cv=3*)
Fits wrapper.

Parameters

- **X** (*array-like, shape (n_features,n_samples)*) – The training input samples.
- **y** (*array-like, shape (n_features,n_samples)*) – The target values.
- **cv** (*int*) – Number of folds in cross-validation.

Returns

Return type None

Examples

```
>>> from ITMO_FS.wrappers import SequentialForwardSelection
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.datasets import make_classification
>>> import numpy as np
>>> dataset = make_classification(n_samples=100, n_features=20, n_
↪informative=4, n_redundant=0, shuffle=False)
>>> data, target = np.array(dataset[0]), np.array(dataset[1])
>>> model = SequentialForwardSelection(LogisticRegression(), 5, 'f1_macro')
>>> model.fit(data, target)
>>> print(model.selected_features)
```

Deterministic wrapper function

wrappers.deterministic.qpfs_wrapper(X, y, alpha)	Performs Quadratic Programming Feature Selection algorithm.

ITMO_FS.wrappers.deterministic.qpfs_wrapper

ITMO_FS.wrappers.deterministic.**qpfs_wrapper**(*X*, *y*, *alpha*, *r=None*, *sigma=None*, *solv='quadprog'*, *fn=<function pearson_corr>*)
Performs Quadratic Programming Feature Selection algorithm. Note that this realization requires labels to start from 1 and be numberical. This is function for wrapper based on qpfs so alpha parameter must be specified, in case you don't know alpha parameter it is suggested to use qpfs_filter

Parameters

- **X** (*array-like, shape (n_samples,n_features)*) – The input samples.
- **y** (*array-like, shape (n_samples)*) – The classes for the samples.
- **alpha** (*double value*) – That represents balance between relevance and redundancy of features.
- **r** (*int, optional*) – The number of samples to be used in Nystrom optimization.
- **sigma** (*double, optional*) – The threshold for eigenvalues to be used in solving QP optimization.
- **solv** (*string, optional*) – The name of qp solver according to qpsolvers(https://pypi.org/project/qpsolvers/) naming. Note quadprog is used by default.
- **fn** (*function(array, array), optional*) – The function to count correlation, for example pearson correlation or mutual information. Note pearson_corr from ITMO_FS measures is used by default.

Returns **array-like, shape (n_features)**

Return type the ranks of features in dataset, with rank increase, feature relevance increases and redundancy decreases.

See also:

http() //www.jmlr.org/papers/volume11/rodriguez-lujan10a/rodriguez-lujan10a.pdf

Examples

```
>>> import numpy as np
>>> x = np.array([[3, 3, 3, 2, 2], [3, 3, 1, 2, 3], [1, 3, 5, 1, 1], [3, 1, 4, 3,␣
↪1], [3, 1, 2, 3, 1]])
>>> y = np.array([1, 3, 2, 1, 2])
>>> alpha = 0.5
>>> ranks = qpfs_wrapper(x, y, alpha)
>>> print(ranks)
```

3.5.2 ITMO_FS.wrappers.randomized: Randomized wrapper methods

wrappers.randomized.HillClimbingWrapper(...)	
wrappers.randomized.SimulatedAnnealing(...)	Performs feature selection using simulated annealing
wrappers.randomized.TPhMGWO([wolfNumber, ...])	Performs Grey Wolf optimization with Two-Phase Mutation

ITMO_FS.wrappers.randomized.HillClimbingWrapper

class ITMO_FS.wrappers.randomized.**HillClimbingWrapper**(*estimator*, *scorer*)

__init__(*estimator*, *scorer*)
Initialize self. See help(type(self)) for accurate signature.

ITMO_FS.wrappers.randomized.SimulatedAnnealing

class ITMO_FS.wrappers.randomized.**SimulatedAnnealing**(*classifier*, *score*, *seed=1*, *iteration_number=100*, *c=1*, *init_number_of_features=None*)

Performs feature selection using simulated annealing

Parameters

- **seed** (*integer*) – Random seed used to initialize np.random.seed()
- **iteration_number** (*integer*) – number of iterations of algorithm
- **classifier** (*Classifier instance*) – Classifier used for training and testing on provided datasets.
  - Note that algorithm implementation assumes that classifier has fit, predict methods. Default algorithm uses sklearn.neighbors.KNeighborsClassifier
- **c** (*integer*) – constant c is used t o control the rate of feature perturbation
- **init_number_of_features** (*float*) – number of features to initialize start features subset, Note: by default (5-10) percents of number of features is used

Notes

For more details see this paper.

Examples

```
>>> from sklearn.datasets import make_classification
>>> from sklearn.model_selection import KFold
>>> from ITMO_FS.wrappers.randomized import SimulatedAnnealing
>>> x, y = make_classification(1000, 100, n_informative = 10, n_redundant = 30, n_
↪repeated = 10, shuffle = False)
>>> kf = KFold(n_splits=2)
>>> sa = SimulatedAnnealing()
>>> for train_index, test_index in kf.split(x):
...     sa.fit(x[train_index], y[train_index], x[test_index], y[test_index])
...     print(sa.selected_features)
```

__init__(*classifier*, *score*, *seed=1*, *iteration_number=100*, *c=1*, *init_number_of_features=None*)
Initialize self. See help(type(self)) for accurate signature.

fit(*train_x*, *train_y*, *test_x*, *test_y*)
Runs the Simulated Annealing algorithm on the specified dataset and fits the classifier.

Parameters

- **train_x** (*array-like, shape (n_samples, n_features)*) – The input training samples.
- **train_y** (*array-like, shape (n_samples)*) – The classes for training samples.
- **test_x** (*array-like, shape (n_samples, n_features)*) – The input testing samples.
- **test_y** (*array-like, shape (n_samples)*) – The classes for testing samples.

Returns

Return type None

predict (*test_x*)
Predicts labels on test dataset

Parameters **test_x** (*array-like, shape (n_samples, n_features)*) – The input testing samples.

Returns **array-like, shape (n_samples,n_selected_features)**

Return type array of feature numbers

ITMO_FS.wrappers.randomized.TPhMGWO

class ITMO_FS.wrappers.randomized.**TPhMGWO**(*wolfNumber=10, seed=1, alpha=0.01, classifier=KNeighborsClassifier(n_neighbors=10), foldNumber=5, iteration_number=30, Mp=0.5, errorRate=<function mean_squared_error>*)
Performs Grey Wolf optimization with Two-Phase Mutation

Parameters

- **wolfNumber** (*integer*) – Number of search agents used to find solution for features selection problem
- **seed** (*integer*) – Random seed used to initialize np.random.seed()
- **alpha** (*float*) – weight of importance of classification accuracy Note alpha is used in equation that counts fitness as fitness = alpha * score + beta * |selected_features| / |features| where alpha = 1 - beta
- **classifier** (*classifier used for training and testing on provided dataset*) – Note that algorithm implementation assumes that classifier has fit, predict methods Default algorithm uses sklearn.neighbors.KNeighborsClassifier
- **foldNumber** (*integer*) – fold number to train and test classifier
- **iteration_number** (*integer*) – number of iterations of algorithm
- **Mp** (*float*) – probability of mutation

Notes

For more details see this paper.

Examples

```
>>> import numpy as np
>>> from ITMO_FS.wrappers.randomized import TPhMGWO
>>> from sklearn.datasets import make_classification
>>> tphmgwo = TPhMGWO()
>>> x, y = make_classification(500, 50, n_informative = 10, n_redundant = 30, n_
↪repeated = 10, shuffle = True)
>>> result = tphmgwo.run(x, y)
>>> print(np.where(result == 1))
```

__init__(*wolfNumber=10*, *seed=1*, *alpha=0.01*, *classifier=KNeighborsClassifier(n_neighbors=10)*, *foldNumber=5*, *iteration_number=30*, *Mp=0.5*, *errorRate=<function mean_squared_error>*)
Initialize self. See help(type(self)) for accurate signature.

exception ClassifierMethodsException

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

run(*X*, *y*)
Runs the TPhGWO algorithm on the specified dataset.

Parameters

- **X** (*array-like, shape (n_samples,n_features)*) – The input samples.
- **y** (*array-like, shape (n_samples)*) – The classes for the samples.

Returns **array-like, shape (n_samples,n_selected_features)**

Return type 0-1 array where 1 means feature is selected and 0 not

CHAPTER 4

Getting started

Information to install, test, and contribute to the package.

CHAPTER 5

User Guide

User guide of ITMO_FS

CHAPTER 6

API

The main documentation. This contains an in-depth description of all algorithms and how to apply them.

CHAPTER 7

API Documentation

The exact API of all functions and classes, as given in the doctring. The API documents expected types and allowed features for all functions, and all parameters available for the algorithms.

Python Module Index

e

h

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

M