DiviK

Release 3.2.2

Grzegorz Mrukwa

Jan 29, 2023

COMMAND LINE INTERFACE

Here you can find a list of documentation topics covered by this page.

CHAPTER

ONE

CLUSTER ANALYSIS WITH `FIT-CLUSTERS` EXECUTABLE

Note: `fit-clusters` requires installation with `gin` extras, e.g. `pip install divik[gin]`

fit-clusters is just one CLI executable that allows you to run DiviK algorithm, any other clustering algorithms supported by *scikit-learn* or even a pipeline with pre-processing.

1.1 Usage

1.1.1 CLI interface

There are two types of parameters:

1. `--param` - this way you can set the value of a parameter during *fit-clusters* executable launch, i.e. you can overwrite parameter provided in a config file or a default.
2. `--config` - this way you can provide a list of config files. Their content will be treated as a one big (ordered) list of settings. In case of conflict, the later file overwrites a setting provided by earlier one.

These go directly to the CLI.

```
usage: fit-clusters [-h] [--param [PARAM [PARAM ...]]]
                    [--config [CONFIG [CONFIG ...]]]

optional arguments:
-h, --help            show this help message and exit
--param [PARAM [PARAM ...]]
                        List of Gin parameter bindings
--config [CONFIG [CONFIG ...]]
                        List of paths to the config files
```

Sample `fit-clusters` call:

```
fit-clusters \
  --param \
    load_data.path='/data/my_data.csv' \
    DiviK.distance='euclidean' \
    DiviK.use_logfilters=False \
    DiviK.n_jobs=-1 \
  --config \
```

(continues on next page)

(continued from previous page)

```
    my-defaults.gin \
    my-overrides.gin
```

The elaboration of all the parameters is included in *Experiment configuration* and *Model setup*.

1.1.2 Experiment configuration

Following parameters are available when launching experiments:

1. `load_data.path` - path to the file with data for clustering. Observations in rows, features in columns.
2. `load_xy.path` - path to the file with X and Y coordinates for the observations. The number of coordinate pairs must be the same as the number of observations. Only integer coordinates are supported now.
3. `experiment.model` - the clustering model to fit to the data. See more in *Model setup*.
4. `experiment.steps_that_require_xy` - when using scikit-learn Pipeline, it may be required to provide spatial coordinates to fit specific algorithms. This parameter accepts the list of the steps that should be provided with spatial coordinates during pipeline execution (e.g. `EximsSelector`).
5. `experiment.destination` - the destination directory for the experiment outputs. Default `result`.
6. `experiment.omit_datetime` - if `True`, the destination directory will be directly populated with the results of the experiment. Otherwise, a subdirectory with date and time will be created to keep separation between runs. Default `False`.
7. `experiment.verbose` - if `True`, extends the messaging on the console. Default *False*.
8. `experiment.exist_ok` - if `True`, the experiment will not fail if the destination directory exists. This is to avoid results overwrites. Default `False`.

1.2 Model setup

1.2.1 `divik` models

To use DiviK algorithm in the experiment, a config file must:

1. Import the algorithms to the scope, e.g.:

```
import divik.cluster
```

2. Point experiment which algorithm to use, e.g.:

```
experiment.model = @DiviK()
```

3. Configure the algorithm, e.g.:

```
DiviK.distance = 'euclidean'
DiviK.verbose = True
```

1.2.1.1 Sample config with `KMeans`

Below you can check sample configuration file, that sets up simple KMeans:

```
import divik.cluster

KMeans.n_clusters = 3
KMeans.distance = "correlation"
KMeans.init = "kdtree_percentile"
KMeans.leaf_size = 0.01
KMeans.percentile = 99.0
KMeans.max_iter = 100
KMeans.normalize_rows = True

experiment.model = @KMeans()
experiment.omit_datetime = True
experiment.verbose = True
experiment.exist_ok = True
```

1.2.1.2 Sample config with `DiviK`

Below is the configuration file with full setup of DiviK. `DiviK` requires an automated clustering method for stop condition and a separate one for clustering. Here we use `GAPSearch` for stop condition and `DunnSearch` for selecting the number of clusters. These in turn require a `KMeans` method set for a specific distance method, etc.:

```
import divik.cluster

KMeans.n_clusters = 1
KMeans.distance = "correlation"
KMeans.init = "kdtree_percentile"
KMeans.leaf_size = 0.01
KMeans.percentile = 99.0
KMeans.max_iter = 100
KMeans.normalize_rows = True

GAPSearch.kmeans = @KMeans()
GAPSearch.max_clusters = 2
GAPSearch.n_jobs = 1
GAPSearch.seed = 42
GAPSearch.n_trials = 10
GAPSearch.sample_size = 1000
GAPSearch.drop_unfit = True
GAPSearch.verbose = True

DunnSearch.kmeans = @KMeans()
DunnSearch.max_clusters = 10
DunnSearch.method = "auto"
DunnSearch.inter = "closest"
DunnSearch.intra = "furthest"
DunnSearch.sample_size = 1000
DunnSearch.seed = 42
DunnSearch.n_jobs = 1
```

(continues on next page)

(continued from previous page)

```
DunnSearch.drop_unfit = True
DunnSearch.verbose = True

DiviK.kmeans = @DunnSearch()
DiviK.fast_kmeans = @GAPSearch()
DiviK.distance = "correlation"
DiviK.minimal_size = 200
DiviK.rejection_size = 2
DiviK.minimal_features_percentage = 0.005
DiviK.features_percentage = 1.0
DiviK.normalize_rows = True
DiviK.use_logfilters = True
DiviK.filter_type = "gmm"
DiviK.n_jobs = 1
DiviK.verbose = True

experiment.model = @DiviK()
experiment.omit_datetime = True
experiment.verbose = True
experiment.exist_ok = True
```

1.2.2 `scikit-learn` models

For a model to be used with `fit-clusters`, it needs to be marked as `gin.configurable`. While it is true for DiviK and remaining algorithms within `divik` package, `scikit-learn` requires additional setup.

1. Import helper module:

   ```
   import divik.core.gin_sklearn_configurables
   ```

2. Point experiment which algorithm to use, e.g.:

   ```
   experiment.model = @MeanShift()
   ```

3. Configure the algorithm, e.g.:

   ```
   MeanShift.n_jobs = -1
   MeanShift.max_iter = 300
   ```

Warning: Importing both `scikit-learn` and `divik` will result in an ambiguity when using e.g. `KMeans`. In such a case it is necesary to point specific algorithms by a full name, e.g. `divik.cluster._kmeans._core.KMeans`.

1.2.2.1 Sample config with `MeanShift`

Below you can check sample configuration file, that sets up simple MeanShift:

```
import divik.core.gin_sklearn_configurables

MeanShift.cluster_all = True
MeanShift.n_jobs = -1
MeanShift.max_iter = 300

experiment.model = @MeanShift()
experiment.omit_datetime = True
experiment.verbose = True
experiment.exist_ok = True
```

1.2.3 Pipelines

`scikit-learn` Pipelines have a separate section to provide an additional explanation, even though these are part of `scikit-learn`.

1. Import helper module:

   ```
   import divik.core.gin_sklearn_configurables
   ```

2. Import the algorithms into the scope:

   ```
   import divik.feature_extraction
   ```

3. Point experiment which algorithm to use, e.g.:

   ```
   experiment.model = @Pipeline()
   ```

4. Configure the algorithms, e.g.:

   ```
   MeanShift.n_jobs = -1
   MeanShift.max_iter = 300
   ```

5. Configure the pipeline:

   ```
   Pipeline.steps = [
       ('histogram_equalization', @HistogramEqualization()),
       ('exims', @EximsSelector()),
       ('pca', @KneePCA()),
       ('mean_shift', @MeanShift()),
   ]
   ```

6. (If needed) configure steps that require spatial coordinates:

   ```
   experiment.steps_that_require_xy = ['exims']
   ```

1.2.3.1 Sample config with `Pipeline`

Below you can check sample configuration file, that sets up simple Pipeline:

```
import divik.core.gin_sklearn_configurables
import divik.feature_extraction

MeanShift.n_jobs = -1
MeanShift.max_iter = 300

Pipeline.steps = [
    ('histogram_equalization', @HistogramEqualization()),
    ('exims', @EximsSelector()),
    ('pca', @KneePCA()),
    ('mean_shift', @MeanShift()),
]

experiment.model = @Pipeline()
experiment.steps_that_require_xy = ['exims']
experiment.omit_datetime = True
experiment.verbose = True
experiment.exist_ok = True
```

1.2.4 Custom models

The `fit-clusters` executable can work with custom algorithms as well.

1. Mark an algorithm class `gin.configurable` at the definition time:

   ```
   import gin

   @gin.configurable
   class MyClustering:
       pass
   ```

   or when importing them from a library:

   ```
   import gin

   gin.external_configurable(MyClustering)
   ```

2. Define artifacts saving methods:

   ```
   from divik.core.io import saver

   @saver
   def save_my_clustering(model, fname_fn, **kwargs):
       if not hasattr(model, 'my_custom_field_'):
           return
       # custom saving logic comes here
   ```

   There are some default savers defined, which are compatible with lots of `divik` and `scikit-learn` algorithms, supporting things like:

- model pickling
- JSON summary saving
- labels saving (`.npy`, `.csv`)
- centroids saving (`.npy`, `.csv`)
- pipeline saving

A `saver` should be highly reusable and could be a pleasant contribution to the `divik` library.

3. In config, import the module which marks your algorithm configurable:

```
import myclustering
```

4. Continue with the algorithm setup and plumbing as in the previous scenarios

CHAPTER

TWO

COMPUTATIONAL MODULES

2.1 divik.cluster module

Clustering methods

class divik.cluster.**DiviK**(*kmeans*, *fast_kmeans=None*, *distance='correlation'*, *minimal_size=None*, *rejection_size=None*, *rejection_percentage=None*, *minimal_features_percentage=0.01*, *features_percentage=0.05*, *normalize_rows=None*, *neutral=None*, *use_logfilters=False*, *filter_type='gmm'*, *n_jobs=None*, *verbose=False*)

DiviK clustering

Parameters

kmeans: AutoKMeans
A self-tuning KMeans estimator for the purpose of clustering. Two implementations are provided in *divik.cluster* package: *DunnSearch* and *GAPSearch*.

fast_kmeans: GAPSearch, optional, default: None
A self-tuning KMeans estimator for the purpose of stop condition check. If None, the *kmeans* parameter is assumed to be the *GAPSearch* instance.

distance: str, optional, default: 'correlation'
The distance metric between points, centroids and for GAP index estimation. One of the distances supported by scipy package.

minimal_size: int or float, optional, default: None
The minimum size of the region (the number of observations) to be considered for any further divisions. If provided number is between 0 and 1, it is considered a rate of training dataset size. When left None, defaults to 0.1% of the training dataset size.

rejection_size: int, optional, default: None
Size under which split will be rejected - if a cluster appears in the split that is below rejection_size, the split is considered improper and discarded. This may be useful for some domains (like there is no justification for a 3-cells cluster in biological data). By default, no segmentation is discarded, as careful post-processing provides the same advantage.

rejection_percentage: float, optional, default: None
An alternative to `rejection_size`, with the same behavior, but this parameter is related to the training data size percentage. By default, no segmentation is discarded.

minimal_features_percentage: float, optional, default: 0.01
The minimal percentage of features that must be preserved after GMM-based feature selection. By default at least 1% of features is preserved in the filtration process.

features_percentage: float, optional, default: 0.05
The target percentage of features that are used by fallback percentage filter for 'outlier' filter.

normalize_rows: bool, optional, default: None
Whether to normalize each row of the data to the norm of 1. By default, it normalizes rows for correlation metric, does no normalization otherwise.

neutral: float, optional, default: None
Element skipped when filtering.

use_logfilters: bool, optional, default: False
Whether to compute logarithm of feature characteristic instead of the characteristic itself. This may improve feature filtering performance, depending on the distribution of features, however all the characteristics (mean, variance) have to be positive for that - filtering will fail otherwise. This is useful for specific cases in biology where the distribution of data may actually require this option for any efficient filtering.

filter_type: {'gmm', 'outlier', 'auto', 'none'}, default: 'gmm'

- 'gmm' - usual Gaussian Mixture Model-based filtering, useful for high

dimensional cases - 'outlier' - robust outlier detection-based filtering, useful for low dimensional cases. In the case of no outliers, percentage-based filtering is applied. - 'auto' - automatically selects between 'gmm' and 'outlier' based on the dimensionality. When more than 250 features are present, 'gmm' is chosen. - 'none' - feature selection is disabled

n_jobs: int, optional, default: None
The number of jobs to use for the computation. This works by computing each of the GAP index evaluations in parallel and by making predictions in parallel.

verbose: bool, optional, default: False
Whether to report the progress of the computations.

Examples

```
>>> from divik.cluster import DiviK, DunnSearch, KMeans
>>> from sklearn.datasets import make_blobs
>>> X, _ = make_blobs(n_samples=1_000,
...     n_features=2,
...     centers=7,
...     random_state=42,
... )
>>> divik = DiviK(
...     kmeans=DunnSearch(  # we want to use Dunn's method for finding the optimal␣
↪number of clusters
...         kmeans=KMeans(
...             n_clusters=2,  # it is required, like in scikit-learn, but you can␣
↪provide any number here,
...                            # DunnSearch will override it anyway
...         ),
...         max_clusters=5,  # for the sake of the example I'll keep it low
...     ),
...     minimal_size=100,  # for the sake of the example, I won't split clusters␣
↪with less than 100 elements
...     filter_type='none',  # we have 2 features in sample data, feature selection␣
↪would be pointless
```

(continues on next page)

(continued from previous page)

```
... ).fit(X)
>>> divik.n_clusters_
22
>>> divik.labels_
array([1, 1, 1, 0, ..., 0, 0], dtype=int32)
>>> divik.predict([[0, ..., 0], [12, ..., 3]])
array([1, 0], dtype=int32)
>>> divik.cluster_centers_
array([[10., ...,  2.],
       ...,
       [ 1, ...,  2.]])
```

Attributes

result_: divik.DivikResult
Hierarchical structure describing all the consecutive segmentations.

labels_:
Labels of each point

centroids_: array, [n_clusters, n_features]
Coordinates of cluster centers. If the algorithm stops before fully converging, these will not be consistent with `labels_`. Also, the distance between points and respective centroids must be captured in appropriate features subspace. This is realized by the `transform` method.

filters_: array, [n_clusters, n_features]
Filters that were applied to the feature space on the level that was the final segmentation for a subset.

depth_: int
The number of hierarchy levels in the segmentation.

n_clusters_: int
The final number of clusters in the segmentation, on the tree leaf level.

paths_: Dict[int, Tuple[int]]
Describes how the cluster number corresponds to the path in the tree. Element of the tuple indicates the sub-segment number on each tree level.

reverse_paths_: Dict[Tuple[int], int]
Describes how the path in the tree corresponds to the cluster number. For more details see `paths_`.

Methods

fit(X[, y])	Compute DiviK clustering.
fit_predict(X[, y])	Compute cluster centers and predict cluster index for each sample.
fit_transform(X[, y])	Compute clustering and transform X to cluster-distance space.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the closest cluster each sample in X belongs to.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Transform X to a cluster-distance space.

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

Compute DiviK clustering.

Parameters

X
[array-like or sparse matrix, shape=(n_samples, n_features)] Training instances to cluster. It must be noted that the data will be converted to C ordering, which will cause a memory copy if the given data is not C-contiguous.

y
[Ignored] not used, present here for API consistency by convention.

`fit_predict`(*X*, *y=None*)

Compute cluster centers and predict cluster index for each sample.

Convenience method; equivalent to calling fit(X) followed by predict(X).

Parameters

X
[{array-like, sparse matrix}, shape = [n_samples, n_features]] New data to transform.

y
[Ignored] not used, present here for API consistency by convention.

Returns

labels
[array, shape [n_samples,]] Index of the cluster each sample belongs to.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Compute clustering and transform X to cluster-distance space.

Equivalent to fit(X).transform(X), but more efficiently implemented.

Parameters

X
[{array-like, sparse matrix}, shape = [n_samples, n_features]] New data to transform.

y
[Ignored] not used, present here for API consistency by convention.

Returns

X_new
[array, shape [n_samples, **self.n_clusters_**]] X transformed in the new space.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Predict the closest cluster each sample in X belongs to.

In the vector quantization literature, *cluster_centers_* is called the code book and each value returned by *predict* is the index of the closest code in the code book.

Parameters

X
[{array-like, sparse matrix}, shape = [n_samples, n_features]] New data to predict.

Returns

labels
[array, shape [n_samples,]] Index of the cluster each sample belongs to.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Transform X to a cluster-distance space.

In the new space, each dimension is the distance to the cluster centers. Note that even if X is sparse, the array returned by *transform* will typically be dense.

Parameters

X
[{array-like, sparse matrix}, shape = [n_samples, n_features]] New data to transform.

Returns

X_new
[array, shape [n_samples, **self.n_clusters_**]] X transformed in the new space.

`class divik.cluster.`**`DunnSearch`**(*kmeans*, *max_clusters*, *min_clusters=2*, *method='full'*, *inter='centroid'*, *intra='avg'*, *sample_size=1000*, *n_trials=10*, *seed=42*, *n_jobs=1*, *drop_unfit=False*, *verbose=False*)

Select best number of clusters for k-means

Parameters

kmeans
[KMeans] KMeans object to tune

max_clusters: int
The maximal number of clusters to form and score.

min_clusters: int, default: 1
The minimal number of clusters to form and score.

method: {'full', 'sampled', 'auto'}, default: 'full'
Whether to run full computations or approximate. - full - always computes full Dunn's index, without sampling - sampled - samples the clusters to reduce computational overhead - auto - switches the above methods to provide best performance-quality trade-off.

inter
[{'centroid', 'closest'}, default: 'centroid'] How the distance between clusters is computed. For more details see *dunn*.

intra
[{'avg', 'furthest'}, default: 'avg'] How the cluster internal distance is computed. For more details see *dunn*.

sample_size
[int, default: 1000] Size of the sample used to compute Dunn index in *auto* or *sampled* scenario.

n_trials
[int, default: 10] Number of trials to use when computing Dunn index in *auto* or *sampled* scenario.

seed
[int, default: 42] Random seed for the reproducibility of subset draws in Dunn *auto* or *sampled* scenario.

n_jobs: int, default: 1
The number of jobs to use for the computation. This works by computing each of the clustering & scoring runs in parallel.

drop_unfit: bool, default: False
If True, drops the estimators that did not fit the data.

verbose: bool, default: False
If True, shows progress with tqdm.

Attributes

cluster_centers_: array, [n_clusters, n_features]
Coordinates of cluster centers.

labels_:
Labels of each point.

estimators_: List[KMeans]
KMeans instances for n_clusters in range [min_clusters, max_clusters].

scores_: array, [max_clusters - min_clusters + 1,]
Array with scores for each estimator.

n_clusters_: int
Estimated optimal number of clusters.

best_score_: float
Score of the optimal estimator.

best_: KMeans
The optimal estimator.

Methods

fit(X[, y])	Compute k-means clustering and estimate optimal number of clusters.
fit_predict(X[, y])	Perform clustering on *X* and returns cluster labels.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the closest cluster each sample in X belongs to.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Transform X to a cluster-distance space.

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

Compute k-means clustering and estimate optimal number of clusters.

Parameters

X
[array-like or sparse matrix, shape=(n_samples, n_features)] Training instances to cluster. It must be noted that the data will be converted to C ordering, which will cause a memory copy if the given data is not C-contiguous.

y
[Ignored] not used, present here for API consistency by convention.

fit_predict(*X*, *y=None*)

Perform clustering on *X* and returns cluster labels.

Parameters

X
[array-like of shape (n_samples, n_features)] Input data.

y
[Ignored] Not used, present for API consistency by convention.

Returns

labels
[ndarray of shape (n_samples,), dtype=np.int64] Cluster labels.

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

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Predict the closest cluster each sample in X belongs to.

In the vector quantization literature, *cluster_centers_* is called the code book and each value returned by *predict* is the index of the closest code in the code book.

Parameters

X
[{array-like, sparse matrix}, shape = [n_samples, n_features]] New data to predict.

Returns

labels
[array, shape [n_samples,]] Index of the cluster each sample belongs to.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Transform X to a cluster-distance space.

In the new space, each dimension is the distance to the cluster centers. Note that even if X is sparse, the array returned by *transform* will typically be dense.

Parameters

X
[{array-like, sparse matrix}, shape = [n_samples, n_features]] New data to transform.

Returns

X_new
[array, shape [n_samples, k]] X transformed in the new space.

class divik.cluster.**GAPSearch**(*kmeans*, *max_clusters*, *min_clusters=1*, *reference_sampler=None*, *n_jobs=1*, *seed=0*, *n_trials=10*, *sample_size=1000*, *drop_unfit=False*, *verbose=False*)

Select best number of cluters for k-means

Parameters

kmeans
[KMeans] KMeans object to tune

max_clusters: int
The maximal number of clusters to form and score.

min_clusters: int, default: 1
The minimal number of clusters to form and score.

reference_sampler: BaseSampler, default: None
Sampler for reference dataset sampling in GAP statistic computations.

n_jobs: int, default: 1
The number of jobs to use for the computation. This works by computing each of the clustering & scoring runs in parallel.

seed: int, default: 0
Random seed for generating uniform data sets.

n_trials: int, default: 10
Number of data sets drawn as a reference.

sample_size
[int, default: 1000] Size of the sample used for GAP statistic computation. Used only if introduces speedup.

drop_unfit: bool, default: False
If True, drops the estimators that did not fit the data.

verbose: bool, default: False
If True, shows progress with tqdm.

Attributes

cluster_centers_: array, [n_clusters, n_features]
Coordinates of cluster centers.

labels_:
Labels of each point.

estimators_: List[KMeans]
KMeans instances for n_clusters in range [min_clusters, max_clusters].

scores_: array, [max_clusters - min_clusters + 1, ?]
Array with scores for each estimator in each row.

n_clusters_: int
Estimated optimal number of clusters.

best_score_: float
Score of the optimal estimator.

best_: KMeans
The optimal estimator.

Methods

fit(X[, y])	Compute k-means clustering and estimate optimal number of clusters.
fit_predict(X[, y])	Perform clustering on *X* and returns cluster labels.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the closest cluster each sample in X belongs to.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Transform X to a cluster-distance space.

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

Compute k-means clustering and estimate optimal number of clusters.

Parameters

X
[array-like or sparse matrix, shape=(n_samples, n_features)] Training instances to cluster. It must be noted that the data will be converted to C ordering, which will cause a memory copy if the given data is not C-contiguous.

y
[Ignored] not used, present here for API consistency by convention.

fit_predict(*X*, *y=None*)

Perform clustering on *X* and returns cluster labels.

Parameters

X
[array-like of shape (n_samples, n_features)] Input data.

y
[Ignored] Not used, present for API consistency by convention.

Returns

labels
[ndarray of shape (n_samples,), dtype=np.int64] Cluster labels.

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

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Predict the closest cluster each sample in X belongs to.

In the vector quantization literature, *cluster_centers_* is called the code book and each value returned by *predict* is the index of the closest code in the code book.

Parameters

X
[{array-like, sparse matrix}, shape = [n_samples, n_features]] New data to predict.

Returns

labels
[array, shape [n_samples,]] Index of the cluster each sample belongs to.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Transform X to a cluster-distance space.

In the new space, each dimension is the distance to the cluster centers. Note that even if X is sparse, the array returned by *transform* will typically be dense.

Parameters

X
[{array-like, sparse matrix}, shape = [n_samples, n_features]] New data to transform.

Returns

X_new
[array, shape [n_samples, k]] X transformed in the new space.

class divik.cluster.**KMeans**(*n_clusters*, *distance='euclidean'*, *init='percentile'*, *percentile=95.0*, *leaf_size=0.01*, *max_iter=100*, *normalize_rows=False*, *allow_dask=False*)

K-Means clustering

Parameters

n_clusters
[int] The number of clusters to form as well as the number of centroids to generate.

distance
[str, optional, default: 'euclidean'] Distance measure. One of the distances supported by scipy package.

init
[{'percentile', 'extreme', 'kdtree', 'kdtree_percentile'}] Method for initialization, defaults to 'percentile':

'percentile' : selects initial cluster centers for k-mean clustering starting from specified percentile of distance to already selected clusters

'extreme': selects initial cluster centers for k-mean clustering starting from the furthest points to already specified clusters

'kdtree': selects initial cluster centers for k-mean clustering starting from centroids of KD-Tree boxes

'kdtree_percentile': selects initial cluster centers for k-means clustering starting from centroids of KD-Tree boxes containing specified percentile. This should be more robust against outliers.

percentile
[float, default: 95.0] Specifies the starting percentile for 'percentile' initialization. Must be within range [0.0, 100.0]. At 100.0 it is equivalent to 'extreme' initialization.

leaf_size
[int or float, optional (default 0.01)] Desired leaf size in kdtree initialization. When int, the box size will be between *leaf_size* and *2 * leaf_size*. When float, it will be between *leaf_size * n_samples* and *2 * leaf_size * n_samples*

max_iter
[int, default: 100] Maximum number of iterations of the k-means algorithm for a single run.

normalize_rows
[bool, default: False] If True, rows are translated to mean of 0.0 and scaled to norm of 1.0.

allow_dask
[bool, default: False] If True, automatically selects dask as computations backend whenever reasonable. Default *False* since it cannot be used together with *multiprocessing.Pool* and everywhere *n_jobs* must be set to *1*.

Attributes

cluster_centers_
[array, [n_clusters, n_features]] Coordinates of cluster centers.

labels_
Labels of each point

Methods

fit(X[, y])	Compute k-means clustering.
fit_predict(X[, y])	Perform clustering on *X* and returns cluster labels.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the closest cluster each sample in X belongs to.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Transform X to a cluster-distance space.

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

Compute k-means clustering.

Parameters

X
[array-like or sparse matrix, shape=(n_samples, n_features)] Training instances to cluster. It must be noted that the data will be converted to C ordering, which will cause a memory copy if the given data is not C-contiguous.

y
[Ignored] not used, present here for API consistency by convention.

fit_predict(*X*, *y=None*)

Perform clustering on *X* and returns cluster labels.

Parameters

X
[array-like of shape (n_samples, n_features)] Input data.

y
[Ignored] Not used, present for API consistency by convention.

Returns

labels
[ndarray of shape (n_samples,), dtype=np.int64] Cluster labels.

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

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*)

Predict the closest cluster each sample in X belongs to.

In the vector quantization literature, *cluster_centers_* is called the code book and each value returned by *predict* is the index of the closest code in the code book.

Parameters

X
[{array-like, sparse matrix}, shape = [n_samples, n_features]] New data to predict.

Returns

labels
[array, shape [n_samples,]] Index of the cluster each sample belongs to.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Transform X to a cluster-distance space.

In the new space, each dimension is the distance to the cluster centers. Note that even if X is sparse, the array returned by *transform* will typically be dense.

Parameters

X
[{array-like, sparse matrix}, shape = [n_samples, n_features]] New data to transform.

Returns

X_new
[array, shape [n_samples, k]] X transformed in the new space.

class divik.cluster.**TwoStep**(*clusterer*, *n_subsets=10*, *random_state=42*)

Perform a two-step clustering with a given clusterer

Separates a dataset into `n_subsets`, processes each of them separately and then combines the results.

Works with centroid-based clustering methods, as it requires cluster representatives to combine the result.

Parameters

clusterer
[Union[AutoKMeans, Pipeline, KMeans]] A centroid-based estimator for the purpose of clustering.

n_subsets
[int, default 10] The number of subsets into which the original dataset should be separated

random_state
[int, default 42] Random state to use for seeding the random number generator.

Examples

```
>>> from sklearn.datasets import make_blobs
>>> from divik.cluster import KMeans, TwoStep
>>> X, _ = make_blobs(
...     n_samples=10_000, n_features=2, centers=3, random_state=42
... )
>>> kmeans = KMeans(n_clusters=3)
>>> ctr = TwoStep(kmeans).fit(X)
```

Methods

fit_predict(X[, y])	Perform clustering on *X* and returns cluster labels.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

fit	
predict	

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

fit_predict(*X*, *y=None*)

Perform clustering on *X* and returns cluster labels.

Parameters

X
[array-like of shape (n_samples, n_features)] Input data.

y
[Ignored] Not used, present for API consistency by convention.

Returns

labels
[ndarray of shape (n_samples,), dtype=np.int64] Cluster labels.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`predict`(*X*, *y=None*)

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

2.2 `divik.feature_extraction` module

Unsupervised feature extraction methods

`class divik.feature_extraction.HistogramEqualization`(*n_bins=256*, *n_jobs=-1*)

Equalize histogram of the features to increase contrast

Based on https://github.com/scikit-image/scikit-image/blob/master/skimage/exposure/exposure.py#L187-L223

Parameters

n_bins
[int, default 256] Number of bins for histogram equalization.

n_jobs
[int, default -1] Number of CPU cores to use during equalization

Attributes

cdf_
[array] Values of cumulative distribution function for all the features

bins_
[array] Bin centers for all the features

Methods

fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

fit	
transform	

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

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

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*, *y=None*)

`class divik.feature_extraction.KneePCA`(*whiten=False*, *refit=False*)

Principal component analysis (PCA) with knee method

PCA with automated components selection based on knee method over cumulative explained variance. Remaining components are discarded.

Parameters

whiten
[bool, optional (default False)] When True (False by default) the `pca_.components_` vectors are multiplied by the square root of n_samples and then divided by the singular values to ensure uncorrelated outputs with unit component-wise variances.

Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making their data respect some hard-wired assumptions.

refit
[bool, optional (default False)] When `True` (`False` by default) the `pca_` is re-fit with the smaller number of components. This could reduce memory footprint, but requires training fitting PCA.

Attributes

pca_
[PCA] Fit PCA estimator.

n_components_
[int] The number of selected components.

Methods

fit(X[, y])	Fit the model from data in X.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
inverse_transform(X)	Transform data back to its original space.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Apply dimensionality reduction to X.

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

Fit the model from data in X.

Parameters

X
[array-like, shape (n_samples, n_features)] Training vector, where `n_samples` is the number of samples and `n_features` is the number of features.

Y: Ignored.

Returns

self
[object] Returns the instance itself.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*X*)

Transform data back to its original space.

In other words, return an input X_original whose transform would be X.

Parameters

X
[array-like, shape (n_samples, n_components)] New data, where `n_samples` is the number of samples and `n_components` is the number of components.

Returns

X_original array-like, shape (n_samples, n_features)

Notes

If whitening is enabled, inverse_transform will compute the exact inverse operation, which includes reversing whitening.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*, *y=None*)

Apply dimensionality reduction to X.

X is projected on the first principal components previously extracted from a training set.

Parameters

X
[array-like, shape (n_samples, n_features)] New data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

X_new
[array-like, shape (n_samples, n_components)]

Examples

```
>>> import numpy as np
>>> from divik.feature_extraction import KneePCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> pca = KneePCA(refit=True)
>>> pca.fit(X)
KneePCA(refit=True)
>>> pca.transform(X)
```

class divik.feature_extraction.**LocallyAdjustedRbfSpectralEmbedding**(*distance='euclidean'*, *n_components=2*, *random_state=None*, *eigen_solver=None*, *n_neighbors=None*, *n_jobs=1*)

Spectral embedding for non-linear dimensionality reduction.

Forms an affinity matrix given by the specified function and applies spectral decomposition to the corresponding graph laplacian. The resulting transformation is given by the value of the eigenvectors for each data point.

Note : Laplacian Eigenmaps is the actual algorithm implemented here.

Parameters

distance
[{'braycurtis', 'canberra', 'chebyshev', 'cityblock',]

'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'atching', 'minkowski', 'rogerstanimoto', 'russellrao', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'}
Distance measure, defaults to `euclidean`. These are the distances supported by scipy package.

n_components
[integer, default: 2] The dimension of the projected subspace.

random_state
[int, RandomState instance or None, optional, default: None] A pseudo random number generator used for the initialization of the lobpcg eigenvectors. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver == amg`.

eigen_solver
[{None, 'arpack', 'lobpcg', or 'amg'}] The eigenvalue decomposition strategy to use. AMG requires pyamg to be installed. It can be faster on very large, sparse problems, but may also lead to instabilities.

n_neighbors
[int, default][max(n_samples/10 , 1)] Number of nearest neighbors for nearest_neighbors graph building.

n_jobs
[int, optional (default = 1)] The number of parallel jobs to run. If `-1`, then the number of jobs is set to the number of CPU cores.

References

- A Tutorial on Spectral Clustering, 2007 Ulrike von Luxburg http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.9323
- On Spectral Clustering: Analysis and an algorithm, 2001 Andrew Y. Ng, Michael I. Jordan, Yair Weiss http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.8100
- Normalized cuts and image segmentation, 2000 Jianbo Shi, Jitendra Malik http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.160.2324

Attributes

embedding_
[array, shape = (n_samples, n_components)] Spectral embedding of the training matrix.

Methods

fit(X[, y])	Fit the model from data in X.
fit_transform(X[, y])	Fit the model from data in X and transform X.
get_params([deep])	Get parameters for this estimator.
save(destination)	Save embedding to a directory
set_params(**params)	Set the parameters of this estimator.

transform	

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

Fit the model from data in X.

Parameters

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

Y: Ignored.

Returns

self
[object] Returns the instance itself.

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

Fit the model from data in X and transform X.

Parameters

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

Y: Ignored.

Returns

X_new
[array-like, shape (n_samples, n_components)]

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`save`(*destination*)

Save embedding to a directory

Parameters

destination
[str] Directory to save the embedding.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

transform(*X*, *y=None*)

2.3 divik.feature_selection module

Unsupervised feature selection methods

class divik.feature_selection.**EximsSelector**

Select features based on their spatial distribution

Preserves features that yield biologically plausible structures.

References

Wijetunge, Chalini D., et al. "EXIMS: an improved data analysis pipeline based on a new peak picking method for EXploring Imaging Mass Spectrometry data." Bioinformatics 31.19 (2015): 3198-3206. https://academic.oup.com/bioinformatics/article/31/19/3198/212150

Methods

fit(X[, y, xy])	Learn data-driven feature thresholds from X.
fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Mask feature names according to selected features.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected.
inverse_transform(X)	Reverse the transformation operation.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

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

Learn data-driven feature thresholds from X.

Parameters

X
[{array-like, sparse matrix}, shape (n_samples, n_features)] Sample vectors from which to compute feature characteristic.

y
[any] Ignored. This parameter exists only for compatibility with sklearn.pipeline.Pipeline.

xy
[array-like, shape (n_samples, 2)] Spatial coordinates of the samples. Expects integers, indices over am image.

Returns

self

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

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Mask feature names according to selected features.

Parameters

input_features
[array-like of str or None, default=None] Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then names are generated: *[x0, x1, …, x(n_features_in_)]*.
- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns

feature_names_out
[ndarray of str objects] Transformed feature names.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If *indices* is False,

this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If *indices* is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`inverse_transform`(*X*)

Reverse the transformation operation.

Parameters

X
[array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_original_features]] *X* with columns of zeros inserted where features would have been removed by *transform()*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Reduce X to the selected features.

Parameters

X
[array of shape [n_samples, n_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

`class divik.feature_selection.GMMSelector`(*stat*, *neutral=None*, *use_log=False*, *n_candidates=None*, *min_features=1*, *min_features_rate=0.0*, *preserve_high=True*, *max_components=10*)

Feature selector that removes low- or high- mean or variance features

Gaussian Mixture Modeling is applied to the features' characteristics and components are obtained. Crossing points of the components are considered candidate thresholds. Out of these up to `n_candidates` components are removed in such a way that at least `min_features` or `min_features_rate` features are retained.

This feature selection algorithm looks only at the features (X), not the desired outputs (y), and can thus be used for unsupervised learning.

Parameters

stat: {'mean', 'var', 'cv'}
Kind of statistic to be computed out of the feature.

neutral: float, optional, default: None
This element will be omitted from the computation of the statistic.

use_log: bool, optional, default: False
Whether to use the logarithm of feature characteristic instead of the characteristic itself. This may improve feature filtering performance, depending on the distribution of features, however all the characteristics (mean, variance) have to be positive for that - filtering will fail otherwise. This is useful for specific cases in biology where the distribution of data may actually require this option for any efficient filtering.

n_candidates: int, optional, default: None
How many candidate thresholds to use at most. `0` preserves all the features (all candidate thresholds are discarded), `None` allows to remove all but one component (all candidate thresholds are retained). Negative value means to discard up to all but `-n_candidates` candidates, e.g. `-1` will retain at least two components (one candidate threshold is removed).

min_features: int, optional, default: 1
How many features must be preserved. Candidate thresholds are tested against this value, and if they retain less features, less conservative thresholds is selected.

min_features_rate: float, optional, default: 0.0
Similar to `min_features` but relative to the input data features number.

preserve_high: bool, optional, default: True
Whether to preserve the high-characteristic features or low-characteristic ones.

max_components: int, optional, default: 10
The maximum number of components used in the GMM decomposition.

Examples

```
>>> import numpy as np
>>> import divik.feature_selection as fs
>>> np.random.seed(42)
>>> labels = np.concatenate([30 * [0] + 20 * [1] + 30 * [2] + 40 * [3]])
>>> data = labels * 5 + np.random.randn(*labels.shape)
>>> fs.GMMSelector('mean').fit_transform(data)
array([[14.78032811 15.35711257 ... 15.75193303]])
>>> fs.GMMSelector('mean', preserve_high=False).fit_transform(data)
array([[ 0.49671415 -0.1382643  ... -0.29169375]])
>>> fs.GMMSelector('mean', n_discard=-1).fit_transform(data)
array([[10.32408397  9.61491772 ... 15.75193303]])
```

Attributes

vals_: array, shape (n_features,)
Computed characteristic of each feature.

threshold_: float
Threshold value to filter the features by the characteristic.

raw_threshold_: float
Threshold value mapped back to characteristic space (no logarithm, etc.)

selected_: array, shape (n_features,)
Vector of binary selections of the informative features.

Methods

fit(X[, y])	Learn data-driven feature thresholds from X.
fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Mask feature names according to selected features.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected.
inverse_transform(X)	Reverse the transformation operation.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

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

Learn data-driven feature thresholds from X.

Parameters

X
[{array-like, sparse matrix}, shape (n_samples, n_features)] Sample vectors from which to compute feature characteristic.

y
[any] Ignored. This parameter exists only for compatibility with sklearn.pipeline.Pipeline.

Returns

self

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

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

get_feature_names_out(*input_features=None*)

Mask feature names according to selected features.

Parameters

input_features
[array-like of str or None, default=None] Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then names are generated: *[x0, x1, …, x(n_features_in_)]*.
- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns

feature_names_out
[ndarray of str objects] Transformed feature names.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If *indices* is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If *indices* is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`inverse_transform`(*X*)

Reverse the transformation operation.

Parameters

X
[array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_original_features]] *X* with columns of zeros inserted where features would have been removed by *transform()*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Reduce X to the selected features.

Parameters

X
[array of shape [n_samples, n_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

`class` divik.feature_selection.**HighAbundanceAndVarianceSelector**(*use_log=False*, *min_features=1*, *min_features_rate=0.0*, *max_components=10*)

Feature selector that removes low-mean and low-variance features

Exercises `GMMSelector` to filter out the low-abundance noise features and select high-variance informative features.

This feature selection algorithm looks only at the features (X), not the desired outputs (y), and can thus be used for unsupervised learning.

Parameters

use_log: bool, optional, default: False
Whether to use the logarithm of feature characteristic instead of the characteristic itself. This may improve feature filtering performance, depending on the distribution of features, however all the characteristics (mean, variance) have to be positive for that - filtering will fail otherwise. This is useful for specific cases in biology where the distribution of data may actually require this option for any efficient filtering.

min_features: int, optional, default: 1
How many features must be preserved.

min_features_rate: float, optional, default: 0.0
Similar to `min_features` but relative to the input data features number.

max_components: int, optional, default: 10
The maximum number of components used in the GMM decomposition.

Examples

```
>>> import numpy as np
>>> import divik.feature_selection as fs
>>> np.random.seed(42)
>>> # Data in this case must be carefully crafted
>>> labels = np.concatenate([30 * [0] + 20 * [1] + 30 * [2] + 40 * [3]])
>>> data = np.vstack(100 * [labels * 10.])
>>> data += np.random.randn(*data.shape)
>>> sub = data[:, :-40]
>>> sub += 5 * np.random.randn(*sub.shape)
>>> # Label 0 has low abundance but high variance
>>> # Label 3 has low variance but high abundance
>>> # Label 1 and 2 has not-lowest abundance and high variance
>>> selector = fs.HighAbundanceAndVarianceSelector().fit(data)
>>> selector.transform(labels.reshape(1,-1))
array([[1 1 1 1 1 ...2 2 2]])
```

Attributes

abundance_selector_: GMMSelector
Selector used to filter out the noise component.

variance_selector_: GMMSelector
Selector used to filter out the non-informative features.

selected_: array, shape (n_features,)
Vector of binary selections of the informative features.

Methods

fit(X[, y])	Learn data-driven feature thresholds from X.
fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Mask feature names according to selected features.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected.
inverse_transform(X)	Reverse the transformation operation.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

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

Learn data-driven feature thresholds from X.

Parameters

X
[{array-like, sparse matrix}, shape (n_samples, n_features)] Sample vectors from which to compute feature characteristic.

y
[any] Ignored. This parameter exists only for compatibility with sklearn.pipeline.Pipeline.

Returns

self

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Mask feature names according to selected features.

Parameters

input_features
[array-like of str or None, default=None] Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then names are generated: *[x0, x1, …, x(n_features_in_)]*.
- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns

feature_names_out
[ndarray of str objects] Transformed feature names.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If *indices* is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If *indices* is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`inverse_transform`(*X*)

Reverse the transformation operation.

Parameters

X
[array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_original_features]] *X* with columns of zeros inserted where features would have been removed by *transform()*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Reduce X to the selected features.

Parameters

X
[array of shape [n_samples, n_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

`class divik.feature_selection.NoSelector`

Dummy selector to use when no selection is supposed to be made.

Methods

fit(X[, y])	Pass data forward
fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Mask feature names according to selected features.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected.
inverse_transform(X)	Reverse the transformation operation.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

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

Pass data forward

Parameters

X
[{array-like, sparse matrix}, shape (n_samples, n_features)] Sample vectors to pass.

y
[any] Ignored. This parameter exists only for compatibility with sklearn.pipeline.Pipeline.

Returns

self

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

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

get_feature_names_out(*input_features=None*)

Mask feature names according to selected features.

Parameters

input_features
[array-like of str or None, default=None] Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then names are generated: *[x0, x1, …, x(n_features_in_)]*.

- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns

feature_names_out
[ndarray of str objects] Transformed feature names.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If *indices* is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If *indices* is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`inverse_transform`(*X*)

Reverse the transformation operation.

Parameters

X
[array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_original_features]] *X* with columns of zeros inserted where features would have been removed by *transform()*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Reduce X to the selected features.

Parameters

X
[array of shape [n_samples, n_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

`class divik.feature_selection.OutlierAbundanceAndVarianceSelector`(*use_log=False*, *min_features_rate=0.01*, *p=0.2*)

Methods

fit(X[, y])	Learn data-driven feature thresholds from X.
fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Mask feature names according to selected features.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected.
inverse_transform(X)	Reverse the transformation operation.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

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

Learn data-driven feature thresholds from X.

Parameters

X
[{array-like, sparse matrix}, shape (n_samples, n_features)] Sample vectors from which to compute feature characteristic.

y
[any] Ignored. This parameter exists only for compatibility with sklearn.pipeline.Pipeline.

Returns

self

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Mask feature names according to selected features.

Parameters

input_features
[array-like of str or None, default=None] Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then names are generated: *[x0, x1, …, x(n_features_in_)]*.
- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns

feature_names_out
[ndarray of str objects] Transformed feature names.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If *indices* is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If *indices* is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`inverse_transform`(*X*)

Reverse the transformation operation.

Parameters

X
[array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_original_features]] *X* with columns of zeros inserted where features would have been removed by *transform()*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Reduce X to the selected features.

Parameters

X
[array of shape [n_samples, n_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

`class divik.feature_selection.OutlierSelector`(*stat*, *use_log=False*, *keep_outliers=False*)

Feature selector that removes outlier features w.r.t. mean or variance

Huberta's outlier detection is applied to the features' characteristics and the outlying features are removed.

This feature selection algorithm looks only at the features (X), not the desired outputs (y), and can thus be used for unsupervised learning.

Parameters

stat: {'mean', 'var'}
Kind of statistic to be computed out of the feature.

use_log: bool, optional, default: False
Whether to use the logarithm of feature characteristic instead of the characteristic itself. This may improve feature filtering performance, depending on the distribution of features, however all the characteristics (mean, variance) have to be positive for that - filtering will fail otherwise. This is useful for specific cases in biology where the distribution of data may actually require this option for any efficient filtering.

keep_outliers: bool, optional, default: False
When True, keeps outliers instead of inlier features.

Attributes

vals_: array, shape (n_features,)
Computed characteristic of each feature.

selected_: array, shape (n_features,)
Vector of binary selections of the informative features.

Methods

fit(X[, y])	Learn data-driven feature thresholds from X.
fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Mask feature names according to selected features.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected.
inverse_transform(X)	Reverse the transformation operation.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

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

Learn data-driven feature thresholds from X.

Parameters

X
[{array-like, sparse matrix}, shape (n_samples, n_features)] Sample vectors from which to compute feature characteristic.

y
[any] Ignored. This parameter exists only for compatibility with sklearn.pipeline.Pipeline.

Returns

self

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

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Mask feature names according to selected features.

Parameters

input_features
[array-like of str or None, default=None] Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then names are generated: *[x0, x1, …, x(n_features_in_)]*.
- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns

feature_names_out
[ndarray of str objects] Transformed feature names.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If *indices* is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If *indices* is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`inverse_transform`(*X*)

Reverse the transformation operation.

Parameters

X
[array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_original_features]] *X* with columns of zeros inserted where features would have been removed by *transform()*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Reduce X to the selected features.

Parameters

X
[array of shape [n_samples, n_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

`class` divik.feature_selection.`PercentageSelector`(*stat*, *use_log=False*, *keep_top=True*, *p=0.2*)

Feature selector that removes / preserves top some percent of features

This feature selection algorithm looks only at the features (X), not the desired outputs (y), and can thus be used for unsupervised learning.

Parameters

stat: {'mean', 'var'}
Kind of statistic to be computed out of the feature.

use_log: bool, optional, default: False
Whether to use the logarithm of feature characteristic instead of the characteristic itself. This may improve feature filtering performance, depending on the distribution of features, however all the characteristics (mean, variance) have to be positive for that - filtering will fail otherwise. This is useful for specific cases in biology where the distribution of data may actually require this option for any efficient filtering.

keep_top: bool, optional, default: True
When True, keeps features with highest value of the characteristic.

p: float, optional, default: 0.2
Rate of features to keep.

Attributes

vals_: array, shape (n_features,)
Computed characteristic of each feature.

threshold_: float
Value of the threshold used for filtering

selected_: array, shape (n_features,)
Vector of binary selections of the informative features.

Methods

fit(X[, y])	Learn data-driven feature thresholds from X.
fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Mask feature names according to selected features.
get_params([deep])	Get parameters for this estimator.
get_support([indices])	Get a mask, or integer index, of the features selected.
inverse_transform(X)	Reverse the transformation operation.
set_params(**params)	Set the parameters of this estimator.
transform(X)	Reduce X to the selected features.

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

Learn data-driven feature thresholds from X.

Parameters

X
[{array-like, sparse matrix}, shape (n_samples, n_features)] Sample vectors from which to compute feature characteristic.

y
[any] Ignored. This parameter exists only for compatibility with sklearn.pipeline.Pipeline.

Returns

self

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

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

get_feature_names_out(*input_features=None*)

Mask feature names according to selected features.

Parameters

input_features
[array-like of str or None, default=None] Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then names are generated: *[x0, x1, …, x(n_features_in_)]*.
- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns

feature_names_out
[ndarray of str objects] Transformed feature names.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If *indices* is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If *indices* is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`inverse_transform`(*X*)

Reverse the transformation operation.

Parameters

X
[array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_original_features]] *X* with columns of zeros inserted where features would have been removed by *transform()*.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Reduce X to the selected features.

Parameters

X
[array of shape [n_samples, n_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

`class divik.feature_selection.SelectorMixin`

Transformer mixin that performs feature selection given a support mask

This mixin provides a feature selector implementation with *transform* and *inverse_transform* functionality given an implementation of *_get_support_mask*.

Methods

fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Mask feature names according to selected features.
get_support([indices])	Get a mask, or integer index, of the features selected.
inverse_transform(X)	Reverse the transformation operation.
transform(X)	Reduce X to the selected features.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_feature_names_out`(*input_features=None*)

Mask feature names according to selected features.

Parameters

input_features
[array-like of str or None, default=None] Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then names are generated: *[x0, x1, …, x(n_features_in_)]*.
- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns

feature_names_out
[ndarray of str objects] Transformed feature names.

`get_support`(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If *indices* is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If *indices* is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

`inverse_transform`(*X*)

Reverse the transformation operation.

Parameters

X
[array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_original_features]] *X* with columns of zeros inserted where features would have been removed by *transform()*.

`transform`(*X*)

Reduce X to the selected features.

Parameters

X
[array of shape [n_samples, n_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

class divik.feature_selection.**StatSelectorMixin**

Transformer mixin that performs feature selection given a support mask

This mixin provides a feature selector implementation with `transform` and `inverse_transform` functionality given that `selected_` is specified during `fit`.

Additionally, provides a `_to_characteristics` and `_to_raw` implementations given `stat`, optionally `use_log` and `preserve_high`.

Methods

fit_transform(X[, y])	Fit to data, then transform it.
get_feature_names_out([input_features])	Mask feature names according to selected features.
get_support([indices])	Get a mask, or integer index, of the features selected.
inverse_transform(X)	Reverse the transformation operation.
transform(X)	Reduce X to the selected features.

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

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

get_feature_names_out(*input_features=None*)

Mask feature names according to selected features.

Parameters

input_features
[array-like of str or None, default=None] Input features.

- If *input_features* is *None*, then *feature_names_in_* is used as feature names in. If *feature_names_in_* is not defined, then names are generated: *[x0, x1, …, x(n_features_in_)]*.
- If *input_features* is an array-like, then *input_features* must match *feature_names_in_* if *feature_names_in_* is defined.

Returns

feature_names_out
[ndarray of str objects] Transformed feature names.

get_support(*indices=False*)

Get a mask, or integer index, of the features selected.

Parameters

indices
[bool, default=False] If True, the return value will be an array of integers, rather than a boolean mask.

Returns

support
[array] An index that selects the retained features from a feature vector. If *indices* is False, this is a boolean array of shape [# input features], in which an element is True iff its corresponding feature is selected for retention. If *indices* is True, this is an integer array of shape [# output features] whose values are indices into the input feature vector.

inverse_transform(*X*)

Reverse the transformation operation.

Parameters

X
[array of shape [n_samples, n_selected_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_original_features]] *X* with columns of zeros inserted where features would have been removed by *transform()*.

transform(*X*)

Reduce X to the selected features.

Parameters

X
[array of shape [n_samples, n_features]] The input samples.

Returns

X_r
[array of shape [n_samples, n_selected_features]] The input samples with only the selected features.

divik.feature_selection.**huberta_outliers**(*v*)

Outlier detection method based on medcouple statistic.

Parameters

v: array-like
An array to filter outlier from.

Returns

Binary vector indicating all the outliers.

References

M. Huberta, E.Vandervierenb (2008) An adjusted boxplot for skewed distributions, Computational Statistics and Data Analysis 52 (2008) 5186–5201

divik.feature_selection.**make_specialized_selector**(*name*, *n_features*, ***kwargs*)

Create a selector by name (`gmm`, `outlier`, `none` or `auto`)

`auto` switches to `gmm` if there is more than 250 features, `outlier` below.

2.4 divik.sampler module

Sampling methods for statistical indices computation purposes

class divik.sampler.**BaseSampler**

Base class for all the samplers

Sampler is Pool-safe, i.e. can simply store a dataset. It will not be serialized by pickle when going to another process, if handled properly.

Before you spawn a pool, a data must be moved to a module-level variable. To simplify that process a contract has been prepared. You open a context and operate within a context:

```
>>> with sampler.parallel() as sampler_,
...         Pool(initializer=sampler_.initializer,
...              initargs=sampler_.initargs) as pool:
...     pool.map(sampler_.get_sample, range(10))
```

Keep in mind, that __iter__ and fit are not accessible in parallel context. __iter__ would yield the same values independently in all the workers. Now it needs to be done consciously and in well-though manner. fit could lead to a non-predictable behaviour. If you need the original sampler, you can get a clone (not fit to the data).

Methods

fit(X[, y])	Fit sampler to data
get_params([deep])	Get parameters for this estimator.
get_sample(seed)	Return specific sample
parallel()	Create parallel context for the sampler to operate
set_params(**params)	Set the parameters of this estimator.

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

Fit sampler to data

It's a base for both supervised and unsupervised samplers.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`abstract get_sample(`*seed*`)`

Return specific sample

Following assumptions should be met: a) sampler.get_sample(x) == sampler.get_sample(x) b) x != y should yield sampler.get_sample(x) != sampler.get_sample(y)

Parameters

seed
[int] The seed to use to draw the sample

Returns

sample
[array_like, (**self.shape_)] Returns the drawn sample

`parallel()`

Create parallel context for the sampler to operate

`set_params(`***params*`)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`class divik.sampler.ParallelSampler(`*sampler*`)`

Helper class for sharing the sampler functionality

Attributes

initargs

Methods

clone()	Clones the original sampler
get_sample(seed)	Return specific sample

initializer	

`clone()`

Clones the original sampler

`get_sample(`*seed*`)`

Return specific sample

property initargs

initializer(*args)

class divik.sampler.StratifiedSampler(*n_rows=100*, *n_samples=None*)

Sample the original data preserving proportions of groups

Parameters

n_rows
[int or float, optional (default 10000)] Allows to limit the number of rows in the drawn samples. If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the sample. If int, represents the absolute number of rows.

n_samples
[int, optional (default None)] Allows to limit the number of samples when iterating

Attributes

X_
[array_like, shape (n_rows, n_features)] Data to sample from

y_
[array_like, shape (n_rows,)] Group labels

Methods

fit(X, y)	Fit the model from data in X.
get_params([deep])	Get parameters for this estimator.
get_sample(seed)	Return specific sample
parallel()	Create parallel context for the sampler to operate
set_params(**params)	Set the parameters of this estimator.

fit(*X*, *y*)

Fit the model from data in X.

Both inputs are preserved inside to sample from the data.

Parameters

X
[array-like, shape (n_rows, n_features)] Training vector, where n_rows is the number of rows and n_features is the number of features.

y: array-like, shape (n_rows,)

Returns

self
[StratifiedSampler] Returns the instance itself.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_sample`(*seed*)

Return specific sample

Sample is drawn from the set of existing rows. A proportion of gorups should be more-or-less the same, depending on the size of the sample.

Parameters

seed
[int] The seed to use to draw the sample

Returns

sample
[array_like, (*self.shape_)] Returns the drawn sample

`parallel`()

Create parallel context for the sampler to operate

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`class divik.sampler.`**`UniformPCASampler`**(*n_rows=None*, *n_samples=None*, *whiten=False*, *refit=False*, *pca='knee'*)

Rotation-invariant uniform sampling

Parameters

n_rows
[int, optional (default None)] Allows to limit the number of rows in the drawn samples

n_samples
[int, optional (default None)] Allows to limit the number of samples when iterating

whiten
[bool, optional (default False)] When True (False by default) the *pca_.components_* vectors are multiplied by the square root of n_samples and then divided by the singular values to ensure uncorrelated outputs with unit component-wise variances.

Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making their data respect some hard-wired assumptions.

refit
[bool, optional (default False)] When True (False by default) the *pca_* is re-fit with the smaller number of components. This could reduce memory footprint, but requires training fitting PCA.

pca: {'knee', 'full'}, default 'knee'
Specifies whether to train full or knee PCA.

Attributes

pca_
[KneePCA or PCA] PCA transform which provided rotation-invariance

sampler_
[UniformSampler] Sampler from the transformed distribution

Methods

fit(X[, y])	Fit the model from data in X.
get_params([deep])	Get parameters for this estimator.
get_sample(seed)	Return specific sample
parallel()	Create parallel context for the sampler to operate
set_params(**params)	Set the parameters of this estimator.

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

Fit the model from data in X.

PCA is fit to estimate the rotation and UniformSampler is fit to transformed data.

Parameters

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

Y: Ignored.

Returns

self
[UniformPCASampler] Returns the instance itself.

get_params(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

get_sample(*seed*)

Return specific sample

Sample is generated from transformed distribution and transformed back to the original space.

Parameters

seed
[int] The seed to use to draw the sample

Returns

sample
[array_like, (*self.shape_)] Returns the drawn sample

`parallel()`

Create parallel context for the sampler to operate

`set_params(**params)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`class divik.sampler.UniformSampler(n_rows=None, n_samples=None)`

Samples uniformly from the boundaries of the data

Parameters

n_rows
[int, optional (default None)] Allows to limit the number of rows in the drawn samples

n_samples
[int, optional (default None)] Allows to limit the number of samples when iterating

Attributes

shape_
[(n_rows, n_cols)] Shape of the drawn samples

scaler_
[MinMaxScaler] Scaler ensuring the proper ranges

Methods

fit(X[, y])	Fit the model from data in X.
get_params([deep])	Get parameters for this estimator.
get_sample(seed)	Return specific sample
parallel()	Create parallel context for the sampler to operate
set_params(**params)	Set the parameters of this estimator.

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

Fit the model from data in X.

Parameters

X
[array-like, shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features.

Y: Ignored.

Returns

self
[UniformSampler] Returns the instance itself.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`get_sample`(*seed*)

Return specific sample

Parameters

seed
[int] The seed to use to draw the sample

Returns

sample
[array_like, (**self.shape_*)] Returns the drawn sample

`parallel`()

Create parallel context for the sampler to operate

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

divik.cluster	Clustering methods
divik.feature_extraction	Unsupervised feature extraction methods
divik.feature_selection	Unsupervised feature selection methods
divik.sampler	Sampling methods for statistical indices computation purposes

CHAPTER

THREE

UTILITY PACKAGES

3.1 divik package

Unsupervised high-throughput data analysis methods

divik.**plot**(*tree*, *with_size=False*)

Plot visualization of splits.

divik.**reject_split**(*tree*, *rejection_size=0*)

Re-apply rejection condition on known result tree.

Return type
Optional[*DivikResult*]

Modules

divik.cluster	Clustering methods
divik.core	Reusable utilities used for building divik library
divik.feature_extraction	Unsupervised feature extraction methods
divik.feature_selection	Unsupervised feature selection methods
divik.sampler	Sampling methods for statistical indices computation purposes
divik.score	

3.2 divik.core module

Reusable utilities used for building divik library

divik.core.**Centroids**

alias of ndarray

divik.core.**Data**

alias of ndarray

class divik.core.**DivikResult**(*clustering: Union[*divik.cluster.GAPSearch, divik.cluster.DunnSearch*]*, *feature_selector:* divik.feature_selection.StatSelectorMixin, *merged: ndarray*, *subregions: List[Optional[*DivikResult*]]*)

Result of DiviK clustering

Attributes

clustering
Alias for field number 0

feature_selector
Alias for field number 1

merged
Alias for field number 2

subregions
Alias for field number 3

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

property clustering

Fitted automated clustering estimator

count(*value*, /)

Return number of occurrences of value.

property feature_selector

Fitted feature selector

index(*value*, *start=0*, *stop=sys.maxsize*, /)

Return first index of value.

Raises ValueError if the value is not present.

property merged

Recursively merged clustering labels

property subregions

DivikResults for all obtained subregions

divik.core.**IntLabels**

alias of ndarray

class divik.core.**Subsets**(*n_splits=10*, *random_state=42*)

Scatter dataset to disjoint random subsets and combine them back

Parameters

n_splits
[int, default 10] Number of subsets that will be generated.

random_state
[int, default 42] Random state to use for seeding the random number generator.

Examples

```
>>> from divik.core import Subsets
>>> subsets = Subsets(n_splits=10, random_state=42)
>>> X_list = subsets.scatter(X)
>>> len(X_list)
10
>>> # do some computations on each subset
>>> y = subsets.combine(y_list)
```

Methods

combine	
scatter	

combine(*X_list*)

scatter(*X*)

divik.core.**build**(*klass*, ***kwargs*)

Build instance of klass using matching kwargs

divik.core.**cached_fit**(*cls*)

Decorate a sklearn-compatible estimator to cache the fitting result

It is a wrapper over joblib.Memory.cache, that supports runtime cache path definition.

Set path definition through gin config with `cache_path.path` identifier.

divik.core.**configurable**(*name_or_fn=None*, *module=None*, *allowlist=None*, *denylist=None*)

Decorator to make a function or class configurable.

This decorator registers the decorated function/class as configurable, which allows its parameters to be supplied from the global configuration (i.e., set through *bind_parameter* or *parse_config*). The decorated function is associated with a name in the global configuration, which by default is simply the name of the function or class, but can be specified explicitly to avoid naming collisions or improve clarity.

If some parameters should not be configurable, they can be specified in *denylist*. If only a restricted set of parameters should be configurable, they can be specified in *allowlist*.

The decorator can be used without any parameters as follows:

@config.configurable def some_configurable_function(param1, param2='a default value'):

...

In this case, the function is associated with the name *'some_configurable_function'* in the global configuration, and both *param1* and *param2* are configurable.

The decorator can be supplied with parameters to specify the configurable name or supply an allowlist/denylist:

@config.configurable('explicit_configurable_name', allowlist='param2') def some_configurable_function(param1, param2='a default value'):

...

In this case, the configurable is associated with the name *'explicit_configurable_name'* in the global configuration, and only *param2* is configurable.

Classes can be decorated as well, in which case parameters of their constructors are made configurable:

@config.configurable class SomeClass:

def __init__(self, param1, param2='a default value'):
...

In this case, the name of the configurable is *'SomeClass'*, and both *param1* and *param2* are configurable.

Args:

name_or_fn: A name for this configurable, or a function to decorate (in
which case the name will be taken from that function). If not set, defaults to the name of the function/class that is being made configurable. If a name is provided, it may also include module components to be used for disambiguation (these will be appended to any components explicitly specified by *module*).

module: The module to associate with the configurable, to help handle naming
collisions. By default, the module of the function or class being made configurable will be used (if no module is specified as part of the name).

allowlist: An allowlisted set of kwargs that should be configurable. All
other kwargs will not be configurable. Only one of *allowlist* or *denylist* should be specified.

denylist: A denylisted set of kwargs that should not be configurable. All
other kwargs will be configurable. Only one of *allowlist* or *denylist* should be specified.

Returns:
When used with no parameters (or with a function/class supplied as the first parameter), it returns the decorated function or class. When used with parameters, it returns a function that can be applied to decorate the target function or class.

`divik.core.`**`context_if`**(*condition*, *context*, **args*, ***kwargs*)

Create context with given params only if the condition is `True`

`divik.core.`**`dump_gin_args`**(*destination*)

Dump gin-config effective configuration

If you have *gin* extras installed, you can call *dump_gin_args* save effective gin configuration to a file.

`divik.core.`**`get_n_jobs`**(*n_jobs*)

Determine the actual number of possible jobs

`divik.core.`**`maybe_pool`**(*processes=None*, **args*, ***kwargs*)

Create `multiprocessing.Pool` if multiple CPUs are allowed

Examples

```
>>> from divik.core import maybe_pool
>>> with maybe_pool(processes=1) as pool:
...     # Runs in sequential
...     pool.map(id, range(10000))
>>> with maybe_pool(processes=-1) as pool:
...     # Runs with all cores
...     pool.map(id, range(10000))
```

divik.core.**normalize_rows**(*data*)

Translate and scale rows to zero mean and vector length equal one

Return type
ndarray

divik.core.**parse_args**()

Parse gin config files and parameter overrides from command line

divik.core.**seed**(*seed_=0*)

Context manager that creates a seeded scope.

divik.core.**seeded**(*wrapped_requires_seed=False*)

Create seeded scope for function call.

Parameters

wrapped_requires_seed: bool, optional, default: False
if true, passes seed parameter to the inner function

divik.core.**share**(*array*)

Share a numpy array between `multiprocessing.Pool` processes

divik.core.**visualize**(*label*, *xy*, *shape=None*)

Create RGB map of labels over with given coordinates

Modules

divik.core.gin_sklearn_configurables	Mark scikit-learn classes as configurable
divik.core.io	Reusable utilities for data and model I/O

3.3 divik.core.io module

Reusable utilities for data and model I/O

divik.core.io.**load_data**(*path*)

Load 2D tabular data from file

Return type
ndarray

divik.core.io.**save**(*model*, *destination*, ***kwargs*)

Save model and related summaries into specified destination directory

divik.core.io.**save_csv**(*array*, *fname*)

Save array to csv

divik.core.io.**saver**(*fn*)

Register the function as handler for saving model and related summaries

The saver function should be reusable for different models exhibiting the required variables. Rather prefer checking the required attributes than the model class.

Examples

```
>>> from divik.core.io import saver
>>> @saver
... def my_saver(model, destination, **kwargs):
...     if not hasattr(model, 'my_custom_field_'):
...         return
...     if not 'my_param' in kwargs:
...         return
...     # custom saving logic comes here
```

You can also make this function configurable:

```
>>> import gin
>>> from divik.core.io import saver
>>> @saver
... @gin.configurable(allowlist=['my_param'])
... def configurable_saver(model, destination, my_param=None, **kwargs):
...     if not hasattr(model, 'my_custom_field_'):
...         return
...     if my_param is None:
...         return
...     # custom saving logic comes here
```

divik.core.io.**try_load_data**(*path*)

Load 2D tabular data from file with logging

divik.core.io.**try_load_xy**(*path*)

Load integer spatial coordinates with logging from file

3.4 divik.core.gin_sklearn_configurables module

Mark scikit-learn classes as configurable

divik	Unsupervised high-throughput data analysis methods
divik.core	Reusable utilities used for building divik library
divik.core.io	Reusable utilities for data and model I/O
divik.core.gin_sklearn_configurables	Mark scikit-learn classes as configurable

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

d

INDEX

G

R

S

T

U

V