# DiviK

***Release 2.1.8***

**Dec 08, 2019**

# Contents:

CHAPTER 1

# divik package

`divik.`**`seeded`**`(`*wrapped_requires_seed:bool=False*`)`

Create seeded scope for function call.

@param wrapped_requires_seed: if true, passes seed parameter to the inner function

**`class`** `divik.`**`DiviK`**`(`*gap_trials: int = 10*, *distance_percentile: float = 99.0*, *max_iter: int = 100*, *distance: str = 'correlation'*, *minimal_size: int = None*, *rejection_size: int = None*, *rejection_percentage: float = None*, *minimal_features_percentage: float = 0.01*, *fast_kmeans_iter: int = 10*, *k_max: int = 10*, *normalize_rows: bool = None*, *use_logfilters: bool = False*, *n_jobs: int = None*, *random_seed: int = 0*, *verbose: bool = False*`)`

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.ClusterMixin`, `sklearn.base.TransformerMixin`

DiviK clustering

**Parameters**

**gap_trials: int, optional, default: 10** The number of random dataset draws to estimate the GAP index for the clustering quality assessment.

**distance_percentile: float, optional, default: 99.0** The percentile of the distance between points and their closest centroid. 100.0 would simply select the furthest point from all the centroids found already. Lower value provides better robustness against outliers. Too low value reduces the capability to detect centroid candidates during initialization.

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

**distance: str, optional, default: 'correlation'** The distance metric between points, centroids and for GAP index estimation.

**minimal_size: int, optional, default: None** The minimum size of the region (the number of observations) to be considered for any further divisions. 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.

**fast_kmeans_iter: int, optional, default: 10** Maximum number of iterations of the k-means algorithm for a single run during computation of the GAP index. Decreased with respect to the max_iter, as GAP index requires multiple segmentations to be evaluated.

**k_max: int, optional, default: 10** Maximum number of clusters evaluated during the auto-tuning process. From 1 up to k_max clusters are tested per evaluation.

**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.

**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.

**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.

**random_seed: int, optional, default: 0** Seed to initialize the random number generator.

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

## Examples

```
>>> from divik import DiviK
>>> from sklearn.datasets import make_blobs
>>> X, _ = make_blobs(n_samples=200, n_features=100, centers=20,
...                   random_state=42)
>>> divik = DiviK(distance='euclidean').fit(X)
>>> 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.types.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`*(self, X[, y]) | Compute DiviK clustering. |
| *`fit_predict`*(self, X[, y]) | Compute cluster centers and predict cluster index for each sample. |
| *`fit_transform`*(self, X[, y]) | Compute clustering and transform X to cluster-distance space. |
| `get_params`(self[, deep]) | Get parameters for this estimator. |
| *`predict`*(self, X) | Predict the closest cluster each sample in X belongs to. |
| `set_params`(self, **params) | Set the parameters of this estimator. |
| *`transform`*(self, X) | Transform X to a cluster-distance space. |

**`fit`** (*self*, *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`** (*self*, *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**(*self*, *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, **n_clusters_**]] X transformed in the new space.

**predict**(*self*, *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.

**transform**(*self*, *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, **n_clusters_**]] X transformed in the new space.

**class** divik.**AutoKMeans**(*max_clusters: int*, *min_clusters: int = 1*, *n_jobs: int = 1*, *method: str = 'dunn'*, *distance: str = 'euclidean'*, *init: str = 'percentile'*, *percentile: float = 95.0*, *max_iter: int = 100*, *normalize_rows: bool = False*, *gap=None*, *verbose: bool = False*)

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.ClusterMixin`, `sklearn.base.TransformerMixin`

K-Means clustering with automated selection of number of clusters

**Parameters**

**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.

**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.

**method: {'dunn', 'gap'}** The method to select the best number of clusters.

'dunn' : computes score that relates dispersion inside a cluster to distances between clusters. Never selects 1 cluster.

'gap' : compares dispersion of a clustering to a dispersion in grouping of a reference uniformly distributed dataset

**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.

**init** [{'percentile' or 'extreme'}] 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

**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.

**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.

**gap** [dict] Configuration of GAP statistic in a form of dict.

**max_iter** [int, default: 10] Maximal number of iterations KMeans will do for computing statistic.

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

**trials** [int, default: 10] Number of data sets drawn as a reference.

**correction** [bool, default: True] If True, the correction is applied and the first feasible solution is selected. Otherwise the globally maximal GAP is used.

Default: {'max_iter': 10, 'seed': 0, 'trials': 10, 'correction': True}

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

**fit**(*self, X, y=None, pool:<bound method BaseContext.Pool of <multiprocessing.context.DefaultContext object at 0x7f40ab3e8d68>>=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.

**pool: Pool** used for parallelization of computations reusing single pool

**predict**(*self, 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.

**transform**(*self, 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.**KMeans**(*n_clusters: int, distance: str = 'euclidean', init: str = 'percentile', percentile: float = 95.0, max_iter: int = 100, normalize_rows: bool = False*)

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.ClusterMixin`, `sklearn.base.TransformerMixin`

K-Means clustering

**Parameters**

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

**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.

**init** [{'percentile' or 'extreme'}] 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

**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.

**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.

**Attributes**

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

**labels_ :** Labels of each point

## Methods

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

**`fit`** (*self*, *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.

**`predict`** (*self*, *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.

**transform** (*self*, *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.**GMMSelector** (*stat: str*, *use_log: bool = False*, *n_candidates: int = None*, *min_features: int = 1*, *min_features_rate: float = 0.0*, *preserve_high: bool = True*, *max_components: int = 10*)

Bases: `sklearn.base.BaseEstimator`, `sklearn.feature_selection.base.SelectorMixin`

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'}** 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.

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

**fit** (*self, 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**

**class** divik.**HighAbundanceAndVarianceSelector** (*use_log: bool = False, min_features: int = 1, min_features_rate: float = 0.0, max_components: int = 10*)
Bases: `sklearn.base.BaseEstimator`, `sklearn.feature_selection.base.SelectorMixin`

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

Continued on next page

Table 5 – continued from previous page

| | |
|---|---|
| `transform`(self, X) | Reduce X to the selected features. |

**fit**(*self*, *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**

`divik.`**depth**(*tree*, *children_collection_name='subregions'*)

Get tree depth.

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

Plot visualization of splits.

`divik.`**reject_split**(*tree:Union[divik._utils.DivikResult, NoneType], rejection_size:int=0*) → Union[divik._utils.DivikResult, NoneType]

Re-apply rejection condition on known result tree.

# CHAPTER 2

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index