scikit-hubness

Release 0.21.2

Roman Feldbauer

Sep 01, 2020

GETTING STARTED

`scikit-hubness` is a Python package for analysis of hubness in high-dimensional data. It provides hubness reduction and approximate nearest neighbor search via a drop-in replacement for sklearn.neighbors.

CHAPTER

ONE

INSTALLATION

1.1 Installation from PyPI

The current release of `scikit-hubness` can be installed from PyPI:

```
pip install scikit-hubness
```

1.2 Dependencies

All strict dependencies of `scikit-hubness` are automatically installed by `pip`. Some optional dependencies (certain ANN libraries) may not yet be available from PyPI. If you require one of these libraries, please refer to the library's documentation for building instructions. For example, at the time of writing, `puffinn` was not available on PyPI. Building and installing is straight-forward:

```
git clone https://github.com/puffinn/puffinn.git
cd puffinn
python3 setup.py build
pip install .
```

1.3 Installation from source

You can always grab the latest version of `scikit-hubness` directly from GitHub:

```
cd install_dir
git clone git@github.com:VarIr/scikit-hubness.git
cd scikit-hubness
pip install -e .
```

This is the recommended approach, if you want to contribute to the development of `scikit-hubness`.

1.4 Supported platforms

`scikit-hubness` currently supports all major operating systems:

- Linux
- MacOS X
- Windows

Note, that not all approximate nearest neighbor algorithms used in `scikit-hubness` are available on all platforms. This is because we rely on third-party libraries, which in some cases are not available for all platforms. The table below indicates, which libraries and algorithms are currently supported on your operating system. All exact nearest neighbor algorithms (as provided by scikit-learn) are available on all platforms.

library	algorithm	Linux	MacOS	Windows
nmslib	hnsw	x	x	x
annoy	rptree	x	x	x
ngtpy	nng	x	x	
falconn	falconn_lsh	x	x	
puffinn	lsh	x	x	
sklearn	(all exact)	x	x	x

CHAPTER

TWO

QUICK START EXAMPLE

Users of `scikit-hubness` typically want to

1. analyse, whether their data show hubness
2. reduce hubness
3. perform learning (classification, regression, ...)

The following example shows all these steps for an example dataset from the text domain (dexter). Please make sure you have installed `scikit-hubness` (installation instructions).

First, we load the dataset and inspect its size.

```
from skhubness.data import load_dexter
X, y = load_dexter()
print(f'X.shape = {X.shape}, y.shape={y.shape}')
```

Dexter is embedded in a high-dimensional space, and could, thus, be prone to hubness. Therefore, we assess the actual degree of hubness.

```
from skhubness import Hubness
hub = Hubness(k=10, metric='cosine')
hub.fit(X)
k_skew = hub.score()
print(f'Skewness = {k_skew:.3f}')
```

As a rule-of-thumb, skewness > 1.2 indicates significant hubness. Additional hubness indices are available, for example:

```
print(f'Robin hood index: {hub.robinhood_index:.3f}')
print(f'Antihub occurrence: {hub.antihub_occurrence:.3f}')
print(f'Hub occurrence: {hub.hub_occurrence:.3f}')
```

There is considerable hubness in dexter. Let's see, whether hubness reduction can improve kNN classification performance.

```
from sklearn.model_selection import cross_val_score
from skhubness.neighbors import KNeighborsClassifier

# vanilla kNN
knn_standard = KNeighborsClassifier(n_neighbors=5,
                                    metric='cosine')
acc_standard = cross_val_score(knn_standard, X, y, cv=5)

# kNN with hubness reduction (mutual proximity)
```

(continues on next page)

(continued from previous page)

```
knn_mp = KNeighborsClassifier(n_neighbors=5,
                              metric='cosine',
                              hubness='mutual_proximity')
acc_mp = cross_val_score(knn_mp, X, y, cv=5)

print(f'Accuracy (vanilla kNN): {acc_standard.mean():.3f}')
print(f'Accuracy (kNN with hubness reduction): {acc_mp.mean():.3f}')
```

Accuracy was considerably improved by mutual proximity (MP). But did MP actually reduce hubness?

```
hub_mp = Hubness(k=10, metric='cosine',
                 hubness='mutual_proximity')
hub_mp.fit(X)
k_skew_mp = hub_mp.score()
print(f'Skewness after MP: {k_skew_mp:.3f} '
      f'(reduction of {k_skew - k_skew_mp:.3f})')
print(f'Robin hood: {hub_mp.robinhood_index:.3f} '
      f'(reduction of {hub.robinhood_index - hub_mp.robinhood_index:.3f})')
```

Yes!

The neighbor graph can also be created directly, with or without hubness reduction:

```
from skhubness.neighbors import kneighbors_graph
neighbor_graph = kneighbors_graph(X,
                                  n_neighbors=5,
                                  hubness='mutual_proximity')
```

You may want to precompute the graph like this, in order to avoid computing it repeatedly for subsequent hubness estimation and learning.

CHAPTER

THREE

USER GUIDE

Welcome to `scikit-hubness`! Here we describe the core functionality of the package (hubness analysis, hubness reduction, neighbor search), and provide several usage examples.

3.1 Core Concepts

There are three main parts of `scikit-hubness`. Before we describe the corresponding subpackages, we briefly introduce the *hubness* phenomenon itself.

3.1.1 The hubness phenomenon

Hubness is a phenomenon of intrinsically high-dimensional data, detrimental to data mining and learning tasks. It refers to the tendency of *hub* and *antihub* emergence in k-nearest neighbor graphs (kNNGs): Hubs are objects that appear unwontedly often among the k-nearest neighbor lists of other objects, while antihubs hardly or never appear in these lists. Thus, hubs propagate their metainformation (such as class labels) widely within a kNNG. Conversely, information carried by antihubs is effectively lost. As a result, hubness leads to semantically distorted spaces, that negatively impact a large variety of tasks.

Music information retrieval is a show-case example for hubness: It has been shown, that recommendation lists based on music similarity scores tend to completely ignore certain songs (*antihubs*). On the other hand, different songs are recommended over and over again (*hubs*), sometimes even when they do not fit. Both effects are problematic: Users are provided with unsuitable (hub) recommendations, while artists that (unknowingly) producing antihub songs, may remain fameless unjustifiably.

3.1.2 The scikit-hubness package

`scikit-hubness` reflects our effort to make hubness analysis and hubness reduction readily available and easy-to-use for both machine learning researchers and practitioners.

The package builds upon `scikit-learn`. When feasible, their design decisions, code style, development practise etc. are adopted, so that new users can work their way into `scikit-hubness` rapidly. Workflows, therefore, comprise the well-known `fit`, `predict`, and `score` methods.

Two subpackages offer complementary functionality to `scikit-learn`:

- *`skhubness.analysis`* allows to estimate hubness in data
- *`skhubness.reduction`* provides hubness reduction algorithms

The *`skhubness.neighbors`* subpackage, on the other hand, acts as a drop-in replacement for `sklearn.neighbors`. It provides all of its functionality, and adds two major components:

- transparent hubness reduction
- approximate nearest neighbor (ANN) search

and combinations of both. From the coding point-of-view, this is achieved by adding a handful new parameters to most classes (*KNeighborsClassifier*, *RadiusNeighborRegressor*, *NearestNeighbors*, etc).

- `hubness` defines the hubness reduction algorithm used to compute the nearest neighbor graph (kNNG). Supported algorithms and corresponding parameter values are presented *here*, and are available as a Python list in `<skhubness.reduction.hubness_algorithms>`.
- `algorithm` defines the kNNG construction algorithm similarly to the way `sklearn` does it. That is, all of `sklearn`'s algorithms are available, but in addition, several approximate nearest neighbor algorithms are provided as well. *See below* for a list of currently supported algorithms and their corresponding parameter values.

By providing the two arguments above, you select algorithms for hubness reduction and nearest neighbor search, respectively. Most of these algorithms can be further tuned by individual hyperparameters. These are not explicitly made accessible in high-level classes like `KNeighborsClassifier`, in order to avoid very long lists of arguments, because they differ from algorithm to algorithm. Instead, two dictionaries

- `hubness_params` and
- `algorithm_params`

are available in all high-level classes to set the nested arguments for ANN and hubness reduction methods.

The following example shows how to perform approximate hubness estimation (1) without, and (2) with hubness reduction by local scaling in an artificial data set.

In part 1, we estimate hubness in the original data.

```
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1_000_000,
                           n_features=500,
                           n_informative=400,
                           random_state=123)

from sklearn.model_selection import train_test_split
X_train, X_test = train_test_split(X, test_size=0.1, random_state=456)

from skhubness.analysis import Hubness
hub = Hubness(k=10,
                    metric='euclidean',
                    algorithm='hnsw',
                    algorithm_params={'n_candidates': 100,
                                      'metric': 'euclidean',
                                      'post_processing': 2,
                                      },
                    return_value='robinhood',
                    n_jobs=8,
                    )
hub.fit(X_train)
robin_hood = hub.score(X_test)
print(robin_hood)
0.7873205555555555  # before hubness reduction
```

There is high hubness in this dataset. In part 2, we estimate hubness after reduction by local scaling.

```
hub = Hubness(k=10,
              metric='euclidean',
```

(continues on next page)

(continued from previous page)

```
                hubness='local_scaling',
                hubness_params={'k': 7},
                algorithm='hnsw',
                algorithm_params={'n_candidates': 100,
                                  'metric': 'euclidean',
                                  'post_processing': 2,
                                 },
                return_value='robinhood',
                verbose=2
                )
hub.fit(X_train)
robin_hood = hub.score(X_test)
print(robin_hood)
0.6614583333333331  # after hubness reduction
```

3.1.3 Approximate nearest neighbor search methods

Set the parameter `algorithm` to one of the following in order to enable ANN in most of the classes from *skhubness.neighbors* or *Hubness*:

- 'hnsw' uses *hierarchical navigable small-world graphs* (provided by the `nmslib` library) in the wrapper class *HNSW*.
- 'lsh' uses *locality sensitive hashing* (provided by the `puffinn` library) in the wrapper class *PuffinnLSH*.
- 'falconn_lsh' uses *locality sensitive hashing* (provided by the `falconn` library) in the wrapper class *FalconnLSH*.
- 'nng' uses ANNG or ONNG (provided by the `NGT` library) in the wrapper class *NNG*.
- 'rptree' uses random projections trees (provided by the `annoy` library) in the wrapper class *RandomProjectionTree*.

Configure parameters of the chosen algorithm with `algorithm_params`. This dictionary is passed to the corresponding wrapper class. Take a look at their documentation in order to see, which parameters are available for each individual class.

3.1.4 Hubness reduction methods

Set the parameter `hubness` to one of the following identifiers in order to use the corresponding hubness reduction algorithm:

- 'mp' or 'mutual_proximity' use *mutual proximity* (Gaussian or empiric distribution) as implemented in *MutualProximity*.
- 'ls' or 'local_scaling' use *local scaling* or *NICDM* as implemented in *LocalScaling*.
- 'dsl' or 'dis_sim_local' use *DisSim Local* as implemented in *DisSimLocal*.

Variants and additional parameters are set with the `hubness_params` dictionary. Have a look at the individual hubness reduction classes for available parameters.

3.1.5 Approximate hubness reduction

Exact hubness reduction scales at least quadratically with the number of samples. To reduce computational complexity, *approximate* hubness reduction can be applied, as described in the paper "Fast approximate hubness reduction for large high-dimensional data" (ICBK2018, on IEEE Xplore, also available as technical report).

The general idea behind approximate hubness reduction works as follows:

1. retrieve `n_candidates`-nearest neighbors using an ANN method
2. refine and reorder the candidate list by hubness reduction
3. return `n_neighbors` nearest neighbors from the reordered candidate list

The procedure is implemented in scikit-hubness by simply passing both `algorithm` and `hubness` parameters to the relevant classes.

Also consider passing `algorithm_params={'n_candidates': n_candidates}`. Make sure to set the `n_candidates` high enough, for high sensitivity (towards "good" nearest neighbors). Too large values may, however, lead to long query times. As a rule of thumb for this trade-off, you can start by retrieving ten times as many candidates as you need nearest neighbors.

3.2 Hubness analysis

You can use the *skhubness.analysis* subpackage to assess whether your data is prone to hubness. Currently, the *Hubness* class acts as a one-stop-shop for hubness estimation. It provides several hubness measures, that are all computed from a nearest neighbor graph (kNNG). More specifically, hubness is measured from *k-occurrence*, that is, how often does an object occur in the k-nearest neighbor lists of other objects (reverse nearest neighbors). Traditionally, hubness has been measured by the skewness of the k-occurrence histogram, where higher skewness to the right indicates higher hubness (due to objects that appear very often as nearest neighbors). Recently, additional indices borrowed from inequality research have been proposed for measuring hubness, such as calculating the Robin Hood index or Gini index from k-occurrences, which may have more desirable features w.r.t to large datasets and interpretability.

The *Hubness* class provides a variety of these measures. It is based on scikit-learn's `BaseEstimator`, and thus follows scikit-learn principles. When a new instance is created, sensible default parameters are used, unless specific choices are made. Typically, the user may want to choose a parameter `k` to define the size of nearest neighbor lists, or `metric`, in case the default Euclidean distances do not fit the data well. Parameter `return_value` defines which hubness measures to use. *VALID_HUBNESS_MEASURES* provides a list of available measures. If `return_values=='all'`, all available measures are computed. The `algorithm` parameter defines how to compute the kNN graph. This is especially relevant for large datasets, as it provides more efficient index structures and approximate nearest neighbor algorithms. For example, `algorithm='hnsw'` uses a hierarchical navigable small-world graph to compute the hubness measures in log-linear time (instead of quadratic).

Hubness uses *fit* and *score* methods to estimate hubness. In this fictional example, we estimate hubness in terms of the Robin Hood index in some large dataset:

```
>>> X = (some large dataset)
>>> hub = Hubness(k=10,
>>>               return_value='robinhood',
>>>               algorithm='hnsw')
>>> hub.fit(X)  # Creates the HNSW index
>>> hub.score()
0.56
```

A Robin Hood index of 0.56 indicates, that 56% of all slots in nearest neighbor lists would need to be redistributed, in order to obtain equal k-occurrence for all objects. We'd consider this rather high hubness.

In order to evaluate, whether hubness reduction might be beneficial for downstream tasks (learning etc.), we can perform the same estimation with hubness reduction enabled. We use the same code as above, but add the `hubness` parameter:

```
>>> X = (some large dataset)
>>> hub = Hubness(k=10,
>>>               return_value='robinhood',
>>>               algorithm='hnsw',
>>>               hubness='local_scaling')
>>> hub.fit(X)
>>> hub.score()
0.35
```

Here, the hubness reduction method *local scaling* resulted in a markedly lower Robin Hood index.

Note, that we used the complete data set `X` in the examples above. We can also split the data into some `X_train` and `X_test`:

```
>>> hub.fit(X_train)
>>> hub.score(X_test)
0.36
```

This is useful, when you want to tune hyperparameters towards low hubness, and prevent data leakage.

3.2.1 Hubness measures

The degree of hubness in a dataset typically measured from its k-occurrence histogram O^k. For an individual data object **x**, its k-occurrence $O^k(x)$ is defined as the number of times **x** resides among the *k*-nearest neighbors of all other objects in the data set. In the notion of network analysis, $O^k(x)$ is the indegree of **x** in a directed kNN graph. It is also known as reverse neighbor count.

The following measures are provided in *`Hubness`* by passing the corresponding argument values (e.g. `hubness='robinhood'`):

- 'k_skewness': Skewness, the third central moment of the k-occurrence distribution, as introduced by Radovanović et al. 2010
- 'k_skewness_truncnorm': skewness of truncated normal distribution estimated from k-occurrence distribution.
- 'atkinson': the Atkinson index of inequality, which can be tuned in order to be more sensitive towards antihub or hubs.
- 'gini': the Gini coefficient of inequality, defined as the half of the relative mean absolute difference
- 'robinhood': the Robin Hood or Hoover index, which gives the amount that needs to be redistributed in order to obtain equality (e.g. proportion of total income, so that there is equal income for all; or the number of nearest neighbor slot, so that all objects are among the k-nearest neighbors of others exactly k times).
- 'antihubs': returns the indices of antihubs in data set **X** (which are never among the nearest neighbors of other objects.
- 'antihub_occurrence': proportion of antihubs in the data set (percentage of total objects, which are antihubs).
- 'hubs': indices of hub objects **x** in data set **X** (with $O^k(x) >$

3.3 Hubness reduction

The *skhubness.reduction* subpackage provides several hubness reduction methods. Currently, the supported methods are

- Mutual proximity (independent Gaussian distance distribution), provided by *MutualProximity* with `method='normal'` (default),
- Mutual proximity (empiric distance distribution), provided by *MutualProximity* with `method='empiric'`,
- Local scaling, provided by *LocalScaling* with `method='standard'` (default),
- Non-iterative contextual dissimilarity measure, provided by *LocalScaling* with `method='nicdm'`,
- DisSim Local, provided by *DisSimLocal*,

which represent the most successful hubness reduction methods as identified in our paper "A comprehensive empirical comparison of hubness reduction in high-dimensional spaces", KAIS (2019), DOI. This survey paper also comes with an overview of how the individual methods work.

There are two ways to use perform hubness reduction in scikit-hubness:

- Implicitly, using the classes in *skhubness.neighbors* (see *User Guide: Nearest neighbors*),
- Explicitly, using the classes in *skhubness.reduction*.

The former is the common approach, if you simply want to improve your learning task by hubness reduction. Most examples here also do so. The latter may, however, be more useful for researchers, who would like to investigate the hubness phenomenon itself.

All hubness reducers inherit from a common base class `HubnessReduction`. This abstract class defines two important methods: `fit` and `transform`, thus allowing to transform previously unseen data after the initial fit. Most hubness reduction methods do not operate on vector data, but manipulate pre-computed distances, in order to obtain *secondary distances*. Therefore, `fit` and `transform` take neighbor graphs as input, instead of vectors. Have a look at their signatures:

```
@abstractmethod
def fit(self, neigh_dist, neigh_ind, X, assume_sorted, *args, **kwargs):
    pass  # pragma: no cover

@abstractmethod
def transform(self, neigh_dist, neigh_ind, X, assume_sorted, return_distance=True):
    pass  # pragma: no cover
```

The arguments `neigh_dist` and `neigh_ind` are two arrays representing the nearest neighbor graph with shape `(n_indexed, n_neighbors)` during fit, and shape `(n_query, n_neighbors)` during transform. The i-th row in each array corresponds to the i-th object in the data set. The j-th column in `neigh_ind` contains the index of one of the k-nearest neighbors among the indexed objects, while the j-th column in `neigh_dist` contains the corresponding distance. Note, that this is the same format as obtained by scikit-learn's `kneighbors(return_distances=True)` method.

This way, the user has full flexibility on how to calculate primary distances (Euclidean, cosine, KL divergence, etc). *DisSimLocal* (DSL) is the exception to this rule, because it is formulated specifically for Euclidean distances. DSL, therefore, also requires the training vectors in `fit(..., X=X_train)`, and the test set vectors in `transform(..., X=X_test)`. Argument `X` is ignored in the other hubness reduction methods.

When the neighbor graph is already sorted (lowest to highest distance), `assume_sorted=True` should be set, so that hubness reduction methods will not sort the arrays again, thus saving computational time.

Hubness reduction methods transform the primary distance graph, and return secondary distances. Note that for efficiency reasons, the returned arrays are not sorted. Please make sure to sort the arrays, if downstream tasks assume sorted arrays.

3.4 Nearest neighbors

The *skhubness.neighbors* subpackage provides several neighbors-based learning methods. It is designed as a drop-in replacement for scikit-learn's `neighbors`. The package provides all functionality from `sklearn.neighbors`, and adds support for transparent hubness reduction, where applicable, including

- classification (e.g. *KNeighborsClassifier*),
- regression (e.g. *RadiusNeighborsRegressor*),
- unsupervised learning (e.g. *NearestNeighbors*),
- outlier detection (*LocalOutlierFactor*), and
- kNN graphs (*kneighbors_graph*).

In addition, scikit-hubness provides approximate nearest neighbor (ANN) search, in order to support large data sets with millions of data objects and more. A list of currently provided ANN methods is available *here*.

Hubness reduction and ANN search can be used independently or in conjunction, the latter yielding *approximate hubness reduction*. User of scikit-learn will find that only minor modification of their code is required to enable one or both of the above. We describe how to do so *here*.

For general information and details about nearest neighbors, we refer to the excellent scikit-learn User Guide on Nearest Neighbors.

3.5 Examples

In this section, we provide usage examples for `skhubness`.

3.5.1 Example: Hubness reduction

These examples show how to perform hubness reduction in kNN classification in (nested) cross-validation and pipelines.

Example: skhubness in Pipelines

Estimators from scikit-hubness can - of course - be used in a scikit-learn `Pipeline`. In this example, we select the best hubness reduction method and several other hyperparameters in grid search w.r.t. to classification performance.

```
from sklearn.datasets import make_classification
from sklearn.decomposition import PCA
from sklearn.model_selection import StratifiedKFold, train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

from skhubness.neighbors import KNeighborsClassifier

# Not so high-dimensional data
```

(continues on next page)

(continued from previous page)

```
X, y = make_classification(n_samples=1_000,
                           n_features=50,
                           n_informative=20,
                           n_classes=2,
                           random_state=3453)

X, X_test, y, y_test = train_test_split(X, y,
                                        test_size=100,
                                        stratify=y,
                                        shuffle=True,
                                        random_state=124)

# Pipeline of standardization, dimensionality reduction, and kNN classification
pipe = Pipeline([('scale', StandardScaler(with_mean=True, with_std=True)),
                 ('pca', PCA(n_components=20, random_state=1213)),
                 ('knn', KNeighborsClassifier(n_neighbors=10, algorithm='lsh',␣
↪hubness='mp'))])

# Exhaustive search for best algorithms and hyperparameters
param_grid = {'pca__n_components': [10, 20, 30],
              'knn__n_neighbors': [5, 10, 20],
              'knn__algorithm': ['auto', 'hnsw', 'lsh', 'falconn_lsh', 'nng', 'rptree
↪'],
              'knn__hubness': [None, 'mp', 'ls', 'dsl']}
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1354)
search = GridSearchCV(pipe, param_grid, n_jobs=5, cv=cv, verbose=1)
search.fit(X, y)

# Performance on hold-out data
acc = search.score(y_test, y_test)
print(acc)
# 0.79

print(search.best_params_)
# {'knn__algorithm': 'auto',
#  'knn__hubness': 'dsl',
#  'knn__n_neighbors': 20,
#  'pca__n_components': 30}
```

Total running time of the script: (0 minutes 0.000 seconds)

Face recognition (Olivetti faces)

This dataset contains a set of face images taken between April 1992 and April 1994 at AT&T Laboratories Cambridge. Image data is typically embedded in very high-dimensional spaces, which might be prone to hubness.

```
import numpy as np
from sklearn.datasets import olivetti_faces
from sklearn.model_selection import cross_val_score, StratifiedKFold,␣
↪RandomizedSearchCV

from skhubness import Hubness
from skhubness.neighbors import KNeighborsClassifier

# Fetch data and have a look
```

(continues on next page)

(continued from previous page)

```
d = olivetti_faces.fetch_olivetti_faces()
X, y = d['data'], d['target']
print(f'Data shape: {X.shape}')
print(f'Label shape: {y.shape}')
# (400, 4096)
# (400,)

# The data is embedded in a high-dimensional space.
# Is there hubness, and can we reduce it?
for hubness in [None, 'dsl', 'ls', 'mp']:
    hub = Hubness(k=10, hubness=hubness, return_value='k_skewness')
    hub.fit(X)
    score = hub.score()
    print(f'Hubness (10-skew): {score:.3f} with hubness reduction: {hubness}')
# Hubness (10-skew): 1.972 with hubness reduction: None
# Hubness (10-skew): 1.526 with hubness reduction: dsl
# Hubness (10-skew): 0.943 with hubness reduction: ls
# Hubness (10-skew): 0.184 with hubness reduction: mp

# There is some hubness, and all hubness reduction methods can reduce it (to varying␣
↪degree)
# Let's assess the best kNN strategy and its estimated performance.
cv_perf = StratifiedKFold(n_splits=5, shuffle=True, random_state=7263)
cv_select = StratifiedKFold(n_splits=5, shuffle=True, random_state=32634)

knn = KNeighborsClassifier(algorithm_params={'n_candidates': 100})

# specify parameters and distributions to sample from
param_dist = {"n_neighbors": np.arange(1, 26),
              "weights": ['uniform', 'distance'],
              "hubness": [None, 'dsl', 'ls', 'mp']}

# Inner cross-validation to select best hyperparameters (incl hubness reduction␣
↪method)
search = RandomizedSearchCV(estimator=knn,
                            param_distributions=param_dist,
                            n_iter=100,
                            cv=cv_select,
                            random_state=2345,
                            verbose=1)

# Outer cross-validation to estimate performance
score = cross_val_score(search, X, y, cv=cv_perf, verbose=1)
print(f'Scores: {score}')
print(f'Mean acc = {score.mean():.3f} +/- {score.std():.3f}')

# Select model that maximizes accuracy
search.fit(X, y)

# The best model's parameters
print(search.best_params_)

# Does it correspond to the results of hubness reduction above?
# Scores: [0.95   0.9625 1.     0.95   0.925 ]
# Mean acc = 0.957 +/- 0.024
# {'weights': 'distance', 'n_neighbors': 23, 'hubness': 'mp'}
```

Total running time of the script: (0 minutes 0.000 seconds)

3.5.2 Example: Approximate nearest neighbor search

This example shows how to perform approximate nearest neighbor search.

Retrieving GLOVE word vectors

In this example we will retrieve similar words from GLOVE embeddings with an ANNG graph.

Precomputed ground-truth nearest neighbors are available from ANN benchmarks.

```
# For this example, the `h5py` package is required in addition to the requirements of␣
↪scikit-hubness.
# You may install it from PyPI by the following command (if you're in an IPython/
↪Jupyter environment):
# !pip install h5py

import numpy as np
import h5py
from skhubness.neighbors import NearestNeighbors

# Download the dataset with the following command.
# If the dataset is already available in the current working dir, you can skip this:
# !wget http://ann-benchmarks.com/glove-100-angular.hdf5
f = h5py.File('glove-100-angular.hdf5', 'r')

# Extract the split and ground-truth
X_train = f['train']
X_test = f['test']
neigh_true = f['neighbors']
dist = f['distances']

# How many object have we got?
for k in f.keys():
    print(f'{k}: shape = {f[k].shape}')

# APPROXIMATE NEAREST NEIGHBOR SEARCH
# In order to retrieve most similar words from the GLOVE embeddings,
# we use the unsupervised `skhubness.neighbors.NearestNeighbors` class.
# The (approximate) nearest neighbor algorithm is set to NNG by passing `algorithm=
↪'nng'`.
# We can pass additional parameters to `NNG` via the `algorithm_params` dict.
# Here we set `n_jobs=8` to enable parallelism.
# Create the nearest neighbor index
nn_plain = NearestNeighbors(n_neighbors=100,
                            algorithm='nng',
                            algorithm_params={'n_candidates': 1_000,
                                              'index_dir': 'auto',
                                              'n_jobs': 8},
                            verbose=2,
                            )
nn_plain.fit(X_train)

# Note that NNG must save its index. By setting `index_dir='auto'`,
# NNG will try to save it to shared memory, if available, otherwise to $TMP.
```

(continues on next page)

(continued from previous page)

```
# This index is NOT removed automatically, as one will typically want build an index␣
↪once and use it often.
# Retrieve nearest neighbors for each test object
neigh_pred_plain = nn_plain.kneighbors(X_test,
                                       n_neighbors=100,
                                       return_distance=False)

# Calculate the recall per test object
recalled_plain = [np.intersect1d(neigh_true[i], neigh_pred_plain)
                  for i in range(len(X_test))]
recall_plain = np.array([recalled_plain[i].size / neigh_true.shape[1]
                         for i in range(len(X_test))])

# Statistics
print(f'Mean = {recall_plain.mean():.4f}, '
      f'stdev = {recall_plain.std():.4f}')


# ANN with HUBNESS REDUCTION
# Here we set `n_candidates=1000`, so that for each query,
# 1000 neighbors will be retrieved first by `NNG`,
# that are subsequently refined by hubness reduction.
# Hubness reduction is performed by local scaling as specified with `hubness='ls'`.
# Creating the NN index with hubness reduction enabled
nn = NearestNeighbors(n_neighbors=100,
                      algorithm='nng',
                      algorithm_params={'n_candidates': 1_000,
                                        'n_jobs': 8},
                      hubness='ls',
                      verbose=2,
                      )
nn.fit(X_train)

# Retrieve nearest neighbors for each test object
neigh_pred = nn.kneighbors(X_test,
                           n_neighbors=100,
                           return_distance=False)

# Measure recall per object and on average
recalled = [np.intersect1d(neigh_true[i], neigh_pred)
            for i in range(len(X_test))]
recall = np.array([recalled[i].size / neigh_true.shape[1]
                   for i in range(len(X_test))])
print(f'Mean = {recall.mean():.4f}, '
      f'stdev = {recall.std():.4f}')

# If the second results are significantly better than the first,
# this could indicate that the chosen ANN method is more prone
# to hubness than exact NN, which might be an interesting research question.
```

Total running time of the script: (0 minutes 0.000 seconds)

3.5.3 Example: Approximate hubness reduction

These examples show how to combine approximate nearest neighbor search and hubness reduction.

Example: Reusing index structures

This example shows how to reuse index structures. If you want to first estimate hubness, and then perform kNN, you can avoid recomputing the ANN index structure, which can be costly.

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

from skhubness.analysis import Hubness
from skhubness.neighbors import KNeighborsClassifier

X, y = make_classification(n_samples=100_000,
                           n_features=500,
                           n_informative=400,
                           random_state=543)

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.01,
                                                    stratify=y,
                                                    shuffle=True,
                                                    random_state=2346)

# Approximate hubness estimation: Creates LSH index and computes local scaling factors
hub = Hubness(k=10,
              return_value='robinhood',
              algorithm='falconn_lsh',
              hubness='ls',
              random_state=2345,
              shuffle_equal=False,
              verbose=1)
hub.fit(X_train)

robin_hood = hub.score(X_test)
print(f'Hubness (Robin Hood): {robin_hood}:.4f')
# 0.9060

# Approximate hubness reduction for classification: Reuse index & factors
knn = KNeighborsClassifier(n_neighbor=10,
                           algorithm='falconn_lsh',
                           hubness='ls',
                           n_jobs=1)

knn.fit(hub.nn_index_, y_train)  # REUSE INDEX HERE
acc = knn.score(X_test, y_test)
print(f'Test accuracy: {acc:.3f}')
# 0.959
```

Total running time of the script: (0 minutes 0.000 seconds)

Example: Approximate hubness reduction

This example shows how to combine approximate nearest neighbor search and hubness reduction in order to perform approximate hubness reduction for large data sets.

```
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

from skhubness.analysis import Hubness
from skhubness.neighbors import KNeighborsClassifier

# High-dimensional artificial data
X, y = make_classification(n_samples=1_000_000,
                           n_features=500,
                           n_informative=400,
                           random_state=543)

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=10_000,
                                                    stratify=y,
                                                    shuffle=True,
                                                    random_state=2346)

# Approximate hubness estimation
hub = Hubness(k=10,
              return_value='robinhood',
              algorithm='hnsw',
              random_state=2345,
              shuffle_equal=False,
              n_jobs=-1,
              verbose=2)
hub.fit(X_train)
robin_hood = hub.score(X_test)
print(f'Hubness (Robin Hood): {robin_hood:.3f}')
# 0.944

# Approximate hubness reduction for classification
knn = KNeighborsClassifier(n_neighbor=10,
                           algorithm='hnsw',
                           hubness='ls',
                           n_jobs=-1,
                           verbose=2)

knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
acc = accuracy_score(y_test, y_pred)
print(f'Test accuracy: {acc:.3f}')
# Test accuracy: 0.987
```

Total running time of the script: (0 minutes 0.000 seconds)

3.5.4 scikit-learn examples adapted for scikit-hubness

Examples concerning using *skhubness.neighbors* as drop-in replacement for `sklearn.neighbors`.

These examples are taken from scikit-learn and demonstrate the ease of transition from `sklearn.neighbors` to `skhubness.neighbors`. You will find that many examples require no more than modifying an import line, and/or adding one argument when instantiating an estimator.

Note, that these examples are not intended to demonstrate improved learning performance due to hubness reduction (the data are rather low-dimensional).

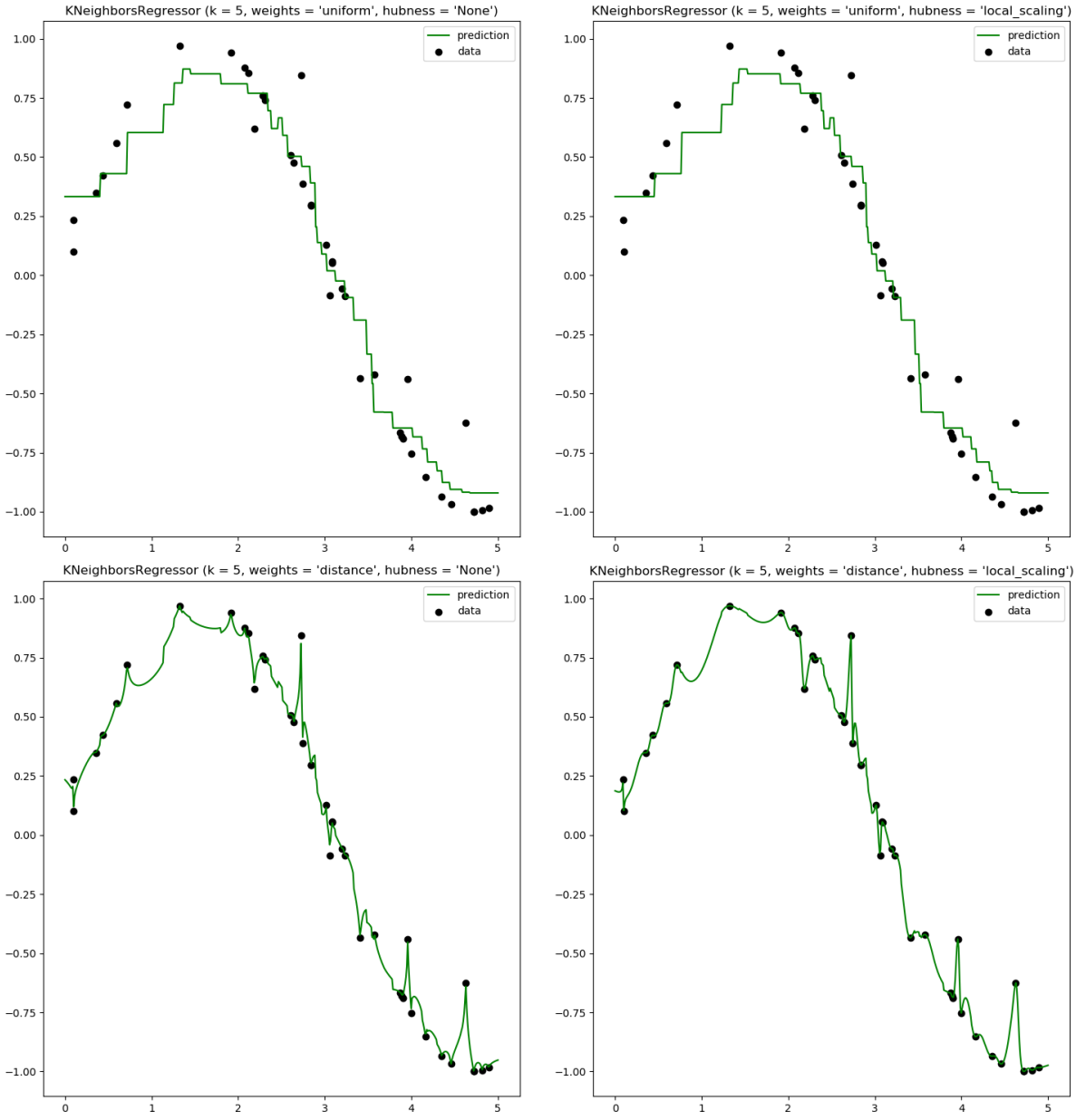
Note: Click *here* to download the full example code

Nearest Neighbors regression

Demonstrate the resolution of a regression problem using a k-Nearest Neighbor and the interpolation of the target using both barycenter and constant weights.

Hubness reduction of this low-dimensional dataset shows only small effects.

Adapted from https://scikit-learn.org/stable/auto_examples/neighbors/plot_regression.html



Out:

```
/home/user/feldbauer/PycharmProjects/hubness/examples/sklearn/plot_regression.py:60:␣
↪UserWarning: Matplotlib is currently using agg, which is a non-GUI backend, so␣
↪cannot show the figure.
  plt.show()
```

```
print(__doc__)
```

(continues on next page)

(continued from previous page)

```
# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#         Fabian Pedregosa <fabian.pedregosa@inria.fr>
#
# License: BSD 3 clause (C) INRIA


# #############################################################################
# Generate sample data
import numpy as np
import matplotlib.pyplot as plt
from skhubness.neighbors import KNeighborsRegressor

np.random.seed(0)
X = np.sort(5 * np.random.rand(40, 1), axis=0)
T = np.linspace(0, 5, 500)[:, np.newaxis]
y = np.sin(X).ravel()

# Add noise to targets
y[::5] += 1 * (0.5 - np.random.rand(8))

# #############################################################################
# Fit regression model
n_neighbors = 5

f = plt.figure()
for i, weights in enumerate(['uniform', 'distance']):
    for j, hubness in enumerate([None, 'local_scaling']):
        knn = KNeighborsRegressor(n_neighbors,
                                  algorithm_params={'n_candidates': 39},
                                  weights=weights,
                                  hubness=hubness)
        y_ = knn.fit(X, y).predict(T)

        plt.subplot(2, 2, i * 2 + j + 1)
        f.set_figheight(15)
        f.set_figwidth(15)
        plt.scatter(X, y, c='k', label='data')
        plt.plot(T, y_, c='g', label='prediction')
        plt.axis('tight')
        plt.legend()
        plt.title(f"KNeighborsRegressor (k = {n_neighbors}, weights = '{weights}',␣
↪hubness = '{hubness}')")

plt.tight_layout()
plt.show()
```

Total running time of the script: (0 minutes 0.737 seconds)

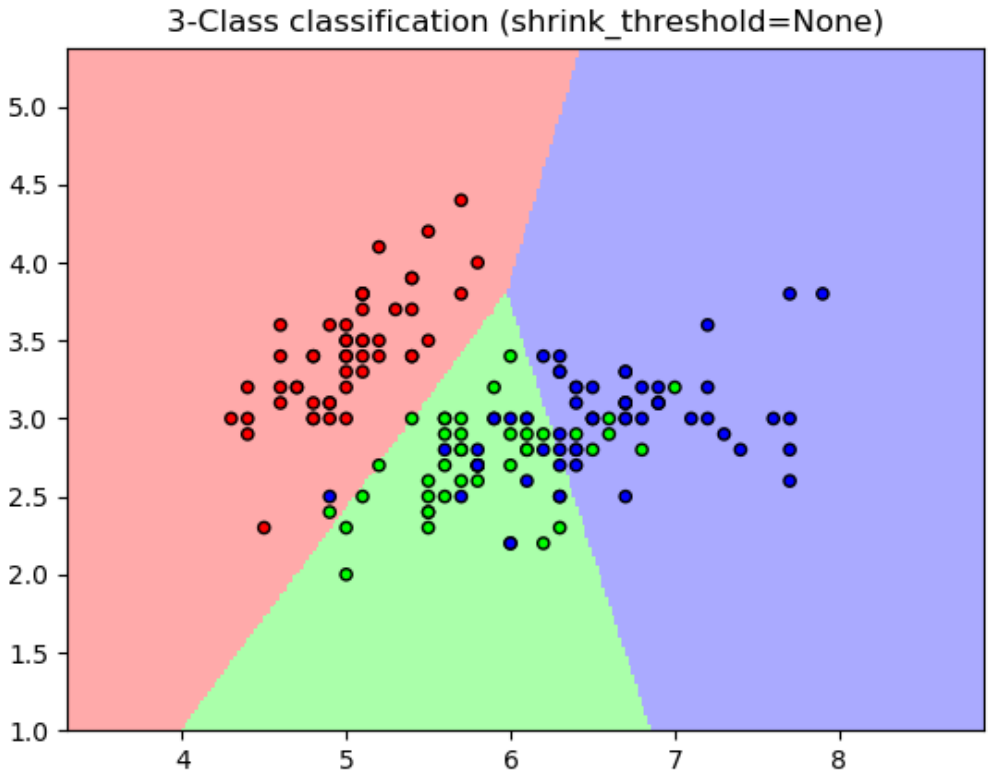
Note: Click *here* to download the full example code

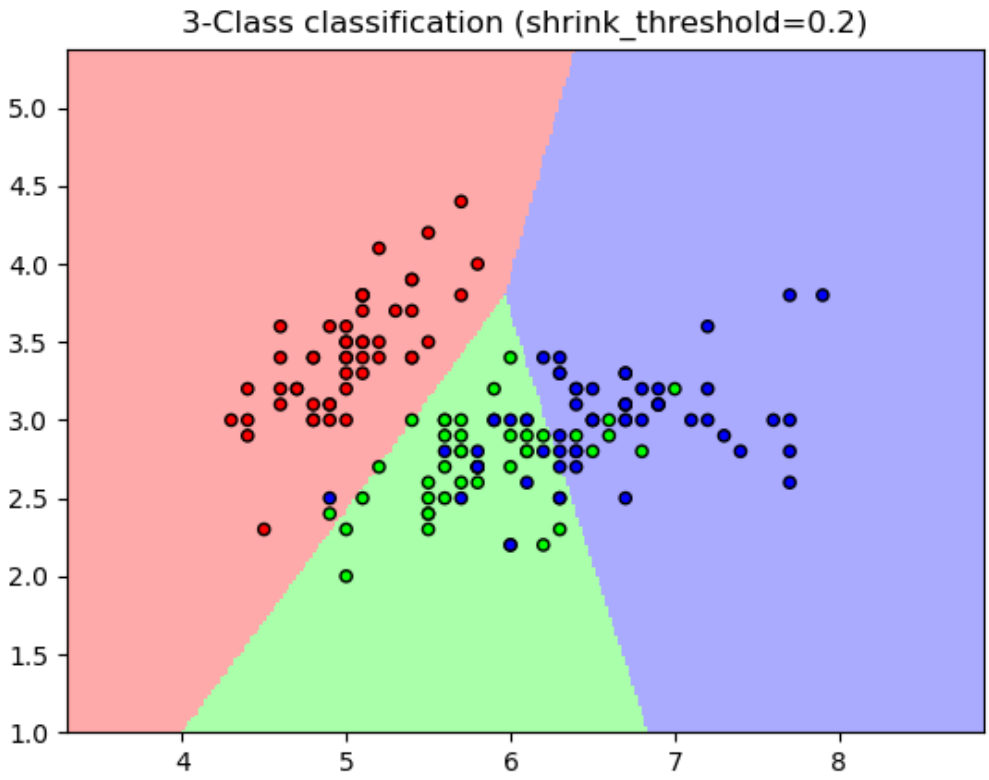
Nearest Centroid Classification

Sample usage of Nearest Centroid classification. It will plot the decision boundaries for each class.

Note that no hubness reduction is currently implemented for centroids. However, *hubness.neighbors* retains all the features of *sklearn.neighbors*, in order to act as a full drop-in replacement.

Adapted from https://scikit-learn.org/stable/auto_examples/neighbors/plot_nearest_centroid.html





-

Out:

```
None 0.8133333333333334
0.2 0.82
/home/user/feldbauer/PycharmProjects/hubness/examples/sklearn/plot_nearest_centroid.
↪py:64: UserWarning: Matplotlib is currently using agg, which is a non-GUI backend,␣
↪so cannot show the figure.
  plt.show()
```

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from skhubness.neighbors import NearestCentroid

n_neighbors = 15

# import some data to play with
iris = datasets.load_iris()
# we only take the first two features. We could avoid this ugly
```

(continues on next page)

(continued from previous page)

```
# slicing by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target

h = .02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

for shrinkage in [None, .2]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = NearestCentroid(shrink_threshold=shrinkage)
    clf.fit(X, y)
    y_pred = clf.predict(X)
    print(shrinkage, np.mean(y == y_pred))
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.title("3-Class classification (shrink_threshold=%r)"
              % shrinkage)
    plt.axis('tight')

plt.show()
```

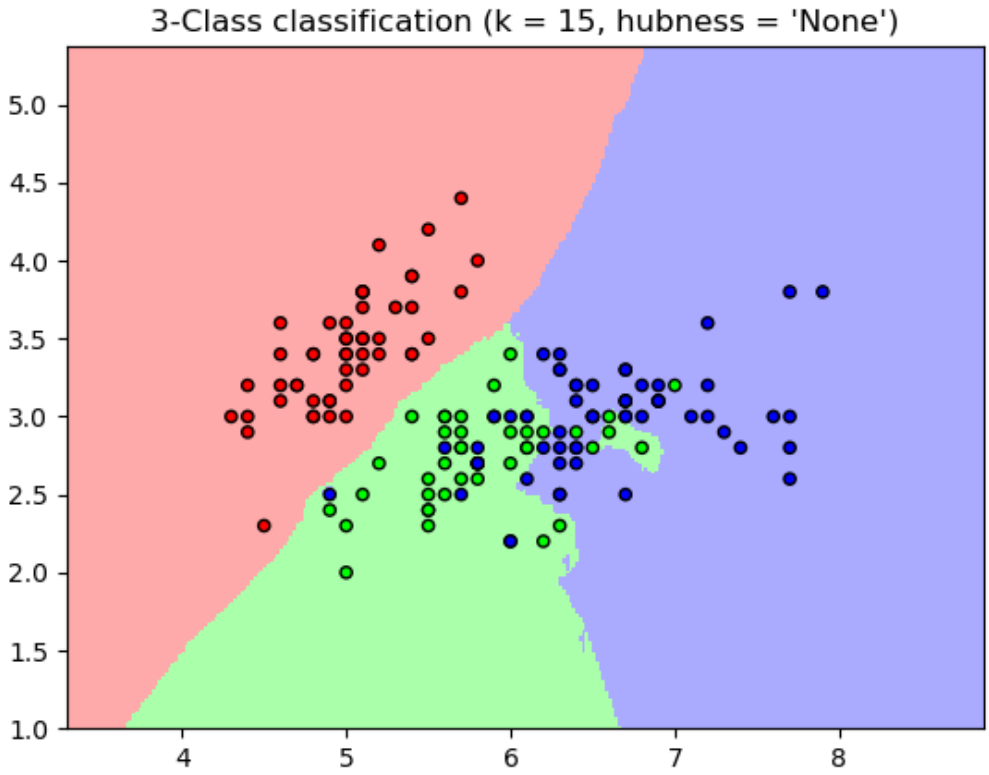
Total running time of the script: (0 minutes 0.737 seconds)

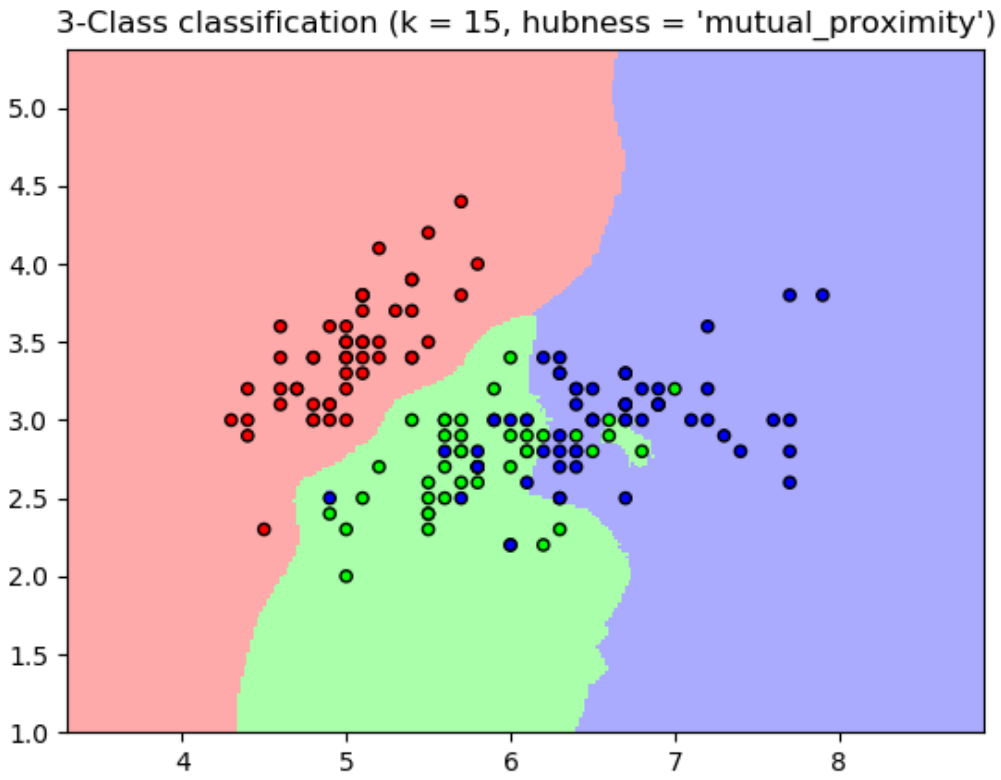
Note: Click *here* to download the full example code

Nearest Neighbors Classification

Sample usage of Nearest Neighbors classification. It will plot the decision boundaries for each class.

Adapted from https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html





-

Out:

```
/home/user/feldbauer/PycharmProjects/hubness/examples/sklearn/plot_classification.
↪py:61: UserWarning: Matplotlib is currently using agg, which is a non-GUI backend,␣
↪so cannot show the figure.
  plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from skhubness.neighbors import KNeighborsClassifier

n_neighbors = 15

# import some data to play with
iris = datasets.load_iris()

# we only take the first two features. We could avoid this ugly
# slicing by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target
```

(continues on next page)

(continued from previous page)

```
h = .02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

for hubness in [None, 'mutual_proximity']:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = KNeighborsClassifier(n_neighbors,
                               hubness=hubness,
                               weights='distance')
    clf.fit(X, y)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("3-Class classification (k = %i, hubness = '%s')"
              % (n_neighbors, hubness))

plt.show()
```

Total running time of the script: (0 minutes 25.940 seconds)

Note: Click *here* to download the full example code

Dimensionality Reduction with Neighborhood Components Analysis

Sample usage of Neighborhood Components Analysis for dimensionality reduction.

This example compares different (linear) dimensionality reduction methods applied on the Digits data set. The data set contains images of digits from 0 to 9 with approximately 180 samples of each class. Each image is of dimension 8x8 = 64, and is reduced to a two-dimensional data point.

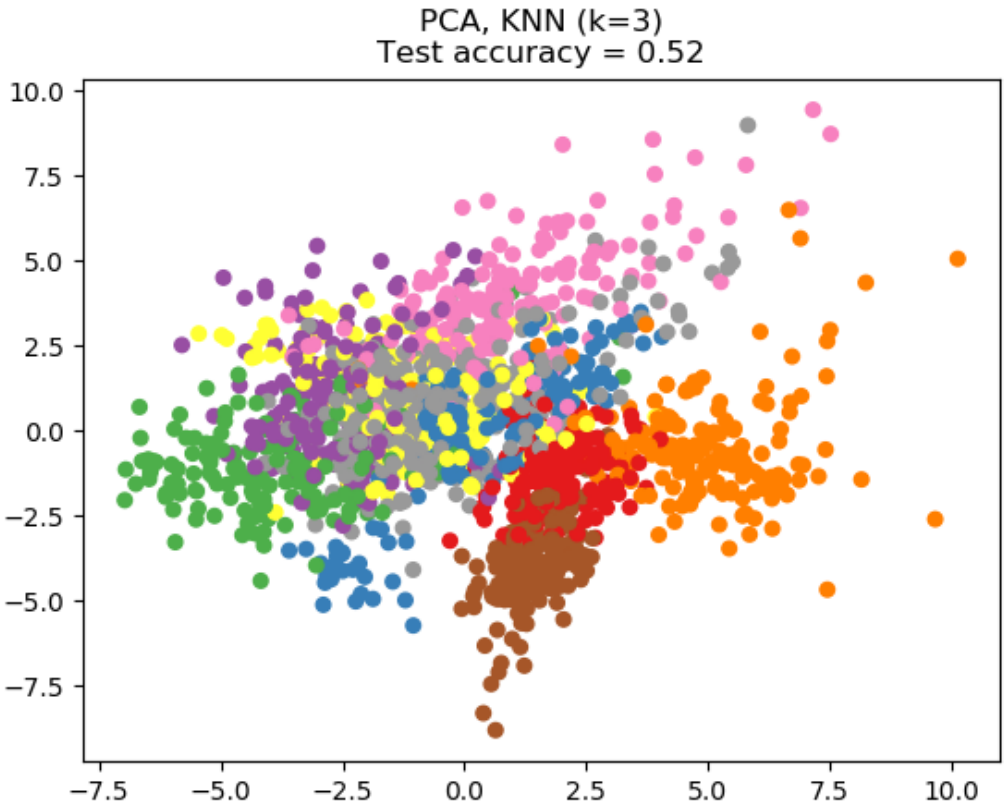
Principal Component Analysis (PCA) applied to this data identifies the combination of attributes (principal components, or directions in the feature space) that account for the most variance in the data. Here we plot the different samples on the 2 first principal components.

Linear Discriminant Analysis (LDA) tries to identify attributes that account for the most variance *between classes*. In particular, LDA, in contrast to PCA, is a supervised method, using known class labels.

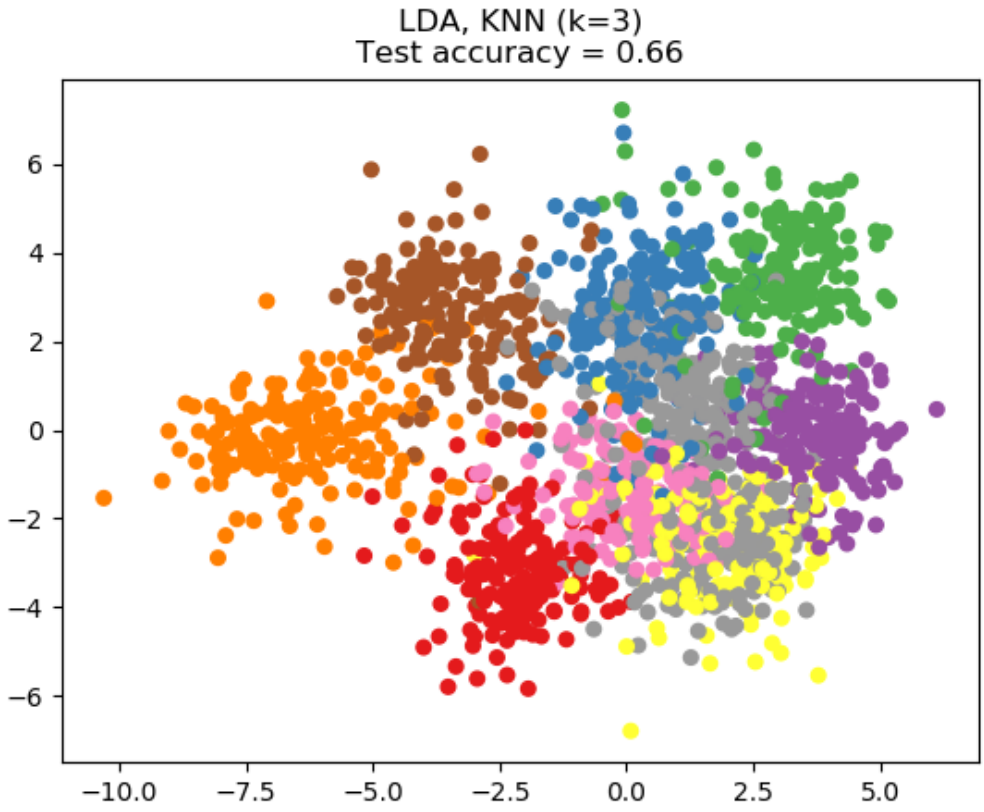
Neighborhood Components Analysis (NCA) tries to find a feature space such that a stochastic nearest neighbor algorithm will give the best accuracy. Like LDA, it is a supervised method.

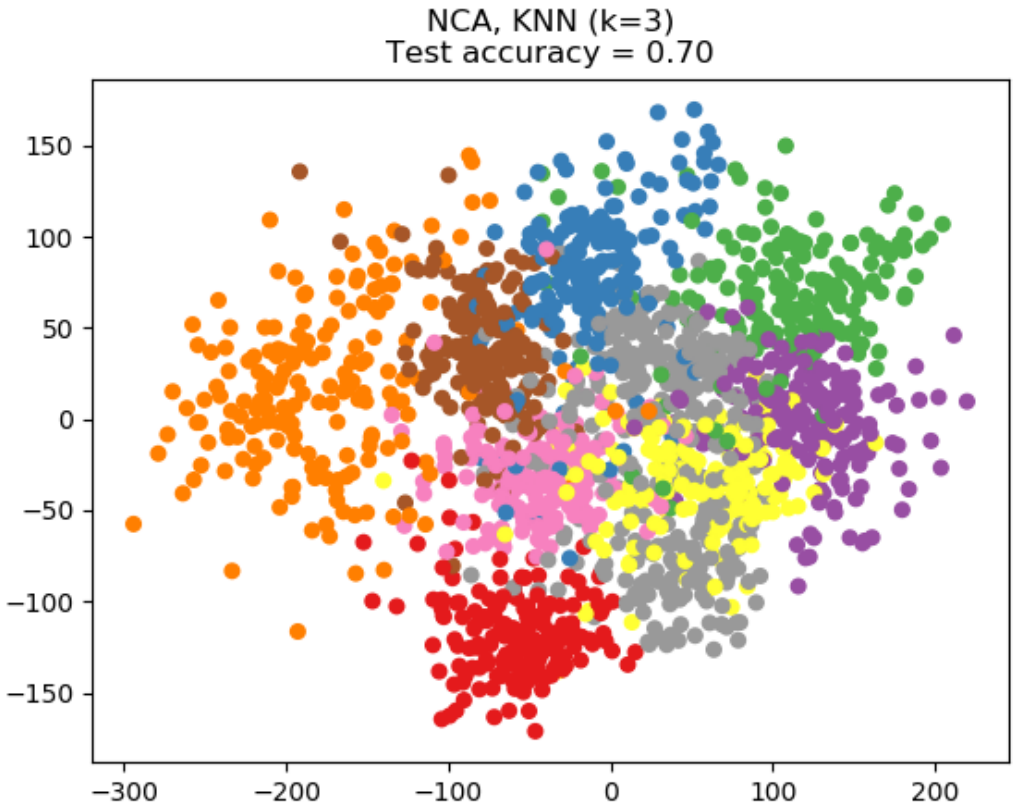
One can see that NCA enforces a clustering of the data that is visually meaningful despite the large reduction in dimension.

Adapted from https://scikit-learn.org/stable/auto_examples/neighbors/plot_nca_dim_reduction.html



•


LDA, KNN (k=3)
Test accuracy = 0.66
6
4
2
0
−2
−4
−6
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0



-

Out:

```
/home/user/feldbauer/miniconda3/envs/hubness/lib/python3.7/site-packages/sklearn/
↪discriminant_analysis.py:388: UserWarning: Variables are collinear.
  warnings.warn("Variables are collinear.")
/home/user/feldbauer/miniconda3/envs/hubness/lib/python3.7/site-packages/sklearn/
↪discriminant_analysis.py:388: UserWarning: Variables are collinear.
  warnings.warn("Variables are collinear.")
/home/user/feldbauer/PycharmProjects/hubness/examples/sklearn/plot_nca_dim_reduction.
↪py:103: UserWarning: Matplotlib is currently using agg, which is a non-GUI backend,␣
↪so cannot show the figure.
  plt.show()
```

```
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.pipeline import make_pipeline
```

(continues on next page)

(continued from previous page)

```
from sklearn.preprocessing import StandardScaler

from skhubness.neighbors import (KNeighborsClassifier,
                                 NeighborhoodComponentsAnalysis)

print(__doc__)

n_neighbors = 3
random_state = 0

# Load Digits dataset
digits = datasets.load_digits()
X, y = digits.data, digits.target

# Split into train/test
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.5, stratify=y,
                     random_state=random_state)

dim = len(X[0])
n_classes = len(np.unique(y))

# Reduce dimension to 2 with PCA
pca = make_pipeline(StandardScaler(),
                    PCA(n_components=2, random_state=random_state))

# Reduce dimension to 2 with LinearDiscriminantAnalysis
lda = make_pipeline(StandardScaler(),
                    LinearDiscriminantAnalysis(n_components=2))

# Reduce dimension to 2 with NeighborhoodComponentAnalysis
nca = make_pipeline(StandardScaler(),
                    NeighborhoodComponentsAnalysis(n_components=2,
                                                   random_state=random_state))

# Use a nearest neighbor classifier to evaluate the methods
knn = KNeighborsClassifier(n_neighbors=n_neighbors)

# Make a list of the methods to be compared
dim_reduction_methods = [('PCA', pca), ('LDA', lda), ('NCA', nca)]

# plt.figure()
for i, (name, model) in enumerate(dim_reduction_methods):
    plt.figure()
    # plt.subplot(1, 3, i + 1, aspect=1)

    # Fit the method's model
    model.fit(X_train, y_train)

    # Fit a nearest neighbor classifier on the embedded training set
    knn.fit(model.transform(X_train), y_train)

    # Compute the nearest neighbor accuracy on the embedded test set
    acc_knn = knn.score(model.transform(X_test), y_test)

    # Embed the data set in 2 dimensions using the fitted model
    X_embedded = model.transform(X)
```

(continues on next page)

(continued from previous page)

```
    # Plot the projected points and show the evaluation score
    plt.scatter(X_embedded[:, 0], X_embedded[:, 1], c=y, s=30, cmap='Set1')
    plt.title("{}, KNN (k={})\nTest accuracy = {:.2f}".format(name,
                                                              n_neighbors,
                                                              acc_knn))
plt.show()
```

Total running time of the script: (0 minutes 5.249 seconds)

Note: Click *here* to download the full example code

Face completion with a multi-output estimators

This example shows the use of multi-output estimator to complete images. The goal is to predict the lower half of a face given its upper half.

The first column of images shows true faces. The next columns illustrate how extremely randomized trees, linear regression, ridge regression, and k nearest neighbors with or without hubness reduction complete the lower half of those faces.

Adapted from https://scikit-learn.org/stable/auto_examples/plot_multioutput_face_completion.html

Out:

```
/home/user/feldbauer/PycharmProjects/hubness/examples/sklearn/plot_multioutput_face_
↪completion.py:106: UserWarning: Matplotlib is currently using agg, which is a non-
↪GUI backend, so cannot show the figure.
  plt.show()
```

```
print(__doc__)

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

(continues on next page)

(continued from previous page)

```
from sklearn.datasets import fetch_olivetti_faces
from sklearn.utils.validation import check_random_state

from sklearn.ensemble import ExtraTreesRegressor
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import RidgeCV

from skhubness.neighbors import KNeighborsRegressor

# Load the faces datasets
data = fetch_olivetti_faces()
targets = data.target

data = data.images.reshape((len(data.images), -1))
train = data[targets < 30]
test = data[targets >= 30]  # Test on independent people

# Test on a subset of people
n_faces = 5
rng = check_random_state(4)
face_ids = rng.randint(test.shape[0], size=(n_faces, ))
test = test[face_ids, :]

n_pixels = data.shape[1]
# Upper half of the faces
X_train = train[:, :(n_pixels + 1) // 2]
# Lower half of the faces
y_train = train[:, n_pixels // 2:]
X_test = test[:, :(n_pixels + 1) // 2]
y_test = test[:, n_pixels // 2:]

# Fit estimators
ESTIMATORS = {
    "Extra trees": ExtraTreesRegressor(n_estimators=10, max_features=32,
                                       random_state=0),
    "k-NN": KNeighborsRegressor(weights='distance'),
    "k-NN MP": KNeighborsRegressor(hubness='mp',
                                   hubness_params={'method': 'normal'},
                                   weights='distance'),
    "Linear regression": LinearRegression(),
    "Ridge": RidgeCV(),
}

y_test_predict = dict()
for name, estimator in ESTIMATORS.items():
    estimator.fit(X_train, y_train)
    y_test_predict[name] = estimator.predict(X_test)

# Plot the completed faces
image_shape = (64, 64)

n_cols = 1 + len(ESTIMATORS)
plt.figure(figsize=(2. * n_cols, 2.26 * n_faces))
plt.suptitle("Face completion with multi-output estimators", size=16)

for i in range(n_faces):
    true_face = np.hstack((X_test[i], y_test[i]))
```

(continues on next page)

(continued from previous page)

```
    if i:
        sub = plt.subplot(n_faces, n_cols, i * n_cols + 1)
    else:
        sub = plt.subplot(n_faces, n_cols, i * n_cols + 1,
                          title="true faces")

    sub.axis("off")
    sub.imshow(true_face.reshape(image_shape),
               cmap=plt.cm.gray,
               interpolation="nearest")

    for j, est in enumerate(sorted(ESTIMATORS)):
        completed_face = np.hstack((X_test[i], y_test_predict[est][i]))

        if i:
            sub = plt.subplot(n_faces, n_cols, i * n_cols + 2 + j)

        else:
            sub = plt.subplot(n_faces, n_cols, i * n_cols + 2 + j,
                              title=est)

        sub.axis("off")
        sub.imshow(completed_face.reshape(image_shape),
                   cmap=plt.cm.gray,
                   interpolation="nearest")

plt.show()
```

Total running time of the script: (0 minutes 3.385 seconds)

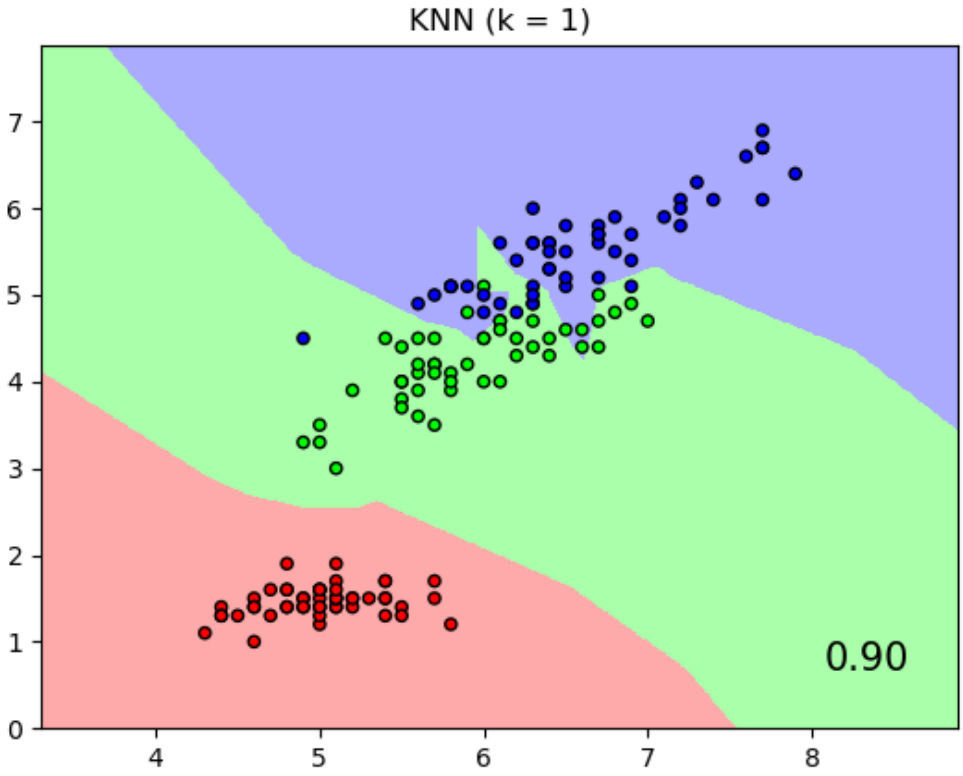
Note: Click *here* to download the full example code

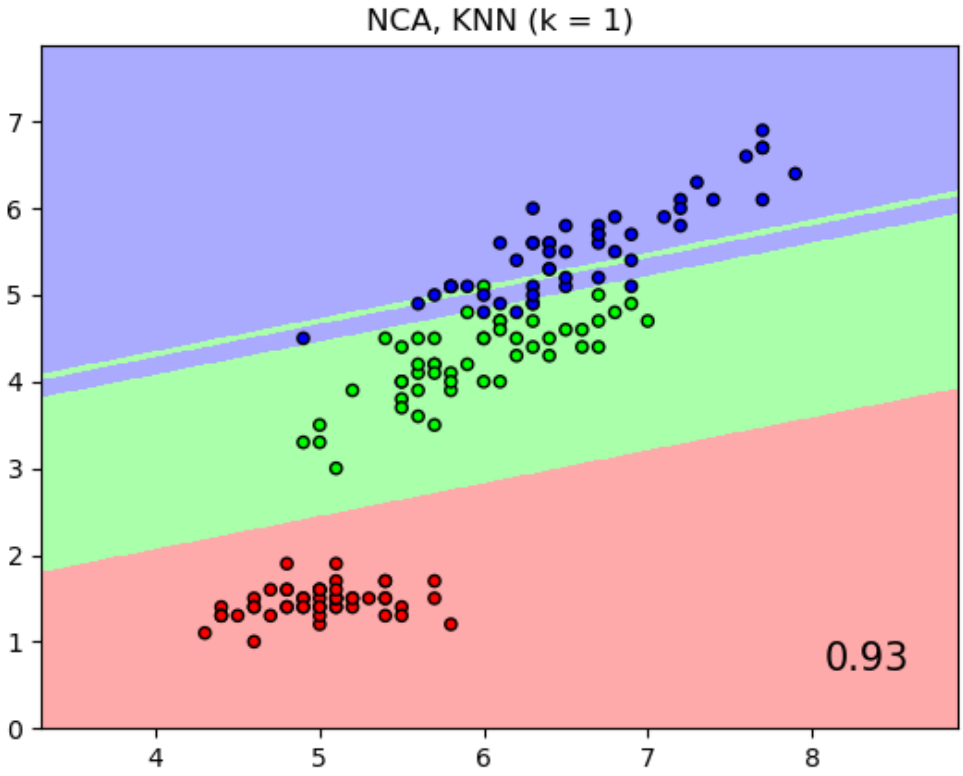
Comparing Nearest Neighbors with and without Neighborhood Components Analysis

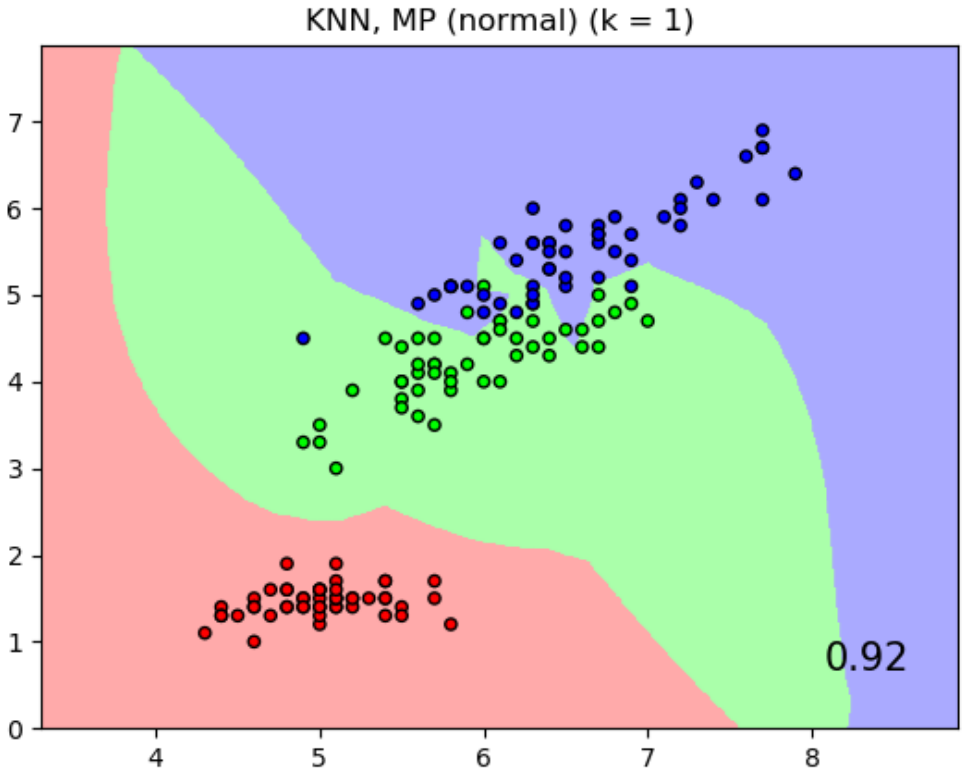
An example comparing nearest neighbors classification with and without Neighborhood Components Analysis.

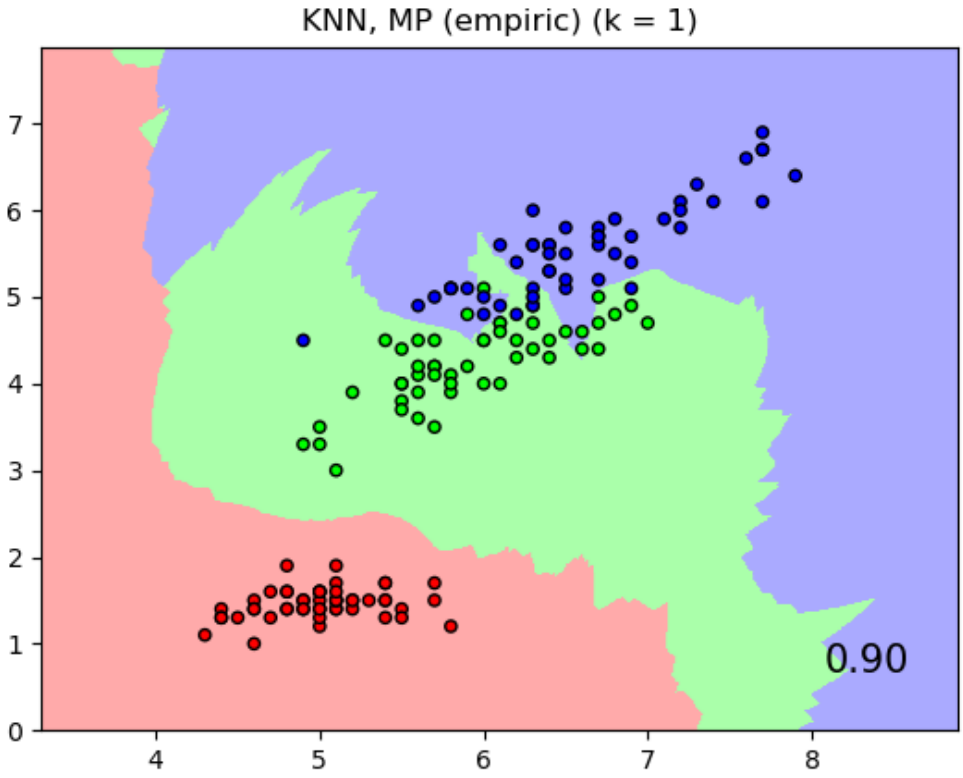
It will plot the class decision boundaries given by a Nearest Neighbors classifier when using the Euclidean distance on the original features, versus using the Euclidean distance after the transformation learned by Neighborhood Components Analysis. The latter aims to find a linear transformation that maximises the (stochastic) nearest neighbor classification accuracy on the training set.

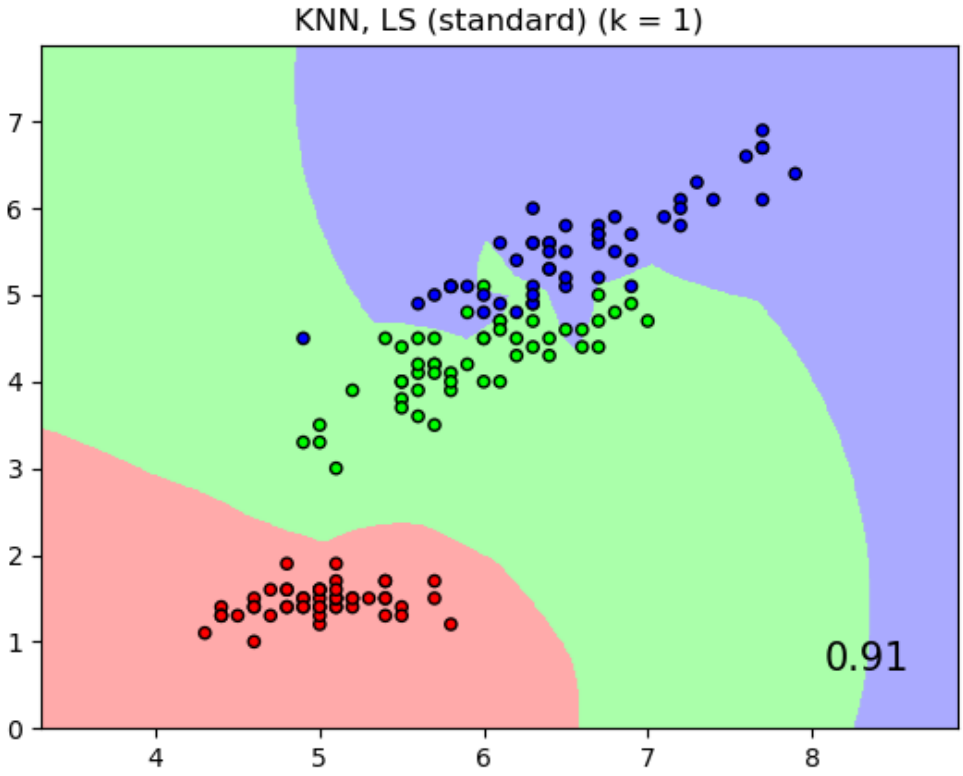
Adapted from https://scikit-learn.org/stable/auto_examples/neighbors/plot_nca_classification.html

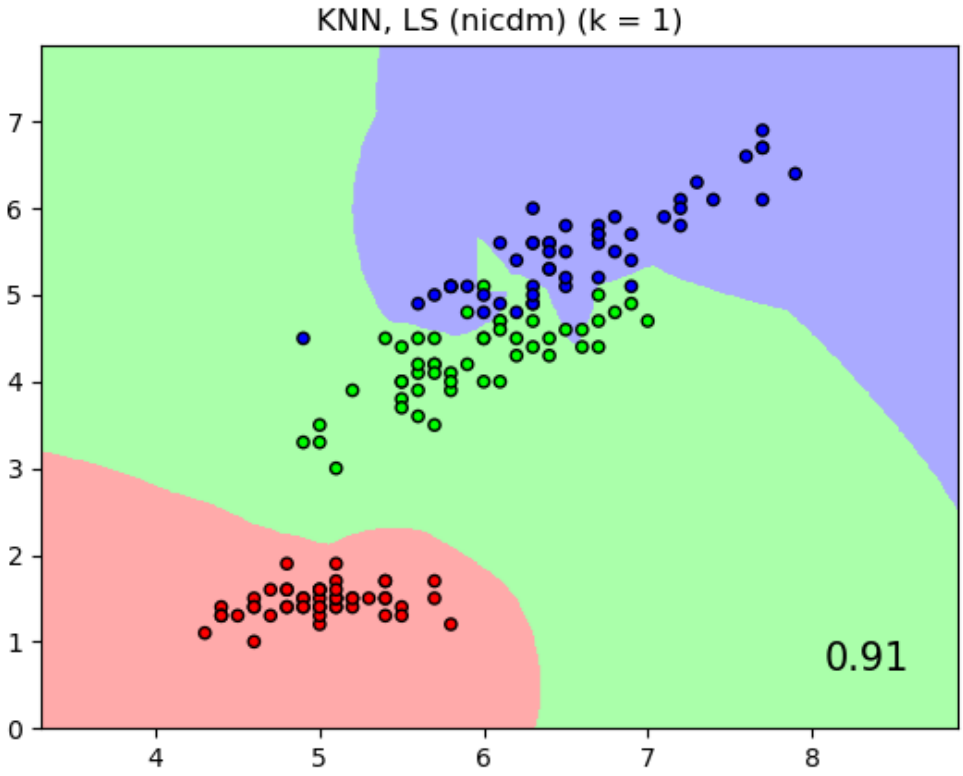

KNN (k = 1)
0.90
0
1
2
3
4
5
6
7
4
5
6
7
8


NCA, KNN (k = 1)
0.93
0
1
2
3
4
5
6
7
4
5
6
7
8


KNN, MP (normal) (k = 1)
0.92
0
1
2
3
4
5
6
7
4
5
6
7
8


KNN, MP (empiric) (k = 1)
0.90
0
1
2
3
4
5
6
7
4
5
6
7
8


KNN, LS (standard) (k = 1)
0.91
0
1
2
3
4
5
6
7
4
5
6
7
8



-

Out:

```
```

```
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

from skhubness.neighbors import (KNeighborsClassifier,
                                 NeighborhoodComponentsAnalysis)
import warnings
warnings.filterwarnings('ignore')

print(__doc__)

n_neighbors = 1
```

(continues on next page)

(continued from previous page)

```
dataset = datasets.load_iris()
X, y = dataset.data, dataset.target

# we only take two features. We could avoid this ugly
# slicing by using a two-dim dataset
X = X[:, [0, 2]]

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, stratify=y, test_size=0.7, random_state=42)

h = .01  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

names = ['KNN',
         'NCA, KNN',
         'KNN, MP (normal)',
         'KNN, MP (empiric)',
         'KNN, LS (standard)',
         'KNN, LS (nicdm)',
        ]

classifiers = [Pipeline([('scaler', StandardScaler()),
                         ('knn', KNeighborsClassifier(n_neighbors=n_neighbors))
                         ]),
               Pipeline([('scaler', StandardScaler()),
                         ('nca', NeighborhoodComponentsAnalysis()),
                         ('knn', KNeighborsClassifier(n_neighbors=n_neighbors))
                         ]),
               Pipeline([('scaler', StandardScaler()),
                         ('knn', KNeighborsClassifier(n_neighbors=n_neighbors,
                                                      hubness='mutual_proximity',
                                                      hubness_params={'method':
↪'normal'}))
                         ]),
               Pipeline([('scaler', StandardScaler()),
                         ('knn', KNeighborsClassifier(n_neighbors=n_neighbors,
                                                      hubness='mutual_proximity',
                                                      hubness_params={'method':
↪'empiric'}))
                         ]),
               Pipeline([('scaler', StandardScaler()),
                         ('knn', KNeighborsClassifier(n_neighbors=n_neighbors,
                                                      hubness='local_scaling',
                                                      hubness_params={'method':
↪'standard'}))
                         ]),
               Pipeline([('scaler', StandardScaler()),
                         ('knn', KNeighborsClassifier(n_neighbors=n_neighbors,
                                                      hubness='local_scaling',
                                                      hubness_params={'method': 'nicdm
↪'}))
                         ]),
               ]
```

(continues on next page)

(continued from previous page)

```
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

for name, clf in zip(names, classifiers):

    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light, alpha=.8)

    # Plot also the training and testing points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("{} (k = {})".format(name, n_neighbors))
    plt.text(0.9, 0.1, '{:.2f}'.format(score), size=15,
             ha='center', va='center', transform=plt.gca().transAxes)

plt.show()
```

Total running time of the script: (6 minutes 19.660 seconds)

Note: Click *here* to download the full example code

Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...

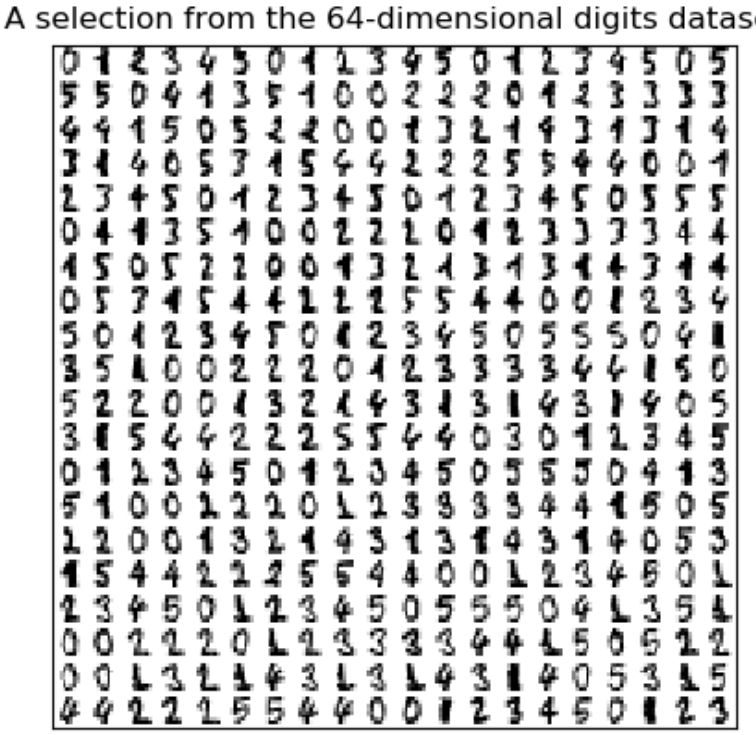
An illustration of various embeddings on the digits dataset.

The RandomTreesEmbedding, from the `sklearn.ensemble` module, is not technically a manifold embedding method, as it learn a high-dimensional representation on which we apply a dimensionality reduction method. However, it is often useful to cast a dataset into a representation in which the classes are linearly-separable.

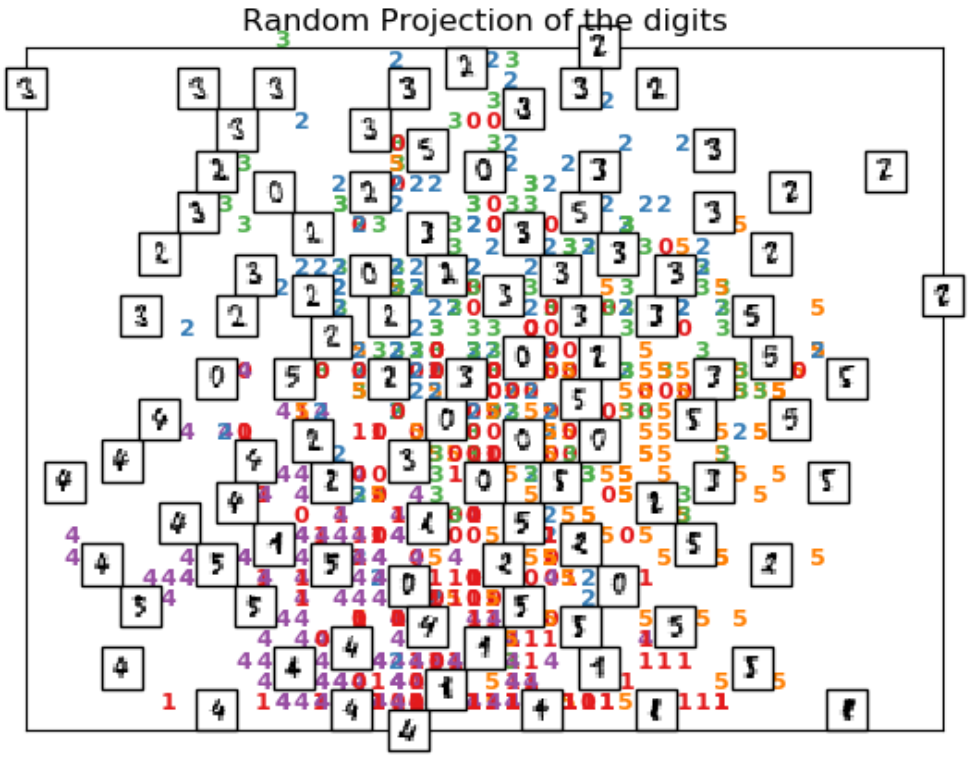
t-SNE will be initialized with the embedding that is generated by PCA in this example, which is not the default setting. It ensures global stability of the embedding, i.e., the embedding does not depend on random initialization.

Linear Discriminant Analysis, from the `sklearn.discriminant_analysis` module, and Neighborhood Components Analysis, from the `sklearn.neighbors` module, are supervised dimensionality reduction method, i.e. they make use of the provided labels, contrary to other methods.

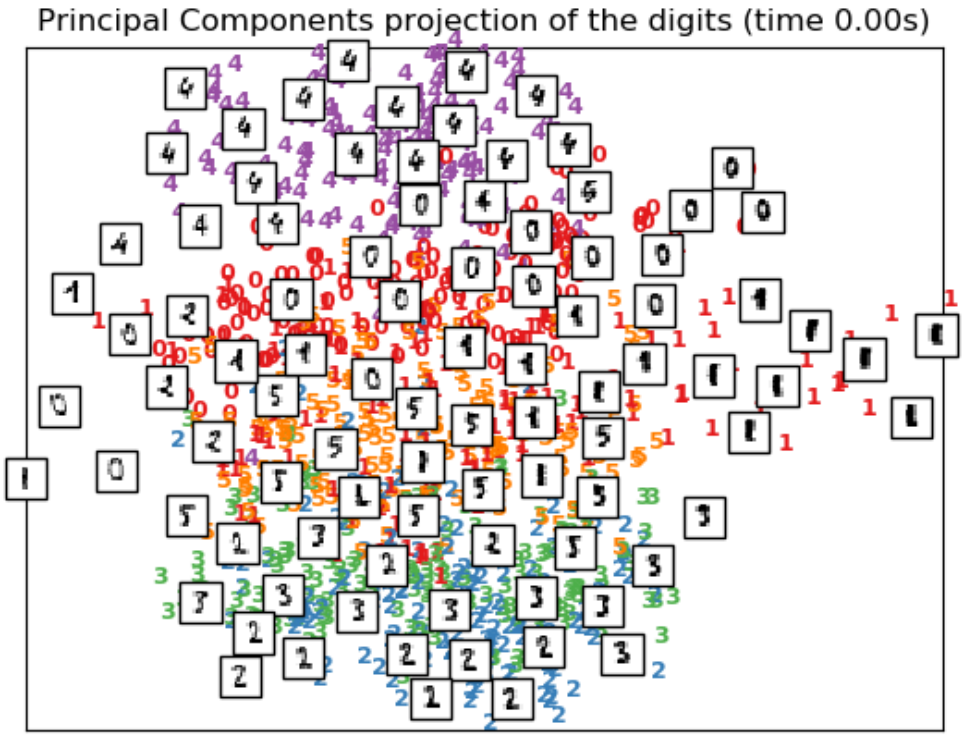
Adapted from https://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html


A selection from the 64-dimensional digits dataset

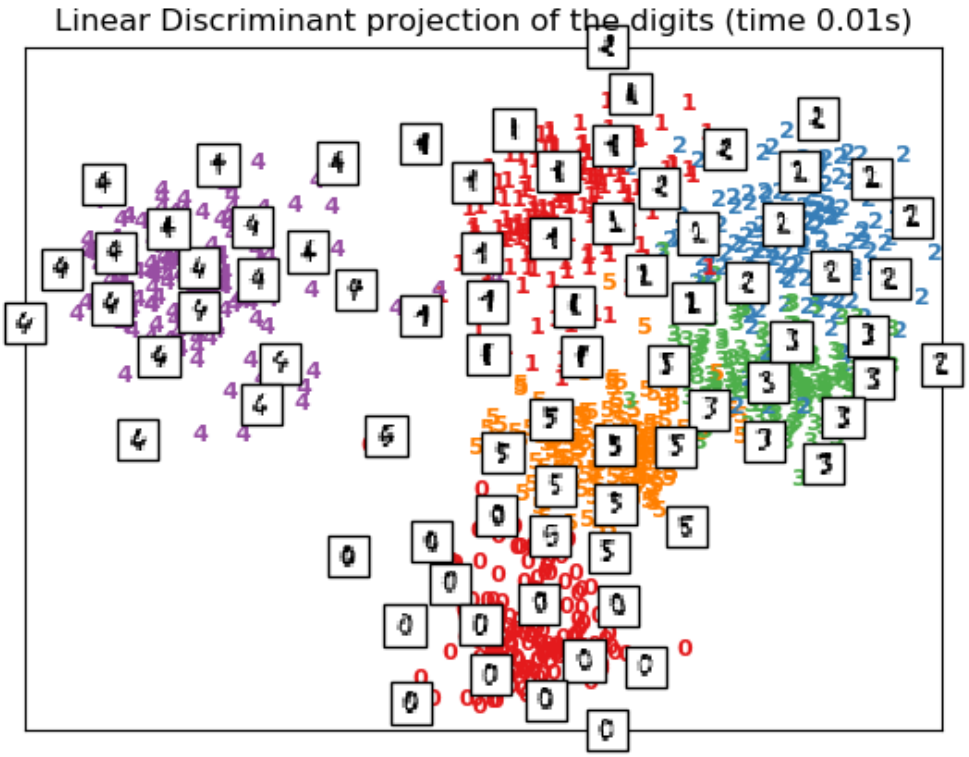
-


Random Projection of the digits

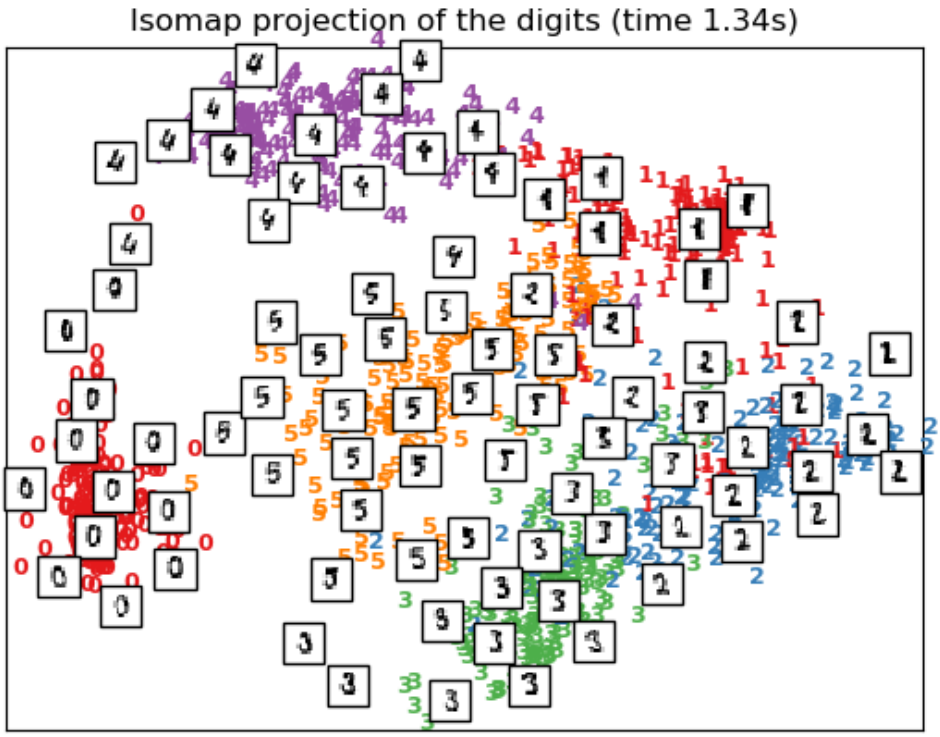
-


Principal Components projection of the digits (time 0.00s)

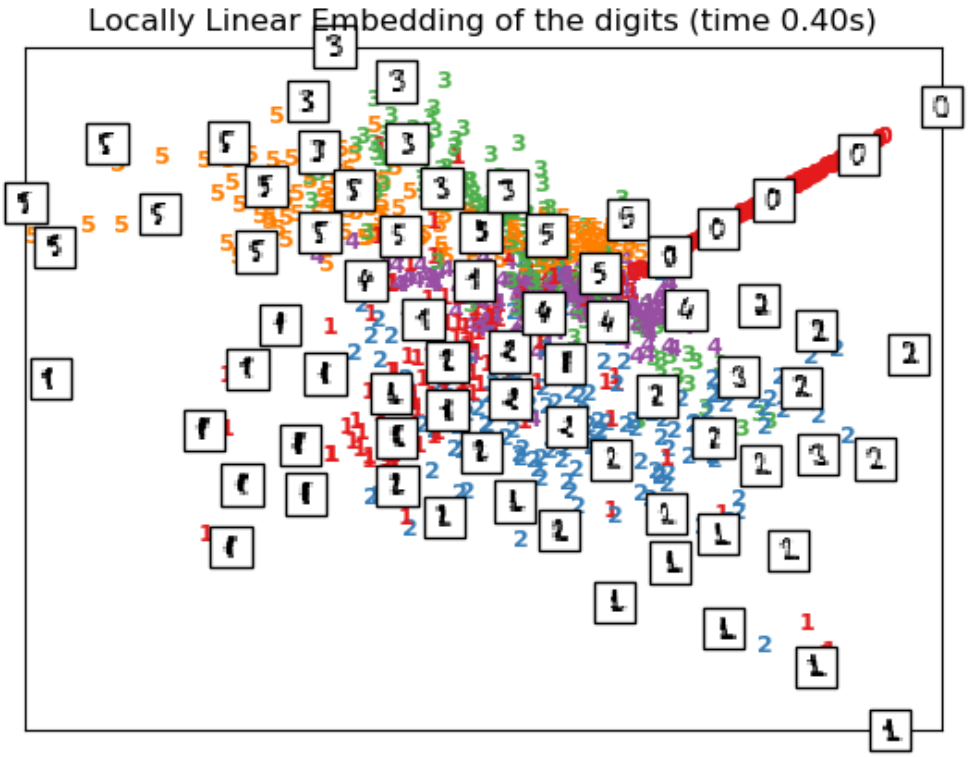
-


Linear Discriminant projection of the digits (time 0.01s)

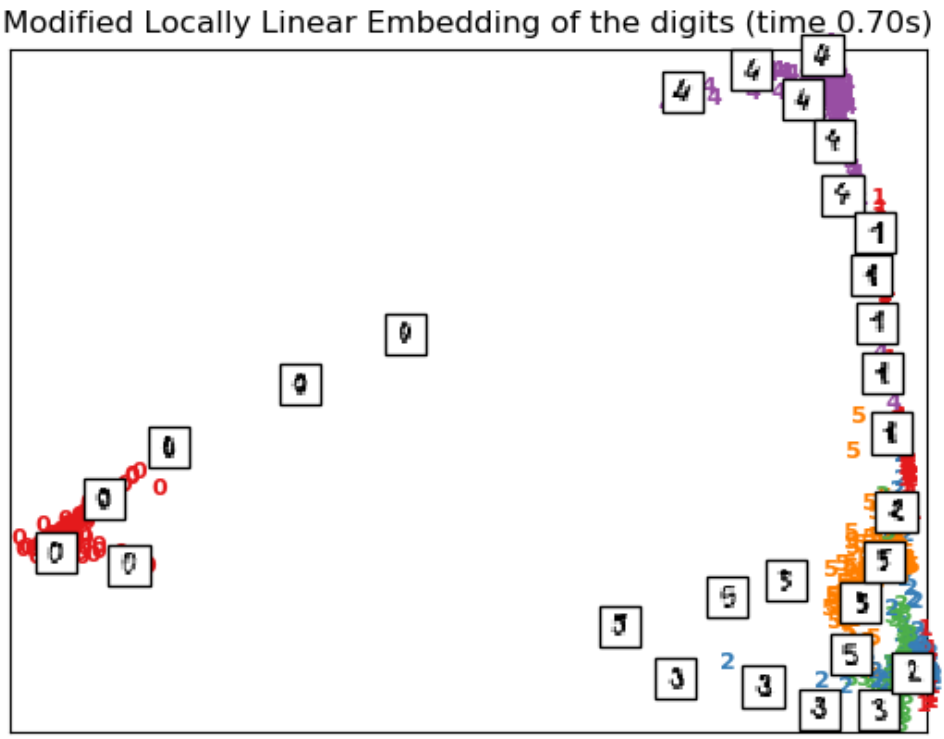
-


Isomap projection of the digits (time 1.34s)

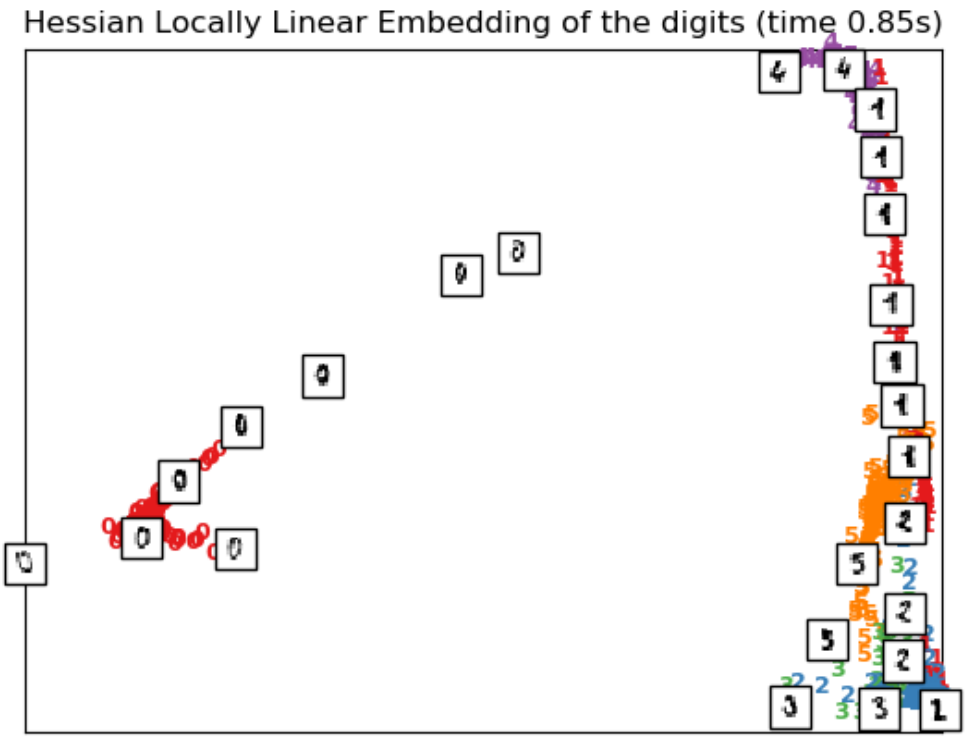
-


Locally Linear Embedding of the digits (time 0.40s)

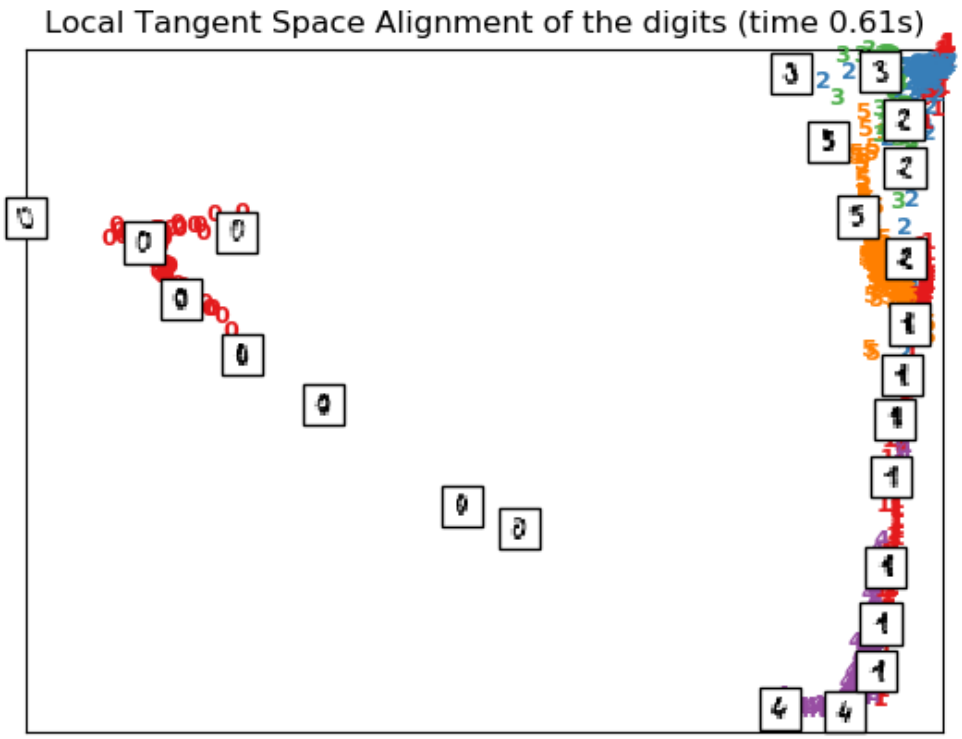
-


Modified Locally Linear Embedding of the digits (time 0.70s)

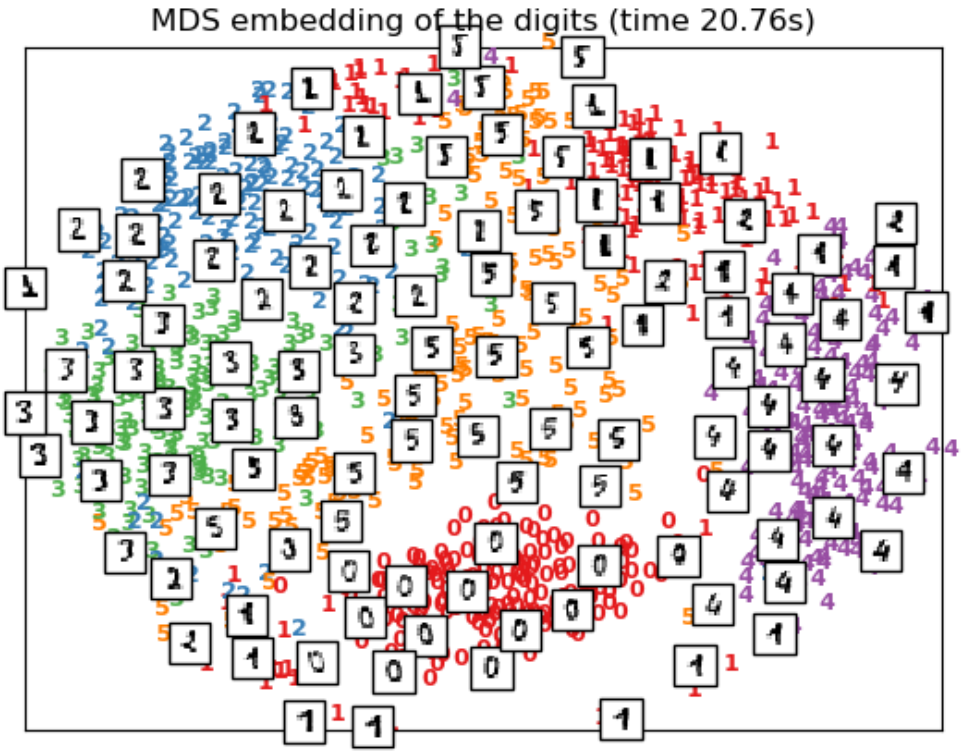
-


Hessian Locally Linear Embedding of the digits (time 0.85s)

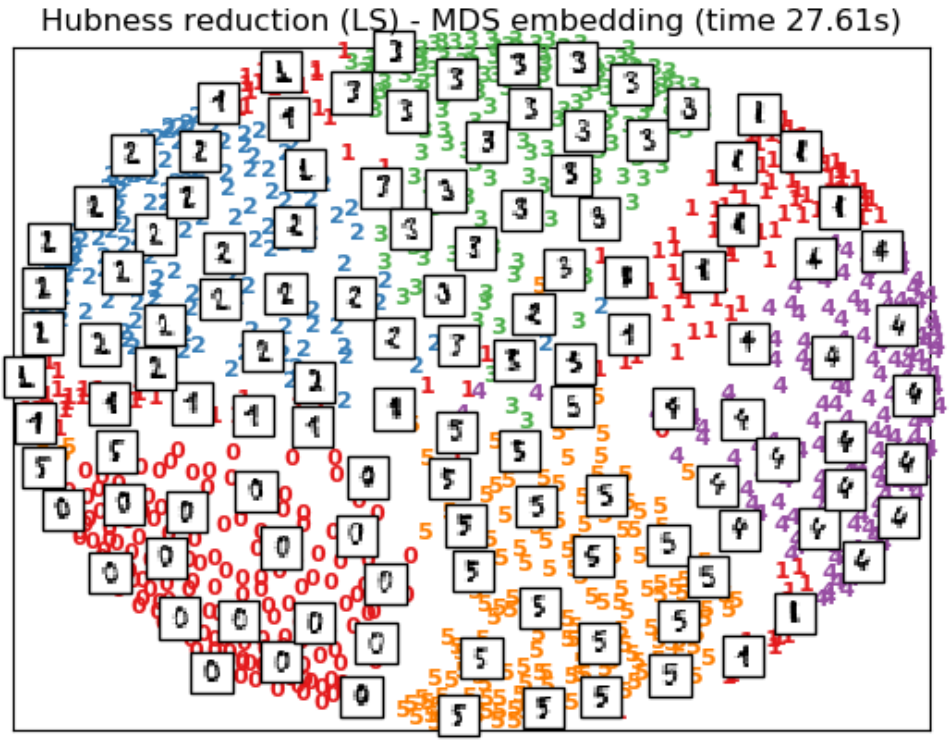
-


Local Tangent Space Alignment of the digits (time 0.61s)

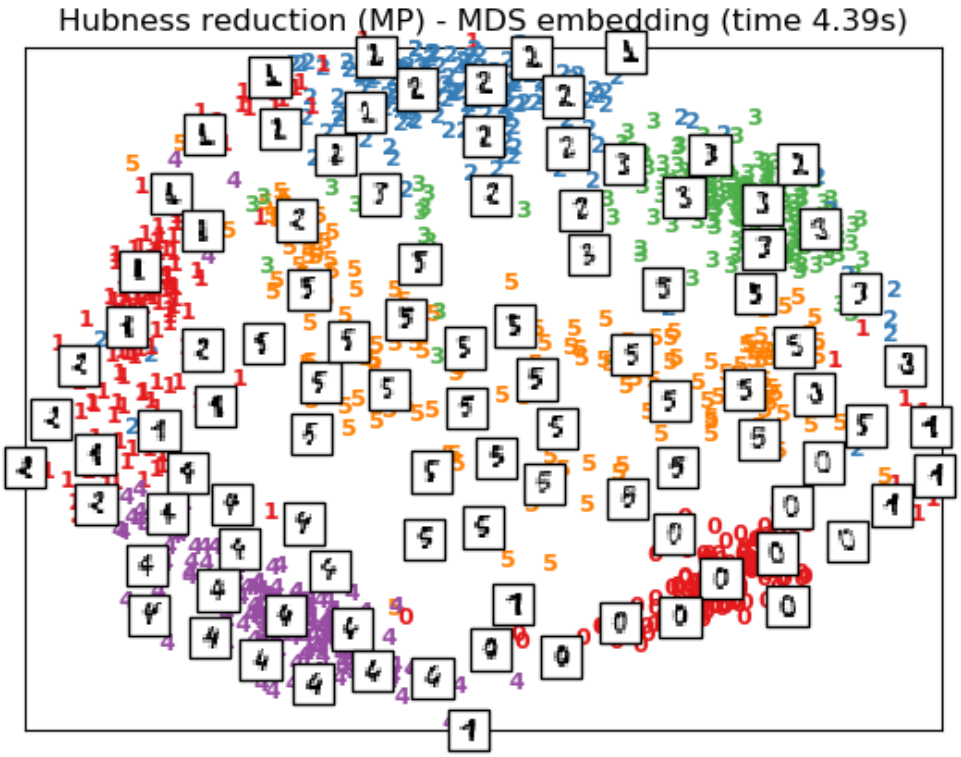
-


MDS embedding of the digits (time 20.76s)

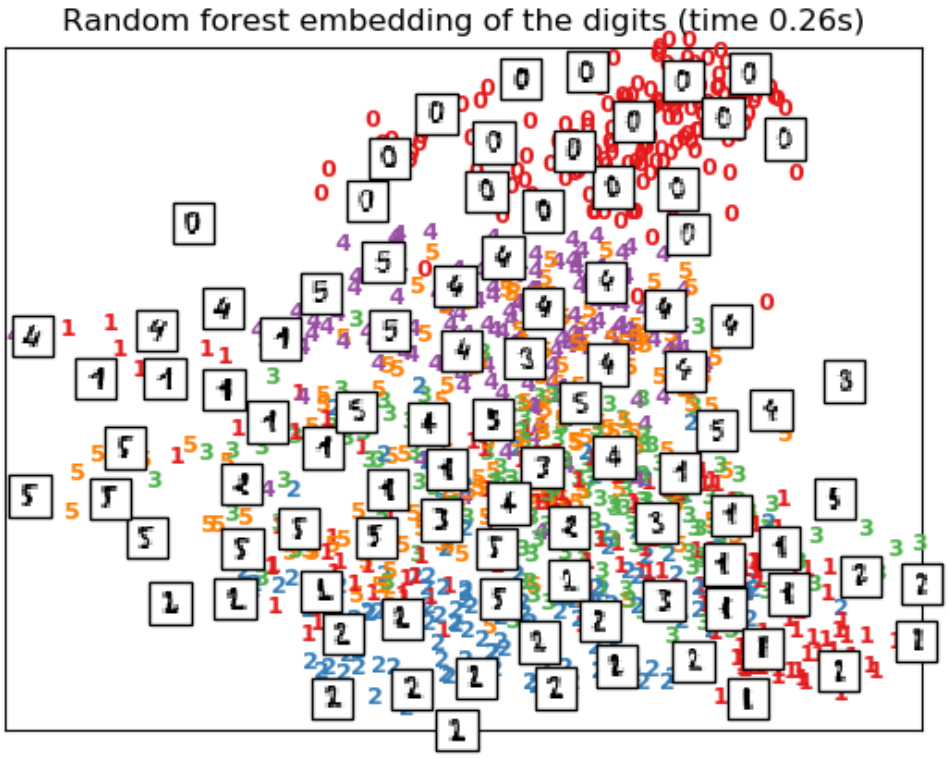
-


Hubness reduction (LS) - MDS embedding (time 27.61s)

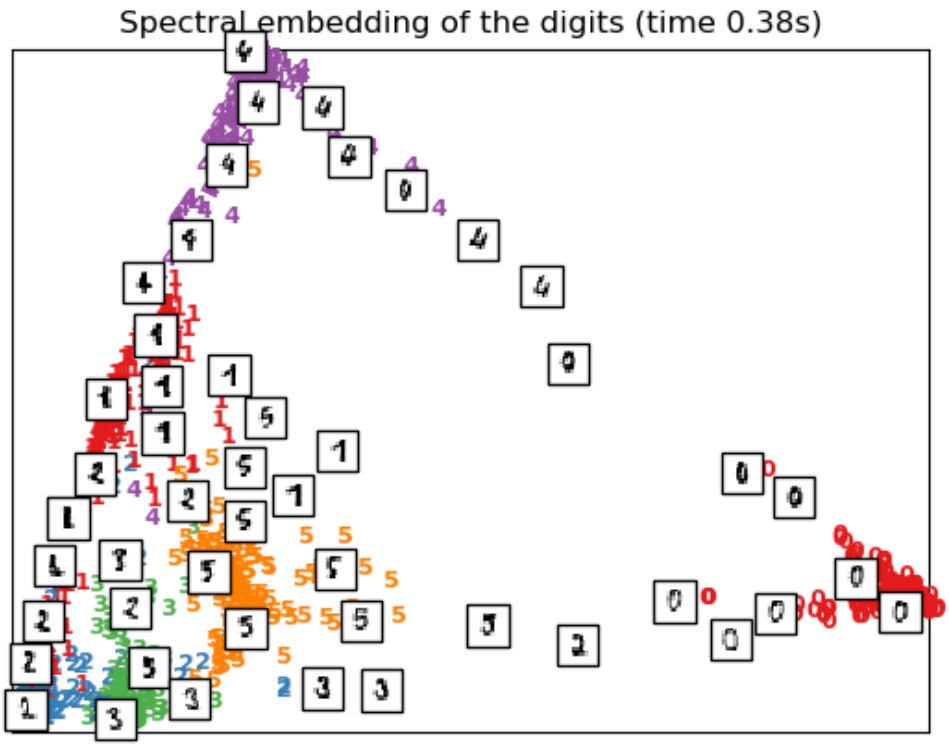
-


Hubness reduction (MP) - MDS embedding (time 4.39s)

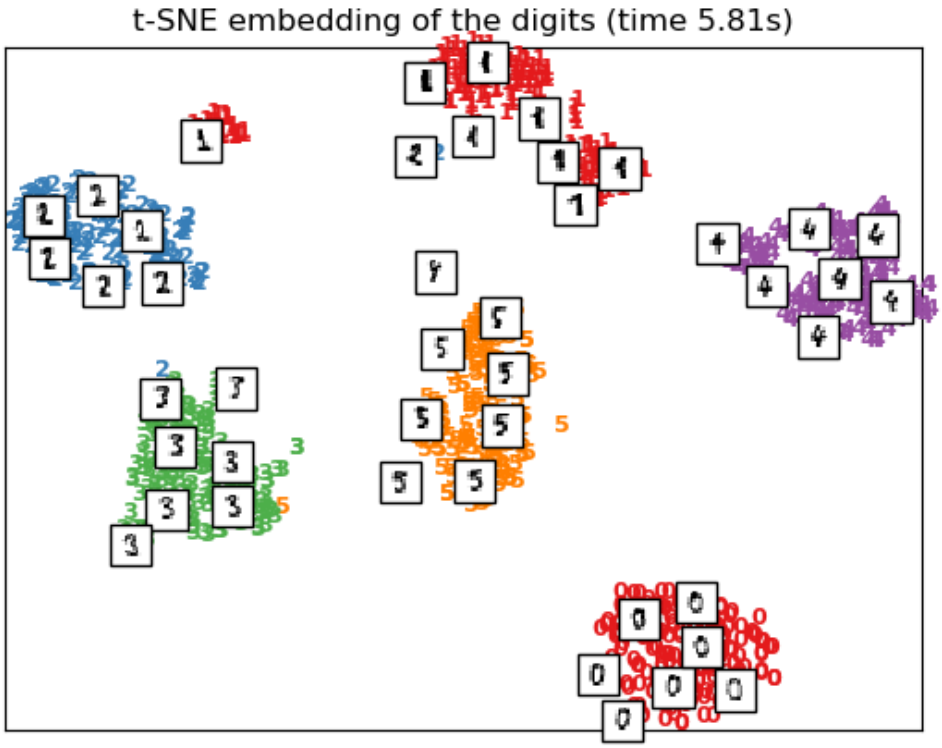
-


Random forest embedding of the digits (time 0.26s)

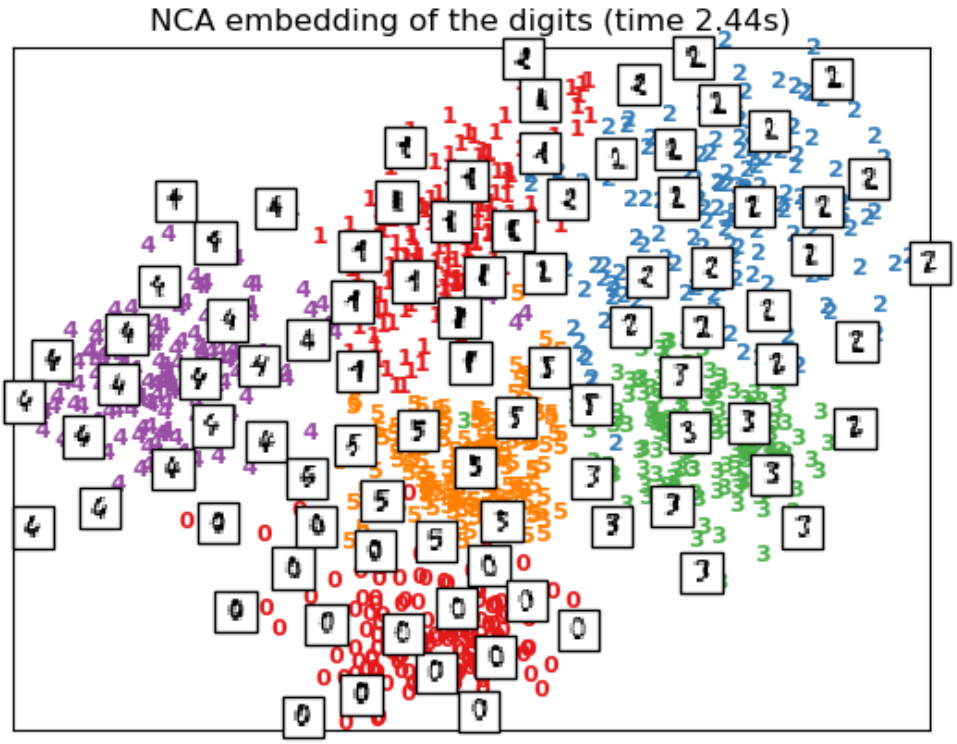
-


Spectral embedding of the digits (time 0.38s)

-


t-SNE embedding of the digits (time 5.81s)

-



-

Out:

```
Computing random projection
Computing PCA projection
Computing Linear Discriminant Analysis projection
Computing Isomap projection
Done.
Computing LLE embedding
Done. Reconstruction error: 1.63545e-06
Computing modified LLE embedding
Done. Reconstruction error: 0.360653
Computing Hessian LLE embedding
Done. Reconstruction error: 0.2128
Computing LTSA embedding
Done. Reconstruction error: 0.2128
Computing MDS embedding
Done. Stress: 135359737.175700
Computing MDS embedding from local scaling neighbors graph
Done. Stress: 89610.656628
Computing MDS embedding from mutual proximity graph
Done. Stress: 25752.919623
Computing Totally Random Trees embedding
Computing Spectral embedding
Computing t-SNE embedding
Computing NCA projection
```

```
# Authors: Fabian Pedregosa <fabian.pedregosa@inria.fr>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Gael Varoquaux
#          Roman Feldbauer
# License: BSD 3 clause (C) INRIA 2011

print(__doc__)
from time import time

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import offsetbox
from sklearn import (manifold, datasets, decomposition, ensemble,
                     discriminant_analysis, random_projection)
from skhubness import neighbors

digits = datasets.load_digits(n_class=6)
X = digits.data
y = digits.target
n_samples, n_features = X.shape
n_neighbors = 30


# ----------------------------------------------------------------------
# Scale and visualize the embedding vectors
def plot_embedding(X, title=None):
    x_min, x_max = np.min(X, 0), np.max(X, 0)
    X = (X - x_min) / (x_max - x_min)

    plt.figure()
    ax = plt.subplot(111)
    for i in range(X.shape[0]):
        plt.text(X[i, 0], X[i, 1], str(y[i]),
                 color=plt.cm.Set1(y[i] / 10.),
                 fontdict={'weight': 'bold', 'size': 9})

    if hasattr(offsetbox, 'AnnotationBbox'):
        # only print thumbnails with matplotlib > 1.0
        shown_images = np.array([[1., 1.]])  # just something big
        for i in range(X.shape[0]):
            dist = np.sum((X[i] - shown_images) ** 2, 1)
            if np.min(dist) < 4e-3:
                # don't show points that are too close
                continue
            shown_images = np.r_[shown_images, [X[i]]]
            imagebox = offsetbox.AnnotationBbox(
                offsetbox.OffsetImage(digits.images[i], cmap=plt.cm.gray_r),
                X[i])
            ax.add_artist(imagebox)
    plt.xticks([]), plt.yticks([])
    if title is not None:
        plt.title(title)
```

(continues on next page)

(continued from previous page)

```
# ----------------------------------------------------------------------
# Plot images of the digits
n_img_per_row = 20
img = np.zeros((10 * n_img_per_row, 10 * n_img_per_row))
for i in range(n_img_per_row):
    ix = 10 * i + 1
    for j in range(n_img_per_row):
        iy = 10 * j + 1
        img[ix:ix + 8, iy:iy + 8] = X[i * n_img_per_row + j].reshape((8, 8))

plt.imshow(img, cmap=plt.cm.binary)
plt.xticks([])
plt.yticks([])
plt.title('A selection from the 64-dimensional digits dataset')


# ----------------------------------------------------------------------
# Random 2D projection using a random unitary matrix
print("Computing random projection")
rp = random_projection.SparseRandomProjection(n_components=2, random_state=42)
X_projected = rp.fit_transform(X)
plot_embedding(X_projected, "Random Projection of the digits")


#----------------------------------------------------------------------
# Projection on to the first 2 principal components

print("Computing PCA projection")
t0 = time()
X_pca = decomposition.TruncatedSVD(n_components=2).fit_transform(X)
plot_embedding(X_pca,
               "Principal Components projection of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# Projection on to the first 2 linear discriminant components

print("Computing Linear Discriminant Analysis projection")
X2 = X.copy()
X2.flat[::X.shape[1] + 1] += 0.01  # Make X invertible
t0 = time()
X_lda = discriminant_analysis.LinearDiscriminantAnalysis(n_components=2).fit_
↪transform(X2, y)
plot_embedding(X_lda,
               "Linear Discriminant projection of the digits (time %.2fs)" %
               (time() - t0))


# ----------------------------------------------------------------------
# Isomap projection of the digits dataset
print("Computing Isomap projection")
t0 = time()
X_iso = manifold.Isomap(n_neighbors, n_components=2).fit_transform(X)
print("Done.")
plot_embedding(X_iso,
               "Isomap projection of the digits (time %.2fs)" %
```

(continues on next page)

(continued from previous page)

```
               (time() - t0))


# ----------------------------------------------------------------------
# Locally linear embedding of the digits dataset
print("Computing LLE embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='standard')
t0 = time()
X_lle = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_lle,
               "Locally Linear Embedding of the digits (time %.2fs)" %
               (time() - t0))


# ----------------------------------------------------------------------
# Modified Locally linear embedding of the digits dataset
print("Computing modified LLE embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='modified')
t0 = time()
X_mlle = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_mlle,
               "Modified Locally Linear Embedding of the digits (time %.2fs)" %
               (time() - t0))


# ----------------------------------------------------------------------
# HLLE embedding of the digits dataset
print("Computing Hessian LLE embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='hessian')
t0 = time()
X_hlle = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_hlle,
               "Hessian Locally Linear Embedding of the digits (time %.2fs)" %
               (time() - t0))


# ----------------------------------------------------------------------
# LTSA embedding of the digits dataset
print("Computing LTSA embedding")
clf = manifold.LocallyLinearEmbedding(n_neighbors, n_components=2,
                                      method='ltsa')
t0 = time()
X_ltsa = clf.fit_transform(X)
print("Done. Reconstruction error: %g" % clf.reconstruction_error_)
plot_embedding(X_ltsa,
               "Local Tangent Space Alignment of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# MDS  embedding of the digits dataset
print("Computing MDS embedding")
```

(continues on next page)

(continued from previous page)

```
clf = manifold.MDS(n_components=2, n_init=1, max_iter=2000,
                   dissimilarity='euclidean', metric=True,
                   )
t0 = time()

X_mds = clf.fit_transform(X)
print("Done. Stress: %f" % clf.stress_)
plot_embedding(X_mds,
               "MDS embedding of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# Hubness reduction (LS) + MDS  embedding of the digits dataset
print("Computing MDS embedding from local scaling neighbors graph")
clf = manifold.MDS(n_components=2, n_init=1, max_iter=2000,
                   dissimilarity='precomputed', metric=True,
                   )
t0 = time()
graph = neighbors.graph.kneighbors_graph(
    X, n_neighbors=X.shape[0]-1, mode='distance', hubness='local_scaling').toarray()
X_mds = clf.fit_transform(graph)
print("Done. Stress: %f" % clf.stress_)
plot_embedding(X_mds,
               "Hubness reduction (LS) - MDS embedding (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# Hubness reduction (MP) + MDS  embedding of the digits dataset
print("Computing MDS embedding from mutual proximity graph")
clf = manifold.MDS(n_components=2, n_init=1, max_iter=2000,
                   dissimilarity='precomputed', metric=True,
                   )
t0 = time()
graph = neighbors.graph.kneighbors_graph(
    X, n_neighbors=1082, mode='distance', hubness='mp').toarray()
X_mds = clf.fit_transform(graph)
print("Done. Stress: %f" % clf.stress_)
plot_embedding(X_mds,
               "Hubness reduction (MP) - MDS embedding (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# Random Trees embedding of the digits dataset
print("Computing Totally Random Trees embedding")
hasher = ensemble.RandomTreesEmbedding(n_estimators=200, random_state=0,
                                       max_depth=5)
t0 = time()
X_transformed = hasher.fit_transform(X)
pca = decomposition.TruncatedSVD(n_components=2)
X_reduced = pca.fit_transform(X_transformed)

plot_embedding(X_reduced,
               "Random forest embedding of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# Spectral embedding of the digits dataset
```

(continues on next page)

(continued from previous page)

```
print("Computing Spectral embedding")
embedder = manifold.SpectralEmbedding(n_components=2, random_state=0,
                                      eigen_solver="arpack")
t0 = time()
X_se = embedder.fit_transform(X)

plot_embedding(X_se,
               "Spectral embedding of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# t-SNE embedding of the digits dataset
print("Computing t-SNE embedding")
tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
t0 = time()
X_tsne = tsne.fit_transform(X)

plot_embedding(X_tsne,
               "t-SNE embedding of the digits (time %.2fs)" %
               (time() - t0))

# ----------------------------------------------------------------------
# NCA projection of the digits dataset
print("Computing NCA projection")
nca = neighbors.NeighborhoodComponentsAnalysis(n_components=2, random_state=0)
t0 = time()
X_nca = nca.fit_transform(X, y)

plot_embedding(X_nca,
               "NCA embedding of the digits (time %.2fs)" %
               (time() - t0))

plt.show()
```

Total running time of the script: (1 minutes 24.114 seconds)

CHAPTER

FOUR

API DOCUMENTATION

This is the API documentation for `scikit-hubness`.

4.1 Analysis: `skhubness.analysis`

The *skhubness.analysis* package provides methods for measuring hubness.

analysis.Hubness	Examine hubness characteristics of data.
analysis.VALID_HUBNESS_MEASURES	Built-in mutable sequence.

4.1.1 skhubness.analysis.Hubness

class skhubness.analysis.**Hubness**(*k: int = 10*, *return_value: str = 'k_skewness'*, *hub_size: float = 2.0*, *metric='euclidean'*, *store_k_neighbors: bool = False*, *store_k_occurrence: bool = False*, *algorithm: str = 'auto'*, *algorithm_params: Optional[dict] = None*, *hubness: Optional[str] = None*, *hubness_params: Optional[dict] = None*, *verbose: int = 0*, *n_jobs: int = 1*, *random_state=None*, *shuffle_equal: bool = True*)

Examine hubness characteristics of data.

Parameters

k: int Neighborhood size

return_value: str, default = "k_skewness" Hubness measure to return by *score()* By default, this is the skewness of the k-occurrence histogram. Use "all" to return a dict of all available measures, or check *skhubness.analysis.VALID_HUBNESS_MEASURE* for available measures.

hub_size: float Hubs are defined as objects with k-occurrence > hub_size * k.

metric: string, one of ['euclidean', 'cosine', 'precomputed'] Metric to use for distance computation. Currently, only Euclidean, cosine, and precomputed distances are supported.

store_k_neighbors: bool Whether to save the k-neighbor lists. Requires O(n_test * k) memory.

store_k_occurrence: bool Whether to save the k-occurrence. Requires O(n_test) memory.

algorithm: {'auto', 'hnsw', 'lsh', 'ball_tree', 'kd_tree', 'brute'}, optional Algorithm used to compute the nearest neighbors:

- 'hnsw' will use `HNSW`

- 'lsh' will use `FalconnLSH`
- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to *`fit()`* method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

algorithm_params: dict, optional Override default parameters of the NN algorithm. For example, with algorithm='lsh' and algorithm_params={n_candidates: 100} one hundred approximate neighbors are retrieved with LSH. If parameter hubness is set, the candidate neighbors are further reordered with hubness reduction. Finally, n_neighbors objects are used from the (optionally reordered) candidates.

hubness: {'mutual_proximity', 'local_scaling', 'dis_sim_local', None}, optional Hubness reduction algorithm

- 'mutual_proximity' or 'mp' will use `MutualProximity`
- 'local_scaling' or 'ls' will use `LocalScaling`
- 'dis_sim_local' or 'dsl' will use `DisSimLocal`

If None, no hubness reduction will be performed (=vanilla kNN).

hubness_params: dict, optional Override default parameters of the selected hubness reduction algorithm. For example, with hubness='mp' and hubness_params={'method': 'normal'} a mutual proximity variant is used, which models distance distributions with independent Gaussians.

random_state: int, RandomState instance or None, optional 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*.

shuffle_equal: bool, optional If true and metric='precomputed', shuffle neighbors with identical distances to avoid artifact hubness. NOTE: This is especially useful for secondary distance measures with a finite number of possible values, e.g. SNN or MP empiric.

n_jobs: int, optional Number of processes for parallel computations. - *1*: Don't use multiprocessing. - *-1*: Use all CPUs Note that not all steps are currently parallelized.

verbose: int, optional Level of output messages

References

[1], [2]

Attributes

k_skewness: float Hubness, measured as skewness of k-occurrence histogram [1]

k_skewness_truncnorm: float Hubness, measured as skewness of truncated normal distribution fitted with k-occurrence histogram

atkinson_index: float Hubness, measured as the Atkinson index of k-occurrence distribution

gini_index: float Hubness, measured as the Gini index of k-occurrence distribution

robinhood_index: float Hubness, measured as Robin Hood index of k-occurrence distribution [2]

antihubs: int Indices to antihubs

antihub_occurrence: float Proportion of antihubs in data set

hubs: int Indices to hubs

hub_occurrence: float Proportion of k-nearest neighbor slots occupied by hubs

groupie_ratio: float Proportion of objects with the largest hub in their neighborhood

k_occurrence: ndarray Reverse neighbor count for each object

k_neighbors: ndarray Indices to k-nearest neighbors for each object

__init__(*k: int = 10*, *return_value: str = 'k_skewness'*, *hub_size: float = 2.0*, *metric='euclidean'*, *store_k_neighbors: bool = False*, *store_k_occurrence: bool = False*, *algorithm: str = 'auto'*, *algorithm_params: Optional[dict] = None*, *hubness: Optional[str] = None*, *hubness_params: Optional[dict] = None*, *verbose: int = 0*, *n_jobs: int = 1*, *random_state=None*, *shuffle_equal: bool = True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([k, return_value, hub_size, ...])	Initialize self.
fit(X[, y])	Fit indexed objects.
get_params([deep])	Get parameters for this estimator.
score([X, y, has_self_distances])	Estimate hubness in a data set.
set_params(**params)	Set the parameters of this estimator.

fit(*X*, *y=None*) → skhubness.analysis.estimation.Hubness
Fit indexed objects.

Parameters

X: {array-like, sparse matrix}, shape (n_samples, n_features) or (n_query, n_indexed) if metric=='precompu
Training data vectors or distance matrix, if metric == 'precomputed'.

y: ignored

Returns

self: Fitted instance of :mod:Hubness

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 [mapping of string to any] Parameter names mapped to their values.

score(*X: Optional[numpy.ndarray] = None*, *y=None*, *has_self_distances: bool = False*) → Union[float, dict]
Estimate hubness in a data set.

Hubness is estimated from the distances between all objects in X to all objects in Y. If Y is None, all-against-all distances between the objects in X are used. If self.metric == 'precomputed', X must be a distance matrix.

Parameters

X: ndarray, shape (n_query, n_features) or (n_query, n_indexed) Array of query vectors, or distance, if self.metric == 'precomputed'

y: ignored

has_self_distances: bool, default = False Define, whether a precomputed distance matrix contains self distances, which need to be excluded.

Returns

hubness_measure: float or dict Return the hubness measure as indicated by *return_value*. Additional hubness indices are provided as attributes (e.g. `robinhood_index_()`). if return_value is 'all', a dict of all hubness measures is returned.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.1.2 skhubness.analysis.VALID_HUBNESS_MEASURES

skhubness.analysis.**VALID_HUBNESS_MEASURES = ['all', 'k_skewness', 'k_skewness_truncnorm',**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

4.2 Neighbors: skhubness.neighbors

The *skhubness.neighbors* package is a drop-in replacement for `sklearn.neighbors`, providing all of its features, while adding transparent support for hubness reduction and approximate nearest neighbor search.

neighbors.BallTree	BallTree for fast generalized N-point problems
neighbors.DistanceMetric	DistanceMetric class
neighbors.KDTree	KDTree for fast generalized N-point problems
neighbors.HNSW	Wrapper for using nmslib
neighbors.KNeighborsClassifier	Classifier implementing the k-nearest neighbors vote.
neighbors.KNeighborsRegressor	Regression based on k-nearest neighbors.
neighbors.FalconnLSH	Wrapper for using falconn LSH
neighbors.NearestCentroid	Nearest centroid classifier.

continues on next page

Table 3 – continued from previous page

neighbors.NearestNeighbors	Unsupervised learner for implementing neighbor searches.
neighbors.NNG	Wrapper for ngtpy and NNG variants.
neighbors.PuffinnLSH	Wrap Puffinn LSH for scikit-learn compatibility.
neighbors.RadiusNeighborsClassifier	Classifier implementing a vote among neighbors within a given radius
neighbors.RadiusNeighborsRegressor	Regression based on neighbors within a fixed radius.
neighbors.RandomProjectionTree	Wrapper for using annoy.AnnoyIndex
neighbors.kneighbors_graph	Computes the (weighted) graph of k-Neighbors for points in X
neighbors.radius_neighbors_graph	Computes the (weighted) graph of Neighbors for points in X
neighbors.KernelDensity	Kernel Density Estimation.
neighbors.LocalOutlierFactor	Unsupervised Outlier Detection using Local Outlier Factor (LOF)
neighbors.NeighborhoodComponentsAnalysis	Neighborhood Components Analysis

4.2.1 skhubness.neighbors.BallTree

class skhubness.neighbors.**BallTree**(*X*, *leaf_size=40*, *metric='minkowski'*, ***kwargs*)

BallTree for fast generalized N-point problems

Parameters

X [array-like of shape (n_samples, n_features)] n_samples is the number of points in the data set, and n_features is the dimension of the parameter space. Note: if X is a C-contiguous array of doubles then data will not be copied. Otherwise, an internal copy will be made.

leaf_size [positive int, default=40] Number of points at which to switch to brute-force. Changing leaf_size will not affect the results of a query, but can significantly impact the speed of a query and the memory required to store the constructed tree. The amount of memory needed to store the tree scales as approximately n_samples / leaf_size. For a specified `leaf_size`, a leaf node is guaranteed to satisfy `leaf_size <= n_points <= 2 * leaf_size`, except in the case that `n_samples < leaf_size`.

metric [str or DistanceMetric object] the distance metric to use for the tree. Default='minkowski' with p=2 (that is, a euclidean metric). See the documentation of the DistanceMetric class for a list of available metrics. ball_tree.valid_metrics gives a list of the metrics which are valid for BallTree.

Additional keywords are passed to the distance metric class.

Note: Callable functions in the metric parameter are NOT supported for KDTree

and Ball Tree. Function call overhead will result in very poor performance.

Examples

Query for k-nearest neighbors

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = BallTree(X, leaf_size=2)
>>> dist, ind = tree.query(X[:1], k=3)
>>> print(ind)  # indices of 3 closest neighbors
[0 3 1]
>>> print(dist)  # distances to 3 closest neighbors
[ 0.          0.19662693  0.29473397]
```

Pickle and Unpickle a tree. Note that the state of the tree is saved in the pickle operation: the tree needs not be rebuilt upon unpickling.

```
>>> import numpy as np
>>> import pickle
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = BallTree(X, leaf_size=2)
>>> s = pickle.dumps(tree)
>>> tree_copy = pickle.loads(s)
>>> dist, ind = tree_copy.query(X[:1], k=3)
>>> print(ind)  # indices of 3 closest neighbors
[0 3 1]
>>> print(dist)  # distances to 3 closest neighbors
[ 0.          0.19662693  0.29473397]
```

Query for neighbors within a given radius

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = BallTree(X, leaf_size=2)
>>> print(tree.query_radius(X[:1], r=0.3, count_only=True))
3
>>> ind = tree.query_radius(X[:1], r=0.3)
>>> print(ind)  # indices of neighbors within distance 0.3
[3 0 1]
```

Compute a gaussian kernel density estimate:

```
>>> import numpy as np
>>> rng = np.random.RandomState(42)
>>> X = rng.random_sample((100, 3))
>>> tree = BallTree(X)
>>> tree.kernel_density(X[:3], h=0.1, kernel='gaussian')
array([ 6.94114649,  7.83281226,  7.2071716 ])
```

Compute a two-point auto-correlation function

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((30, 3))
>>> r = np.linspace(0, 1, 5)
>>> tree = BallTree(X)
```

(continues on next page)

(continued from previous page)

```
>>> tree.two_point_correlation(X, r)
array([ 30,  62, 278, 580, 820])
```

Attributes

data [memory view] The training data

__init__ (**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*args, **kwargs)	Initialize self.
get_arrays(self)	Get data and node arrays.
get_n_calls(self)	Get number of calls.
get_tree_stats(self)	Get tree status.
kernel_density(self, X, h[, kernel, atol, ...])	Compute the kernel density estimate at points X with the given kernel, using the distance metric specified at tree creation.
query(X[, k, return_distance, dualtree, ...])	query the tree for the k nearest neighbors
query_radius(X, r[, return_distance, ...])	query the tree for neighbors within a radius r
reset_n_calls(self)	Reset number of calls to 0.
two_point_correlation(X, r[, dualtree])	Compute the two-point correlation function

Attributes

data
idx_array
node_bounds
node_data
sample_weight
sum_weight
valid_metrics

get_arrays (*self*)
Get data and node arrays.

Returns

arrays: tuple of array Arrays for storing tree data, index, node data and node bounds.

get_n_calls (*self*)
Get number of calls.

Returns

n_calls: int number of distance computation calls

get_tree_stats (*self*)
Get tree status.

Returns

tree_stats: tuple of int (number of trims, number of leaves, number of splits)

kernel_density(*self*, *X*, *h*, *kernel='gaussian'*, *atol=0*, *rtol=1e-08*, *breadth_first=True*, *return_log=False*)

Compute the kernel density estimate at points X with the given kernel, using the distance metric specified at tree creation.

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query. Last dimension should match dimension of training data.

h [float] the bandwidth of the kernel

kernel [str, default="gaussian"] specify the kernel to use. Options are - 'gaussian' - 'tophat' - 'epanechnikov' - 'exponential' - 'linear' - 'cosine' Default is kernel = 'gaussian'

atol, rtol [float, default=0, 1e-8] Specify the desired relative and absolute tolerance of the result. If the true result is K_true, then the returned result K_ret satisfies `abs(K_true - K_ret) < atol + rtol * K_ret` The default is zero (i.e. machine precision) for both.

breadth_first [bool, default=False] If True, use a breadth-first search. If False (default) use a depth-first search. Breadth-first is generally faster for compact kernels and/or high tolerances.

return_log [bool, default=False] Return the logarithm of the result. This can be more accurate than returning the result itself for narrow kernels.

Returns

density [ndarray of shape X.shape[:-1]] The array of (log)-density evaluations

query(*X*, *k=1*, *return_distance=True*, *dualtree=False*, *breadth_first=False*)

query the tree for the k nearest neighbors

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query

k [int, default=1] The number of nearest neighbors to return

return_distance [bool, default=True] if True, return a tuple (d, i) of distances and indices if False, return array i

dualtree [bool, default=False] if True, use the dual tree formalism for the query: a tree is built for the query points, and the pair of trees is used to efficiently search this space. This can lead to better performance as the number of points grows large.

breadth_first [bool, default=False] if True, then query the nodes in a breadth-first manner. Otherwise, query the nodes in a depth-first manner.

sort_results [bool, default=True] if True, then distances and indices of each point are sorted on return, so that the first column contains the closest points. Otherwise, neighbors are returned in an arbitrary order.

Returns

i [if return_distance == False]

(d,i) [if return_distance == True]

d [ndarray of shape X.shape[:-1] + k, dtype=double] Each entry gives the list of distances to the neighbors of the corresponding point.

i [ndarray of shape X.shape[:-1] + k, dtype=int] Each entry gives the list of indices of neighbors of the corresponding point.

query_radius(*X*, *r*, *return_distance=False*, *count_only=False*, *sort_results=False*)
query the tree for neighbors within a radius r

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query

r [distance within which neighbors are returned] r can be a single value, or an array of values of shape x.shape[:-1] if different radii are desired for each point.

return_distance [bool, default=False] if True, return distances to neighbors of each point if False, return only neighbors Note that unlike the query() method, setting return_distance=True here adds to the computation time. Not all distances need to be calculated explicitly for return_distance=False. Results are not sorted by default: see `sort_results` keyword.

count_only [bool, default=False] if True, return only the count of points within distance r if False, return the indices of all points within distance r If return_distance==True, setting count_only=True will result in an error.

sort_results [bool, default=False] if True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. If return_distance == False, setting sort_results = True will result in an error.

Returns

count [if count_only == True]

ind [if count_only == False and return_distance == False]

(ind, dist) [if count_only == False and return_distance == True]

count [ndarray of shape X.shape[:-1], dtype=int] Each entry gives the number of neighbors within a distance r of the corresponding point.

ind [ndarray of shape X.shape[:-1], dtype=object] Each element is a numpy integer array listing the indices of neighbors of the corresponding point. Note that unlike the results of a k-neighbors query, the returned neighbors are not sorted by distance by default.

dist [ndarray of shape X.shape[:-1], dtype=object] Each element is a numpy double array listing the distances corresponding to indices in i.

reset_n_calls(*self*)
Reset number of calls to 0.

two_point_correlation(*X*, *r*, *dualtree=False*)
Compute the two-point correlation function

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query. Last dimension should match dimension of training data.

r [array-like] A one-dimensional array of distances

dualtree [bool, default=False] If True, use a dualtree algorithm. Otherwise, use a single-tree algorithm. Dual tree algorithms can have better scaling for large N.

Returns

counts [ndarray] counts[i] contains the number of pairs of points with distance less than or equal to r[i]

4.2.2 skhubness.neighbors.DistanceMetric

class skhubness.neighbors.**DistanceMetric**

DistanceMetric class

This class provides a uniform interface to fast distance metric functions. The various metrics can be accessed via the *get_metric()* class method and the metric string identifier (see below).

Examples

```
>>> from sklearn.neighbors import DistanceMetric
>>> dist = DistanceMetric.get_metric('euclidean')
>>> X = [[0, 1, 2],
         [3, 4, 5]]
>>> dist.pairwise(X)
array([[ 0.        ,  5.19615242],
       [ 5.19615242,  0.        ]])
```

Available Metrics

The following lists the string metric identifiers and the associated distance metric classes:

Metrics intended for real-valued vector spaces:

identifier	class name	args	distance function
“euclidean”	EuclideanDistance	•	sqrt(sum((x - y)^2))
“manhattan”	ManhattanDistance	•	sum(|x - y|)
“chebyshev”	ChebyshevDistance	•	max(|x - y|)
“minkowski”	MinkowskiDistance	p	sum(|x - y|^p)^(1/p)
“wminkowski”	WMinkowskiDistance	p, w	sum(|w * (x - y)|^p)^(1/p)
“seuclidean”	SEuclideanDistance	V	sqrt(sum((x - y)^2 / V))
“mahalanobis”	MahalanobisDistance	V or VI	sqrt((x - y)' V^-1 (x - y))

Metrics intended for two-dimensional vector spaces: Note that the haversine distance metric requires data in the form of [latitude, longitude] and both inputs and outputs are in units of radians.

identifier	class name	distance function
“haversine”	HaversineDistance	2 arcsin(sqrt(sin^2(0.5*dx) + cos(x1)cos(x2)sin^2(0.5*dy)))

Metrics intended for integer-valued vector spaces: Though intended for integer-valued vectors, these are also valid metrics in the case of real-valued vectors.

identifier	class name	distance function
“hamming”	HammingDistance	`N_unequal(x, y) / N_tot`
“canberra”	CanberraDistance	`sum(|x - y| / (|x| + |y|))`
“braycurtis”	BrayCurtisDistance	`sum(|x - y|) / (sum(|x|) + sum(|y|))`

Metrics intended for boolean-valued vector spaces: Any nonzero entry is evaluated to “True”. In the listings below, the following abbreviations are used:

- N : number of dimensions
- NTT : number of dims in which both values are True
- NTF : number of dims in which the first value is True, second is False
- NFT : number of dims in which the first value is False, second is True
- NFF : number of dims in which both values are False
- NNEQ : number of non-equal dimensions, NNEQ = NTF + NFT
- NNZ : number of nonzero dimensions, NNZ = NTF + NFT + NTT

identifier	class name	distance function
“jaccard”	JaccardDistance	NNEQ / NNZ
“matching”	MatchingDistance	NNEQ / N
“dice”	DiceDistance	NNEQ / (NTT + NNZ)
“kulsinski”	KulsinskiDistance	(NNEQ + N - NTT) / (NNEQ + N)
“rogerstanimoto”	RogersTanimotoDistance	2 * NNEQ / (N + NNEQ)
“russellrao”	RussellRaoDistance	NNZ / N
“sokalmichener”	SokalMichenerDistance	2 * NNEQ / (N + NNEQ)
“sokalsneath”	SokalSneathDistance	NNEQ / (NNEQ + 0.5 * NTT)

User-defined distance:

identifier	class name	args
“pyfunc”	PyFuncDistance	func

Here `func` is a function which takes two one-dimensional numpy arrays, and returns a distance. Note that in order to be used within the BallTree, the distance must be a true metric: i.e. it must satisfy the following properties

1) Non-negativity: d(x, y) >= 0
2) Identity: d(x, y) = 0 if and only if x == y
3) Symmetry: d(x, y) = d(y, x)
4) Triangle Inequality: d(x, y) + d(y, z) >= d(x, z)

Because of the Python object overhead involved in calling the python function, this will be fairly slow, but it will have the same scaling as other distances.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*args, **kwargs)	Initialize self.
dist_to_rdist	Convert the true distance to the reduced distance.
get_metric	Get the given distance metric from the string identifier.
pairwise	Compute the pairwise distances between X and Y
rdist_to_dist	Convert the Reduced distance to the true distance.

dist_to_rdist()
Convert the true distance to the reduced distance.

The reduced distance, defined for some metrics, is a computationally more efficient measure which preserves the rank of the true distance. For example, in the Euclidean distance metric, the reduced distance is the squared-euclidean distance.

get_metric()
Get the given distance metric from the string identifier.

See the docstring of DistanceMetric for a list of available metrics.

Parameters

metric [string or class name] The distance metric to use

****kwargs** additional arguments will be passed to the requested metric

pairwise()
Compute the pairwise distances between X and Y

This is a convenience routine for the sake of testing. For many metrics, the utilities in scipy.spatial.distance.cdist and scipy.spatial.distance.pdist will be faster.

Parameters

X [array_like] Array of shape (Nx, D), representing Nx points in D dimensions.

Y [array_like (optional)] Array of shape (Ny, D), representing Ny points in D dimensions. If not specified, then Y=X.

Returns

———-

dist [ndarray] The shape (Nx, Ny) array of pairwise distances between points in X and Y.

rdist_to_dist()
Convert the Reduced distance to the true distance.

The reduced distance, defined for some metrics, is a computationally more efficient measure which preserves the rank of the true distance. For example, in the Euclidean distance metric, the reduced distance is the squared-euclidean distance.

4.2.3 skhubness.neighbors.KDTree

class skhubness.neighbors.**KDTree**(*X*, *leaf_size=40*, *metric='minkowski'*, ***kwargs*)

KDTree for fast generalized N-point problems

Parameters

X [array-like of shape (n_samples, n_features)] n_samples is the number of points in the data set, and n_features is the dimension of the parameter space. Note: if X is a C-contiguous array of doubles then data will not be copied. Otherwise, an internal copy will be made.

leaf_size [positive int, default=40] Number of points at which to switch to brute-force. Changing leaf_size will not affect the results of a query, but can significantly impact the speed of a query and the memory required to store the constructed tree. The amount of memory needed to store the tree scales as approximately n_samples / leaf_size. For a specified `leaf_size`, a leaf node is guaranteed to satisfy `leaf_size <= n_points <= 2 * leaf_size`, except in the case that `n_samples < leaf_size`.

metric [str or DistanceMetric object] the distance metric to use for the tree. Default='minkowski' with p=2 (that is, a euclidean metric). See the documentation of the DistanceMetric class for a list of available metrics. kd_tree.valid_metrics gives a list of the metrics which are valid for KDTree.

Additional keywords are passed to the distance metric class.

Note: Callable functions in the metric parameter are NOT supported for KDTree

and Ball Tree. Function call overhead will result in very poor performance.

Examples

Query for k-nearest neighbors

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = KDTree(X, leaf_size=2)
>>> dist, ind = tree.query(X[:1], k=3)
>>> print(ind)  # indices of 3 closest neighbors
[0 3 1]
>>> print(dist)  # distances to 3 closest neighbors
[ 0.          0.19662693  0.29473397]
```

Pickle and Unpickle a tree. Note that the state of the tree is saved in the pickle operation: the tree needs not be rebuilt upon unpickling.

```
>>> import numpy as np
>>> import pickle
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = KDTree(X, leaf_size=2)
>>> s = pickle.dumps(tree)
>>> tree_copy = pickle.loads(s)
>>> dist, ind = tree_copy.query(X[:1], k=3)
>>> print(ind)  # indices of 3 closest neighbors
[0 3 1]
>>> print(dist)  # distances to 3 closest neighbors
[ 0.          0.19662693  0.29473397]
```

Query for neighbors within a given radius

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((10, 3))  # 10 points in 3 dimensions
>>> tree = KDTree(X, leaf_size=2)
>>> print(tree.query_radius(X[:1], r=0.3, count_only=True))
3
>>> ind = tree.query_radius(X[:1], r=0.3)
>>> print(ind)  # indices of neighbors within distance 0.3
[3 0 1]
```

Compute a gaussian kernel density estimate:

```
>>> import numpy as np
>>> rng = np.random.RandomState(42)
>>> X = rng.random_sample((100, 3))
>>> tree = KDTree(X)
>>> tree.kernel_density(X[:3], h=0.1, kernel='gaussian')
array([ 6.94114649,  7.83281226,  7.2071716 ])
```

Compute a two-point auto-correlation function

```
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> X = rng.random_sample((30, 3))
>>> r = np.linspace(0, 1, 5)
>>> tree = KDTree(X)
>>> tree.two_point_correlation(X, r)
array([ 30,  62, 278, 580, 820])
```

Attributes

data [memory view] The training data

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*args, **kwargs)	Initialize self.
get_arrays(self)	Get data and node arrays.
get_n_calls(self)	Get number of calls.
get_tree_stats(self)	Get tree status.
kernel_density(self, X, h[, kernel, atol, ...])	Compute the kernel density estimate at points X with the given kernel, using the distance metric specified at tree creation.
query(X[, k, return_distance, dualtree, ...])	query the tree for the k nearest neighbors
query_radius(X, r[, return_distance, ...])	query the tree for neighbors within a radius r
reset_n_calls(self)	Reset number of calls to 0.
two_point_correlation(X, r[, dualtree])	Compute the two-point correlation function

Attributes

data
idx_array
node_bounds
node_data
sample_weight
sum_weight
valid_metrics

get_arrays(*self*)
Get data and node arrays.

Returns

arrays: tuple of array Arrays for storing tree data, index, node data and node bounds.

get_n_calls(*self*)
Get number of calls.

Returns

n_calls: int number of distance computation calls

get_tree_stats(*self*)
Get tree status.

Returns

tree_stats: tuple of int (number of trims, number of leaves, number of splits)

kernel_density(*self*, *X*, *h*, *kernel='gaussian'*, *atol=0*, *rtol=1e-08*, *breadth_first=True*, *return_log=False*)
Compute the kernel density estimate at points X with the given kernel, using the distance metric specified at tree creation.

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query. Last dimension should match dimension of training data.

h [float] the bandwidth of the kernel

kernel [str, default="gaussian"] specify the kernel to use. Options are - 'gaussian' - 'tophat' - 'epanechnikov' - 'exponential' - 'linear' - 'cosine' Default is kernel = 'gaussian'

atol, rtol [float, default=0, 1e-8] Specify the desired relative and absolute tolerance of the result. If the true result is K_true, then the returned result K_ret satisfies `abs(K_true - K_ret) < atol + rtol * K_ret` The default is zero (i.e. machine precision) for both.

breadth_first [bool, default=False] If True, use a breadth-first search. If False (default) use a depth-first search. Breadth-first is generally faster for compact kernels and/or high tolerances.

return_log [bool, default=False] Return the logarithm of the result. This can be more accurate than returning the result itself for narrow kernels.

Returns

density [ndarray of shape X.shape[:-1]] The array of (log)-density evaluations

query (*X*, *k=1*, *return_distance=True*, *dualtree=False*, *breadth_first=False*)
query the tree for the k nearest neighbors

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query

k [int, default=1] The number of nearest neighbors to return

return_distance [bool, default=True] if True, return a tuple (d, i) of distances and indices if False, return array i

dualtree [bool, default=False] if True, use the dual tree formalism for the query: a tree is built for the query points, and the pair of trees is used to efficiently search this space. This can lead to better performance as the number of points grows large.

breadth_first [bool, default=False] if True, then query the nodes in a breadth-first manner. Otherwise, query the nodes in a depth-first manner.

sort_results [bool, default=True] if True, then distances and indices of each point are sorted on return, so that the first column contains the closest points. Otherwise, neighbors are returned in an arbitrary order.

Returns

i [if return_distance == False]

(d,i) [if return_distance == True]

d [ndarray of shape X.shape[:-1] + k, dtype=double] Each entry gives the list of distances to the neighbors of the corresponding point.

i [ndarray of shape X.shape[:-1] + k, dtype=int] Each entry gives the list of indices of neighbors of the corresponding point.

query_radius (*X*, *r*, *return_distance=False*, *count_only=False*, *sort_results=False*)
query the tree for neighbors within a radius r

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query

r [distance within which neighbors are returned] r can be a single value, or an array of values of shape x.shape[:-1] if different radii are desired for each point.

return_distance [bool, default=False] if True, return distances to neighbors of each point if False, return only neighbors Note that unlike the query() method, setting return_distance=True here adds to the computation time. Not all distances need to be calculated explicitly for return_distance=False. Results are not sorted by default: see `sort_results` keyword.

count_only [bool, default=False] if True, return only the count of points within distance r if False, return the indices of all points within distance r If return_distance==True, setting count_only=True will result in an error.

sort_results [bool, default=False] if True, the distances and indices will be sorted before being returned. If False, the results will not be sorted. If return_distance == False, setting sort_results = True will result in an error.

Returns

count [if count_only == True]

ind [if count_only == False and return_distance == False]

(ind, dist) [if count_only == False and return_distance == True]

count [ndarray of shape X.shape[:-1], dtype=int] Each entry gives the number of neighbors within a distance r of the corresponding point.

ind [ndarray of shape X.shape[:-1], dtype=object] Each element is a numpy integer array listing the indices of neighbors of the corresponding point. Note that unlike the results of a k-neighbors query, the returned neighbors are not sorted by distance by default.

dist [ndarray of shape X.shape[:-1], dtype=object] Each element is a numpy double array listing the distances corresponding to indices in i.

reset_n_calls(*self*)
Reset number of calls to 0.

two_point_correlation(*X*, *r*, *dualtree=False*)
Compute the two-point correlation function

Parameters

X [array-like of shape (n_samples, n_features)] An array of points to query. Last dimension should match dimension of training data.

r [array-like] A one-dimensional array of distances

dualtree [bool, default=False] If True, use a dualtree algorithm. Otherwise, use a single-tree algorithm. Dual tree algorithms can have better scaling for large N.

Returns

counts [ndarray] counts[i] contains the number of pairs of points with distance less than or equal to r[i]

4.2.4 skhubness.neighbors.HNSW

class skhubness.neighbors.**HNSW**(*n_candidates: int = 5*, *metric: str = 'euclidean'*, *method: str = 'hnsw'*, *post_processing: int = 2*, *n_jobs: int = 1*, *verbose: int = 0*)
Wrapper for using nmslib

Hierarchical navigable small-world graphs are data structures, that allow for approximate nearest neighbor search. Here, an implementation from nmslib is used.

Parameters

n_candidates: int, default = 5 Number of neighbors to retrieve

metric: str, default = 'euclidean' Distance metric, allowed are "angular", "euclidean", "manhattan", "hamming", "dot"

method: str, default = 'hnsw', ANN method to use. Currently, only 'hnsw' is supported.

post_processing: int, default = 2 More post processing means longer index creation, and higher retrieval accuracy.

n_jobs: int, default = 1 Number of parallel jobs

verbose: int, default = 0 Verbosity level. If verbose >= 2, show progress bar on indexing.

Attributes

valid_metrics: List of valid distance metrics/measures

__init__ (*n_candidates: int = 5*, *metric: str = 'euclidean'*, *method: str = 'hnsw'*, *post_processing: int = 2*, *n_jobs: int = 1*, *verbose: int = 0*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_candidates, metric, method, ...])	Initialize self.
fit(X[, y])	Setup the HNSW index from training data.
kneighbors([X, n_candidates, return_distance])	Retrieve k nearest neighbors.

Attributes

valid_metrics

fit (*X*, *y=None*) → skhubness.neighbors.hnsw.HNSW
Setup the HNSW index from training data.

Parameters

X: np.array Data to be indexed

y: any Ignored

Returns

self: HNSW An instance of HNSW with a built graph

kneighbors (*X: Optional[numpy.ndarray] = None*, *n_candidates: Optional[int] = None*, *return_distance: bool = True*) → Union[Tuple[numpy.array, numpy.array], numpy.array]
Retrieve k nearest neighbors.

Parameters

X: np.array or None, optional, default = None Query objects. If None, search among the indexed objects.

n_candidates: int or None, optional, default = None Number of neighbors to retrieve. If None, use the value passed during construction.

return_distance: bool, default = True If return_distance, will return distances and indices to neighbors. Else, only return the indices.

4.2.5 skhubness.neighbors.KNeighborsClassifier

class skhubness.neighbors.**KNeighborsClassifier** (*n_neighbors: int = 5*, *weights: str = 'uniform'*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *leaf_size: int = 30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, *verbose: int = 0*, ***kwargs*)

Classifier implementing the k-nearest neighbors vote.

Read more in the scikit-learn User Guide

Parameters

n_neighbors: int, optional (default = 5) Number of neighbors to use by default for *kneighbors()* queries.

weights: str or callable, optional (default = 'uniform') weight function used in prediction. Possible values:

- 'uniform': uniform weights. All points in each neighborhood are weighted equally.
- 'distance': weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable]: a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

algorithm [{'auto', 'hnsw', 'lsh', 'falconn_lsh', 'nng', 'rptree', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'hnsw' will use *HNSW*
- 'lsh' will use *PuffinnLSH*
- 'falconn_lsh' will use *FalconnLSH*
- 'nng' will use *NNG*
- 'rptree' will use *RandomProjectionTree*
- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate exact algorithm based on the values passed to *fit()* method. This will not select an approximate nearest neighbor algorithm.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

algorithm_params: dict, optional Override default parameters of the NN algorithm. For example, with algorithm='lsh' and algorithm_params={n_candidates: 100} one hundred approximate neighbors are retrieved with LSH. If parameter hubness is set, the candidate neighbors are further reordered with hubness reduction. Finally, n_neighbors objects are used from the (optionally reordered) candidates.

hubness: {'mutual_proximity', 'local_scaling', 'dis_sim_local', None}, optional Hubness reduction algorithm

- 'mutual_proximity' or 'mp' will use `MutualProximity`
- 'local_scaling' or 'ls' will use `LocalScaling`
- 'dis_sim_local' or 'dsl' will use `DisSimLocal`

If None, no hubness reduction will be performed (=vanilla kNN).

hubness_params: dict, optional Override default parameters of the selected hubness reduction algorithm. For example, with hubness='mp' and hubness_params={'method': 'normal'} a mutual proximity variant is used, which models distance distributions with independent Gaussians.

leaf_size: int, optional (default = 30) Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p: integer, optional (default = 2) Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric: string or callable, default 'minkowski' the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics.

metric_params: dict, optional (default = None) Additional keyword arguments for the metric function.

n_jobs: int or None, optional (default=None) The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details. Doesn't affect *`fit()`* method.

See also:

RadiusNeighborsClassifier

KNeighborsRegressor

RadiusNeighborsRegressor

NearestNeighbors

Notes

See Nearest Neighbors in the scikit-learn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

Warning: Regarding the Nearest Neighbors algorithms, if it is found that two neighbors, neighbor *k+1* and *k*, have identical distances but different labels, the results will depend on the ordering of the training data.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from skhubness.neighbors import KNeighborsClassifier
>>> neigh = KNeighborsClassifier(n_neighbors=3)
>>> neigh.fit(X, y)
KNeighborsClassifier(...)
>>> print(neigh.predict([[1.1]]))
[0]
>>> print(neigh.predict_proba([[0.9]]))
[[0.66666667 0.33333333]]
```

__init__ (*n_neighbors: int = 5*, *weights: str = 'uniform'*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *leaf_size: int = 30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, *verbose: int = 0*, ***kwargs*)

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

Methods

__init__([n_neighbors, weights, algorithm, ...])	Initialize self.
fit(X, y)	Fit the model using X as training data and y as target values
get_params([deep])	Get parameters for this estimator.
kcandidates([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors([X, n_neighbors, return_distance])	TODO
kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
predict(X)	Predict the class labels for the provided data
predict_proba(X)	Return probability estimates for the test data X.
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

fit (*X*, *y*)
Fit the model using X as training data and y as target values

Parameters

X [{array-like, sparse matrix, BallTree, KDTree, HNSW, FalconnLSH, PuffinLSH, NNG, RandomProjectionTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

y [{array-like, sparse matrix}] Target values of shape = [n_samples] or [n_samples, n_outputs]

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 [mapping of string to any] Parameter names mapped to their values.

kcandidates (*X=None*, *n_neighbors=None*, *return_distance=True*) → numpy.ndarray
Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array] Array representing the lengths to points, only present if return_distance=True

ind [array] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from skhubness.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

kneighbors (*X=None*, *n_neighbors=None*, *return_distance=True*)
TODO

kneighbors_graph (*X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

predict(*X*)

Predict the class labels for the provided data

Parameters

X: array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed' Test samples.

Returns

y: array of shape [n_samples] or [n_samples, n_outputs] Class labels for each data sample.

predict_proba(*X*)

Return probability estimates for the test data X.

Parameters

X: array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed' Test samples.

Returns

p: array of shape = [n_samples, n_classes], or a list of n_outputs of such arrays if n_outputs > 1. The class probabilities of the input samples. Classes are ordered by lexicographic order.

score(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

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

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.6 skhubness.neighbors.KNeighborsRegressor

class skhubness.neighbors.**KNeighborsRegressor**(*n_neighbors=5*, *weights='uniform'*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

Regression based on k-nearest neighbors.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Read more in the scikit-learn User Guide.

Parameters

n_neighbors: int, optional (default = 5) Number of neighbors to use by default for *kneighbors()* queries.

weights: str or callable weight function used in prediction. Possible values:

- 'uniform': uniform weights. All points in each neighborhood are weighted equally.
- 'distance': weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable]: a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm [{'auto', 'hnsw', 'lsh', 'falconn_lsh', 'nng', 'rptree', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'hnsw' will use *HNSW*
- 'lsh' will use *PuffinnLSH*
- 'falconn_lsh' will use *FalconnLSH*
- 'nng' will use *NNG*
- 'rptree' will use *RandomProjectionTree*
- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate exact algorithm based on the values passed to *fit()* method. This will not select an approximate nearest neighbor algorithm.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

algorithm_params: dict, optional Override default parameters of the NN algorithm. For example, with algorithm='lsh' and algorithm_params={n_candidates: 100} one hundred approximate neighbors are retrieved with LSH. If parameter hubness is set, the candidate neighbors are further reordered with hubness reduction. Finally, n_neighbors objects are used from the (optionally reordered) candidates.

hubness: {'mutual_proximity', 'local_scaling', 'dis_sim_local', None}, optional Hubness reduction algorithm

- 'mutual_proximity' or 'mp' will use `MutualProximity`
- 'local_scaling' or 'ls' will use `LocalScaling`
- 'dis_sim_local' or 'dsl' will use `DisSimLocal`

If None, no hubness reduction will be performed (=vanilla kNN).

hubness_params: dict, optional Override default parameters of the selected hubness reduction algorithm. For example, with hubness='mp' and hubness_params={'method': 'normal'} a mutual proximity variant is used, which models distance distributions with independent Gaussians.

leaf_size: int, optional (default = 30) Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p: integer, optional (default = 2) Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric: string or callable, default 'minkowski' the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics.

metric_params: dict, optional (default = None) Additional keyword arguments for the metric function.

n_jobs: int or None, optional (default=None) The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learn Glossary for more details. Doesn't affect *`fit()`* method.

See also:

NearestNeighbors

RadiusNeighborsRegressor

KNeighborsClassifier

RadiusNeighborsClassifier

Notes

See Nearest Neighbors in the scikit-learn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

Warning: Regarding the Nearest Neighbors algorithms, if it is found that two neighbors, neighbor *k+1* and *k*, have identical distances but different labels, the results will depend on the ordering of the training data.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from skhubness.neighbors import KNeighborsRegressor
>>> neigh = KNeighborsRegressor(n_neighbors=2)
>>> neigh.fit(X, y)
KNeighborsRegressor(...)
>>> print(neigh.predict([[1.5]]))
[0.5]
```

__init__ (*n_neighbors=5*, *weights='uniform'*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_neighbors, weights, algorithm, ...])	Initialize self.
fit(X, y)	Fit the model using X as training data and y as target values
get_params([deep])	Get parameters for this estimator.
kcandidates([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors([X, n_neighbors, return_distance])	TODO
kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
predict(X)	Predict the target for the provided data
score(X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

fit (*X*, *y*)
Fit the model using X as training data and y as target values

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

y [{array-like, sparse matrix}]

Target values, array of float values, shape = [n_samples] or [n_samples, n_outputs]

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 [mapping of string to any] Parameter names mapped to their values.

kcandidates (*X=None*, *n_neighbors=None*, *return_distance=True*) → numpy.ndarray
Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array] Array representing the lengths to points, only present if return_distance=True

ind [array] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from skhubness.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

kneighbors (*X=None*, *n_neighbors=None*, *return_distance=True*)
TODO

kneighbors_graph (*X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

predict(*X*)

Predict the target for the provided data

Parameters

X: array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed' Test samples.

Returns

y: array of int, shape = [n_samples] or [n_samples, n_outputs] Target values

score(*X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.7 skhubness.neighbors.FalconnLSH

class skhubness.neighbors.**FalconnLSH**(*n_candidates: int = 5*, *radius: float = 1.0*, *metric: str = 'euclidean'*, *num_probes: int = 50*, *n_jobs: int = 1*, *verbose: int = 0*)

Wrapper for using falconn LSH

Falconn is an approximate nearest neighbor library, that uses multiprobe locality-sensitive hashing.

Parameters

n_candidates: int, default = 5 Number of neighbors to retrieve

radius: float or None, optional, default = None Retrieve neighbors within this radius. Can be negative: See Notes.

metric: str, default = 'euclidean' Distance metric, allowed are "angular", "euclidean", "manhattan", "hamming", "dot"

num_probes: int, default = 50 The number of buckets the query algorithm probes. The higher number of probes is, the better accuracy one gets, but the slower queries are.

n_jobs: int, default = 1 Number of parallel jobs

verbose: int, default = 0 Verbosity level. If verbose > 0, show tqdm progress bar on indexing and querying.

Notes

From the falconn docs: radius can be negative, and for the distance function 'negative_inner_product' it actually makes sense.

Attributes

valid_metrics: List of valid distance metrics/measures

__init__ (*n_candidates: int = 5*, *radius: float = 1.0*, *metric: str = 'euclidean'*, *num_probes: int = 50*, *n_jobs: int = 1*, *verbose: int = 0*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_candidates, radius, metric, ...])	Initialize self.
fit(X[, y])	Setup the LSH index from training data.
kneighbors([X, n_candidates, return_distance])	Retrieve k nearest neighbors.
radius_neighbors([X, radius, return_distance])	Retrieve neighbors within a certain radius.

Attributes

valid_metrics

fit (*X: numpy.ndarray*, *y: Optional[numpy.ndarray] = None*) → skhubness.neighbors.lsh.FalconnLSH
Setup the LSH index from training data.

Parameters

X: np.array Data to be indexed

y: any Ignored

Returns

self: FalconnLSH An instance of LSH with a built index

kneighbors (*X: Optional[numpy.ndarray] = None*, *n_candidates: Optional[int] = None*, *return_distance: bool = True*) → Union[Tuple[numpy.array, numpy.array], numpy.array]
Retrieve k nearest neighbors.

Parameters

X: np.array or None, optional, default = None Query objects. If None, search among the indexed objects.

n_candidates: int or None, optional, default = None Number of neighbors to retrieve. If None, use the value passed during construction.

return_distance: bool, default = True If return_distance, will return distances and indices to neighbors. Else, only return the indices.

radius_neighbors (*X: Optional[numpy.ndarray] = None*, *radius: Optional[float] = None*, *return_distance: bool = True*) → Union[Tuple[numpy.array, numpy.array], numpy.array]
Retrieve neighbors within a certain radius.

Parameters

X: np.array or None, optional, default = None Query objects. If None, search among the indexed objects.

radius: float or None, optional, default = None Retrieve neighbors within this radius. Can be negative: See Notes.

return_distance: bool, default = True If return_distance, will return distances and indices to neighbors. Else, only return the indices.

Notes

From the falconn docs: radius can be negative, and for the distance function 'negative_inner_product' it actually makes sense.

4.2.8 skhubness.neighbors.NearestCentroid

class skhubness.neighbors.**NearestCentroid**(***kwargs*)

Nearest centroid classifier.

Each class is represented by its centroid, with test samples classified to the class with the nearest centroid.

Read more in the scikit-learn User Guide.

Parameters

metric [str or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by metrics.pairwise.pairwise_distances for its metric parameter. The centroids for the samples corresponding to each class is the point from which the sum of the distances (according to the metric) of all samples that belong to that particular class are minimized. If the "manhattan" metric is provided, this centroid is the median and for all other metrics, the centroid is now set to be the mean.

Changed in version 0.19: `metric='precomputed'` was deprecated and now raises an error

shrink_threshold [float, default=None] Threshold for shrinking centroids to remove features.

See also:

sklearn.neighbors.KNeighborsClassifier nearest neighbors classifier

Notes

When used for text classification with tf-idf vectors, this classifier is also known as the Rocchio classifier.

References

Tibshirani, R., Hastie, T., Narasimhan, B., & Chu, G. (2002). Diagnosis of multiple cancer types by shrunken centroids of gene expression. Proceedings of the National Academy of Sciences of the United States of America, 99(10), 6567-6572. The National Academy of Sciences.

Examples

```
>>> from sklearn.neighbors import NearestCentroid
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> clf = NearestCentroid()
>>> clf.fit(X, y)
NearestCentroid()
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Attributes

centroids_ [array-like of shape (n_classes, n_features)] Centroid of each class.

classes_ [array of shape (n_classes,)] The unique classes labels.

__init__ (*metric='euclidean'*, ***, *shrink_threshold=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([metric, shrink_threshold])	Initialize self.
fit(X, y)	Fit the NearestCentroid model according to the given training data.
get_params([deep])	Get parameters for this estimator.
predict(X)	Perform classification on an array of test vectors X.
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

fit (*X*, *y*)
Fit the NearestCentroid model according to the given training data.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] Training vector, where n_samples is the number of samples and n_features is the number of features. Note that centroid shrinking cannot be used with sparse matrices.

y [array-like of shape (n_samples,)] Target values (integers)

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 [mapping of string to any] Parameter names mapped to their values.

predict(*X*)

Perform classification on an array of test vectors X.

The predicted class C for each sample in X is returned.

Parameters

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

Returns

C [ndarray of shape (n_samples,)]

Notes

If the metric constructor parameter is "precomputed", X is assumed to be the distance matrix between the data to be predicted and `self.centroids_`.

score(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

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

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.9 skhubness.neighbors.NearestNeighbors

class skhubness.neighbors.**NearestNeighbors**(*n_neighbors=5*, *radius=1.0*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

Unsupervised learner for implementing neighbor searches.

Read more in the scikit-learn User Guide

Parameters

n_neighbors: int, optional (default = 5) Number of neighbors to use by default for *kneighbors()* queries.

radius: float, optional (default = 1.0) Range of parameter space to use by default for *radius_neighbors()* queries.

algorithm [{'auto', 'hnsw', 'lsh', 'falconn_lsh', 'nng', 'rptree', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'hnsw' will use *HNSW*
- 'lsh' will use *PuffinnLSH*
- 'falconn_lsh' will use *FalconnLSH*
- 'nng' will use *NNG*
- 'rptree' will use *RandomProjectionTree*
- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate exact algorithm based on the values passed to *fit()* method. This will not select an approximate nearest neighbor algorithm.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

algorithm_params: dict, optional Override default parameters of the NN algorithm. For example, with algorithm='lsh' and algorithm_params={n_candidates: 100} one hundred approximate neighbors are retrieved with LSH. If parameter hubness is set, the candidate neighbors are further reordered with hubness reduction. Finally, n_neighbors objects are used from the (optionally reordered) candidates.

hubness: {'mutual_proximity', 'local_scaling', 'dis_sim_local', None}, optional Hubness reduction algorithm

- 'mutual_proximity' or 'mp' will use `MutualProximity`
- 'local_scaling' or 'ls' will use `LocalScaling`
- 'dis_sim_local' or 'dsl' will use `DisSimLocal`

If None, no hubness reduction will be performed (=vanilla kNN).

hubness_params: dict, optional Override default parameters of the selected hubness reduction algorithm. For example, with hubness='mp' and hubness_params={'method': 'normal'} a mutual proximity variant is used, which models distance distributions with independent Gaussians.

leaf_size: int, optional (default = 30) Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

metric: string or callable, default 'minkowski' metric to use for distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

Distance matrices are not supported.

Valid values for metric are:

- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

See the documentation for scipy.spatial.distance for details on these metrics.

p: integer, optional (default = 2) Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params: dict, optional (default = None) Additional keyword arguments for the metric function.

n_jobs: int or None, optional (default=None) The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

See also:

KNeighborsClassifier

RadiusNeighborsClassifier

KNeighborsRegressor

RadiusNeighborsRegressor

BallTree

Notes

See Nearest Neighbors in the scikit-learn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> import numpy as np
>>> from skhubness.neighbors import NearestNeighbors
>>> samples = [[0, 0, 2], [1, 0, 0], [0, 0, 1]]
```

```
>>> neigh = NearestNeighbors(2, 0.4)
>>> neigh.fit(samples)
NearestNeighbors(...)
```

```
>>> neigh.kneighbors([[0, 0, 1.3]], 2, return_distance=False)
...
array([[2, 0]]...)
```

```
>>> nbrs = neigh.radius_neighbors([[0, 0, 1.3]], 0.4, return_distance=False)
>>> np.asarray(nbrs[0][0])
array(2)
```

__init__ (*n_neighbors=5*, *radius=1.0*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

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

Methods

__init__([n_neighbors, radius, algorithm, ...])	Initialize self.
fit(X[, y])	Fit the model using X as training data
get_params([deep])	Get parameters for this estimator.
kcandidates([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors([X, n_neighbors, return_distance])	TODO
kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
radius_neighbors([X, radius, return_distance])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph([X, radius, mode])	Computes the (weighted) graph of Neighbors for points in X
set_params(**params)	Set the parameters of this estimator.

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

Fit the model using X as training data

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

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 [mapping of string to any] Parameter names mapped to their values.

kcandidates (*X=None*, *n_neighbors=None*, *return_distance=True*) → numpy.ndarray
Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array] Array representing the lengths to points, only present if return_distance=True

ind [array] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from skhubness.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

kneighbors (*X=None*, *n_neighbors=None*, *return_distance=True*)
TODO

kneighbors_graph (*X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

radius_neighbors(*X=None*, *radius=None*, *return_distance=True*)

Finds the neighbors within a given radius of a point or points.

Return the indices and distances of each point from the dataset lying in a ball with size `radius` around the points of the query array. Points lying on the boundary are included in the results.

The result points are *not* necessarily sorted by distance to their query point.

Parameters

X [array-like, (n_samples, n_features), optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Limiting distance of neighbors to return. (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array, shape (n_samples,) of arrays] Array representing the distances to each point, only present if return_distance=True. The distance values are computed according to the `metric` constructor parameter.

ind [array, shape (n_samples,) of arrays] An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size `radius` around the query points.

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, *radius_neighbors* returns arrays of objects, where each object is a 1D array of indices or distances.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1, 1, 1]:

```
>>> import numpy as np
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from skhubness.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.6)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> rng = neigh.radius_neighbors([[1., 1., 1.]])
>>> print(np.asarray(rng[0][0]))
[1.5 0.5]
>>> print(np.asarray(rng[1][0]))
[1 2]
```

The first array returned contains the distances to all points which are closer than 1.6, while the second array returned contains their indices. In general, multiple points can be queried at the same time.

radius_neighbors_graph (*X=None*, *radius=None*, *mode='connectivity'*)

Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

Parameters

X [array-like, shape = [n_samples, n_features], optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Radius of neighborhoods. (default is the value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse matrix in CSR format, shape = [n_samples, n_samples]] A[i, j] is assigned the weight of edge that connects i to j.

See also:

kneighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 0.],
       [1., 0., 1.]])
```

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.10 skhubness.neighbors.NNG

class skhubness.neighbors.**NNG**(*n_candidates: int = 5*, *metric: str = 'euclidean'*, *index_dir: str = 'auto'*, *optimize: bool = False*, *edge_size_for_creation: int = 80*, *edge_size_for_search: int = 40*, *num_incoming: int = - 1*, *num_outgoing: int = - 1*, *epsilon: float = 0.1*, *n_jobs: int = 1*, *verbose: int = 0*)

Wrapper for ngtpy and NNG variants.

By default, the graph is an ANNG. Only when the *optimize* parameter is set, the graph is optimized to obtain an ONNG.

Parameters

n_candidates: int, default = 5 Number of neighbors to retrieve

metric: str, default = 'euclidean' Distance metric, allowed are 'manhattan', 'L1', 'euclidean', 'L2', 'minkowski', 'Angle', 'Normalized Angle', 'Hamming', 'Jaccard', 'Cosine' or 'Normalized Cosine'.

index_dir: str, default = 'auto' Store the index in the given directory. If None, keep the index in main memory (NON pickleable index), If index_dir is a string, it is interpreted as a directory to store the index into, if 'auto', create a temp dir for the index, preferably in /dev/shm on Linux. Note: The directory/the index will NOT be deleted automatically.

optimize: bool, default = False Use ONNG method by optimizing the ANNG graph. May require long time for index creation.

edge_size_for_creation: int, default = 80 Increasing ANNG edge size improves retrieval accuracy at the cost of more time

edge_size_for_search: int, default = 40 Increasing ANNG edge size improves retrieval accuracy at the cost of more time

epsilon: float, default 0.1 Trade-off in ANNG between higher accuracy (larger epsilon) and shorter query time (smaller epsilon)

num_incoming: int Number of incoming edges in ONNG graph

num_outgoing: int Number of outgoing edges in ONNG graph

n_jobs: int, default = 1 Number of parallel jobs

verbose: int, default = 0 Verbosity level. If verbose > 0, show tqdm progress bar on indexing and querying.

Notes

NNG stores the index to a directory specified in *index_dir*. The index is persistent, and will NOT be deleted automatically. It is the user's responsibility to take care of deletion, when required.

Attributes

valid_metrics: List of valid distance metrics/measures

__init__ (*n_candidates: int = 5*, *metric: str = 'euclidean'*, *index_dir: str = 'auto'*, *optimize: bool = False*, *edge_size_for_creation: int = 80*, *edge_size_for_search: int = 40*, *num_incoming: int = - 1*, *num_outgoing: int = - 1*, *epsilon: float = 0.1*, *n_jobs: int = 1*, *verbose: int = 0*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_candidates, metric, index_dir, ...])	Initialize self.
fit(X[, y])	Build the ngtpy.Index and insert data from X.
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_candidates, return_distance])	Retrieve k nearest neighbors.
set_params(**params)	Set the parameters of this estimator.

Attributes

internal_distance_type
valid_metrics

fit (*X*, *y=None*) → skhubness.neighbors.nng.NNG
Build the ngtpy.Index and insert data from X.

Parameters

X: np.array Data to be indexed

y: any Ignored

Returns

self: NNG An instance of NNG with a built index

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 [mapping of string to any] Parameter names mapped to their values.

kneighbors(*X=None*, *n_candidates=None*, *return_distance=True*) → Union[Tuple[numpy.array, numpy.array], numpy.array]

Retrieve k nearest neighbors.

Parameters

X: np.array or None, optional, default = None Query objects. If None, search among the indexed objects.

n_candidates: int or None, optional, default = None Number of neighbors to retrieve. If None, use the value passed during construction.

return_distance: bool, default = True If return_distance, will return distances and indices to neighbors. Else, only return the indices.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.11 skhubness.neighbors.PuffinnLSH

class skhubness.neighbors.**PuffinnLSH**(*n_candidates: int = 5*, *metric: str = 'euclidean'*, *memory: int = 1073741824*, *recall: float = 0.9*, *n_jobs: int = 1*, *verbose: int = 0*)

Wrap Puffinn LSH for scikit-learn compatibility.

Parameters

n_candidates: int, default = 5 Number of neighbors to retrieve

metric: str, default = 'euclidean' Distance metric, allowed are "angular", "jaccard". Other metrics are partially supported, such as 'euclidean', 'sqeuclidean'. In these cases, 'angular' distances are used to find the candidate set of neighbors with LSH among all indexed objects, and (squared) Euclidean distances are subsequently only computed for the candidates.

memory: int, default = 1GB Max memory usage

recall: float, default = 0.90 Probability of finding the true nearest neighbors among the candidates

n_jobs: int, default = 1 Number of parallel jobs

verbose: int, default = 0 Verbosity level. If verbose > 0, show tqdm progress bar on indexing and querying.

Attributes

valid_metrics: List of valid distance metrics/measures

__init__ (*n_candidates: int = 5*, *metric: str = 'euclidean'*, *memory: int = 1073741824*, *recall: float = 0.9*, *n_jobs: int = 1*, *verbose: int = 0*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_candidates, metric, memory, ...])	Initialize self.
fit(X[, y])	Build the puffinn LSH index and insert data from X.
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_candidates, return_distance])	Retrieve k nearest neighbors.
set_params(**params)	Set the parameters of this estimator.

Attributes

metric_map
valid_metrics

fit (*X*, *y=None*) → skhubness.neighbors.lsh.PuffinnLSH
Build the puffinn LSH index and insert data from X.

Parameters

X: np.array Data to be indexed

y: any Ignored

Returns

self: Puffinn An instance of Puffinn with a built index

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 [mapping of string to any] Parameter names mapped to their values.

kneighbors (*X=None*, *n_candidates=None*, *return_distance=True*) → Union[Tuple[numpy.array, numpy.array], numpy.array]
Retrieve k nearest neighbors.

Parameters

X: np.array or None, optional, default = None Query objects. If None, search among the indexed objects.

n_candidates: int or None, optional, default = None Number of neighbors to retrieve. If None, use the value passed during construction.

return_distance: bool, default = True If return_distance, will return distances and indices to neighbors. Else, only return the indices.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.12 skhubness.neighbors.RadiusNeighborsClassifier

class skhubness.neighbors.**RadiusNeighborsClassifier**(*radius=1.0*, *weights='uniform'*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *outlier_label=None*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

Classifier implementing a vote among neighbors within a given radius

Read more in the scikit-learn User Guide

Parameters

radius: float, optional (default = 1.0) Range of parameter space to use by default for *radius_neighbors()* queries.

weights: str or callable weight function used in prediction. Possible values:

- 'uniform': uniform weights. All points in each neighborhood are weighted equally.
- 'distance': weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable]: a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm: {'auto', 'falconn_lsh', 'ball_tree', 'kd_tree', 'brute'}, optional Algorithm used to compute the nearest neighbors:

- 'falconn_lsh' will use *FalconnLSH*
- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.

- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to *fit()* method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

algorithm_params: dict, optional Override default parameters of the NN algorithm. For example, with algorithm='lsh' and algorithm_params={n_candidates: 100} one hundred approximate neighbors are retrieved with LSH. If parameter hubness is set, the candidate neighbors are further reordered with hubness reduction. Finally, n_neighbors objects are used from the (optionally reordered) candidates.

hubness: {'mutual_proximity', 'local_scaling', 'dis_sim_local', None}, optional Hubness reduction algorithm

- 'mutual_proximity' or 'mp' will use `MutualProximity`
- 'local_scaling' or 'ls' will use `LocalScaling`
- 'dis_sim_local' or 'dsl' will use `DisSimLocal`

If None, no hubness reduction will be performed (=vanilla kNN).

hubness_params: dict, optional Override default parameters of the selected hubness reduction algorithm. For example, with hubness='mp' and hubness_params={'method': 'normal'} a mutual proximity variant is used, which models distance distributions with independent Gaussians.

leaf_size: int, optional (default = 30) Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p: integer, optional (default = 2) Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric: string or callable, default 'minkowski' the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics.

outlier_label: int, optional (default = None) Label, which is given for outlier samples (samples with no neighbors on given radius). If set to None, ValueError is raised, when outlier is detected.

metric_params: dict, optional (default = None) Additional keyword arguments for the metric function.

n_jobs: int or None, optional (default=None) The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

See also:

KNeighborsClassifier

RadiusNeighborsRegressor

KNeighborsRegressor

NearestNeighbors

Notes

See Nearest Neighbors in the scikit-learn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from skhubness.neighbors import RadiusNeighborsClassifier
>>> neigh = RadiusNeighborsClassifier(radius=1.0)
>>> neigh.fit(X, y)
RadiusNeighborsClassifier(...)
>>> print(neigh.predict([[1.5]]))
[0]
```

__init__(*radius=1.0*, *weights='uniform'*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *outlier_label=None*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

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

Methods

__init__([radius, weights, algorithm, ...])	Initialize self.
fit(X, y)	Fit the model using X as training data and y as target values
get_params([deep])	Get parameters for this estimator.
kcandidates([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
predict(X)	Predict the class labels for the provided data
radius_neighbors([X, radius, return_distance])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph([X, radius, mode])	Computes the (weighted) graph of Neighbors for points in X
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

fit(*X*, *y*)

Fit the model using X as training data and y as target values

Parameters

X [{array-like, sparse matrix, BallTree, KDTree, HNSW, FalconnLSH, PuffinLSH, NNG, RandomProjectionTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

y [{array-like, sparse matrix}] Target values of shape = [n_samples] or [n_samples, n_outputs]

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 [mapping of string to any] Parameter names mapped to their values.

kcandidates(*X=None*, *n_neighbors=None*, *return_distance=True*) → numpy.ndarray
Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array] Array representing the lengths to points, only present if return_distance=True

ind [array] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from skhubness.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

predict(*X*)
Predict the class labels for the provided data

Parameters

X: array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed' Test samples.

Returns

y: array of shape [n_samples] or [n_samples, n_outputs] Class labels for each data sample.

radius_neighbors(*X=None*, *radius=None*, *return_distance=True*)
Finds the neighbors within a given radius of a point or points.

Return the indices and distances of each point from the dataset lying in a ball with size `radius` around the points of the query array. Points lying on the boundary are included in the results.

The result points are *not* necessarily sorted by distance to their query point.

Parameters

X [array-like, (n_samples, n_features), optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Limiting distance of neighbors to return. (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array, shape (n_samples,) of arrays] Array representing the distances to each point, only present if return_distance=True. The distance values are computed according to the `metric` constructor parameter.

ind [array, shape (n_samples,) of arrays] An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size `radius` around the query points.

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, *radius_neighbors* returns arrays of objects, where each object is a 1D array of indices or distances.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1, 1, 1]:

```
>>> import numpy as np
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from skhubness.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.6)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> rng = neigh.radius_neighbors([[1., 1., 1.]])
>>> print(np.asarray(rng[0][0]))
[1.5 0.5]
>>> print(np.asarray(rng[1][0]))
[1 2]
```

The first array returned contains the distances to all points which are closer than 1.6, while the second array returned contains their indices. In general, multiple points can be queried at the same time.

radius_neighbors_graph(*X=None*, *radius=None*, *mode='connectivity'*)
Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

Parameters

X [array-like, shape = [n_samples, n_features], optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Radius of neighborhoods. (default is the value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse matrix in CSR format, shape = [n_samples, n_samples]] A[i, j] is assigned the weight of edge that connects i to j.

See also:

kneighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 0.],
       [1., 0., 1.]])
```

score (*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

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

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of self.predict(X) wrt. y.

set_params (***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.13 skhubness.neighbors.RadiusNeighborsRegressor

class skhubness.neighbors.**RadiusNeighborsRegressor**(*radius=1.0*, *weights='uniform'*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

Regression based on neighbors within a fixed radius.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Read more in the scikit-learn User Guide.

Parameters

radius: float, optional (default = 1.0) Range of parameter space to use by default for *radius_neighbors()* queries.

weights: str or callable weight function used in prediction. Possible values:

- 'uniform': uniform weights. All points in each neighborhood are weighted equally.
- 'distance': weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable]: a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm: {'auto', 'falconn_lsh', 'ball_tree', 'kd_tree', 'brute'}, optional Algorithm used to compute the nearest neighbors:

- 'falconn_lsh' will use *FalconnLSH*
- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to *fit()* method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

algorithm_params: dict, optional Override default parameters of the NN algorithm. For example, with algorithm='lsh' and algorithm_params={n_candidates: 100} one hundred approximate neighbors are retrieved with LSH. If parameter hubness is set, the candidate neighbors are further reordered with hubness reduction. Finally, n_neighbors objects are used from the (optionally reordered) candidates.

hubness: {'mutual_proximity', 'local_scaling', 'dis_sim_local', None}, optional Hubness reduction algorithm

- 'mutual_proximity' or 'mp' will use `MutualProximity`
- 'local_scaling' or 'ls' will use `LocalScaling`
- 'dis_sim_local' or 'dsl' will use `DisSimLocal`

If None, no hubness reduction will be performed (=vanilla kNN).

hubness_params: dict, optional Override default parameters of the selected hubness reduction algorithm. For example, with hubness='mp' and hubness_params={'method': 'normal'} a mutual proximity variant is used, which models distance distributions with independent Gaussians.

leaf_size: int, optional (default = 30) Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p: integer, optional (default = 2) Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric: string or callable, default 'minkowski' the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics.

metric_params: dict, optional (default = None) Additional keyword arguments for the metric function.

n_jobs: int or None, optional (default=None) The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See scikit-learn Glossary for more details.

See also:

NearestNeighbors

KNeighborsRegressor

KNeighborsClassifier

RadiusNeighborsClassifier

Notes

See Nearest Neighbors in the scikit-learn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from skhubness.neighbors import RadiusNeighborsRegressor
>>> neigh = RadiusNeighborsRegressor(radius=1.0)
>>> neigh.fit(X, y)
RadiusNeighborsRegressor(...)
>>> print(neigh.predict([[1.5]]))
[0.5]
```

__init__ (*radius=1.0*, *weights='uniform'*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

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

Methods

__init__([radius, weights, algorithm, ...])	Initialize self.
fit(X, y)	Fit the model using X as training data and y as target values
get_params([deep])	Get parameters for this estimator.
kcandidates([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
predict(X)	Predict the target for the provided data
radius_neighbors([X, radius, return_distance])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph([X, radius, mode])	Computes the (weighted) graph of Neighbors for points in X
score(X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

fit (*X*, *y*)

Fit the model using X as training data and y as target values

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

y [{array-like, sparse matrix}]

Target values, array of float values, shape = [n_samples] or [n_samples, n_outputs]

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 [mapping of string to any] Parameter names mapped to their values.

kcandidates(*X=None*, *n_neighbors=None*, *return_distance=True*) → numpy.ndarray
Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array] Array representing the lengths to points, only present if return_distance=True

ind [array] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from skhubness.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

predict(*X*)
Predict the target for the provided data

Parameters

X: array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed' Test samples.

Returns

y: array of float, shape = [n_samples] or [n_samples, n_outputs] Target values

radius_neighbors(*X=None*, *radius=None*, *return_distance=True*)
Finds the neighbors within a given radius of a point or points.

Return the indices and distances of each point from the dataset lying in a ball with size `radius` around the points of the query array. Points lying on the boundary are included in the results.

The result points are *not* necessarily sorted by distance to their query point.

Parameters

X [array-like, (n_samples, n_features), optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Limiting distance of neighbors to return. (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array, shape (n_samples,) of arrays] Array representing the distances to each point, only present if return_distance=True. The distance values are computed according to the `metric` constructor parameter.

ind [array, shape (n_samples,) of arrays] An array of arrays of indices of the approximate nearest points from the population matrix that lie within a ball of size `radius` around the query points.

Notes

Because the number of neighbors of each point is not necessarily equal, the results for multiple query points cannot be fit in a standard data array. For efficiency, *radius_neighbors* returns arrays of objects, where each object is a 1D array of indices or distances.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1, 1, 1]:

```
>>> import numpy as np
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from skhubness.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.6)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> rng = neigh.radius_neighbors([[1., 1., 1.]])
>>> print(np.asarray(rng[0][0]))
[1.5 0.5]
>>> print(np.asarray(rng[1][0]))
[1 2]
```

The first array returned contains the distances to all points which are closer than 1.6, while the second array returned contains their indices. In general, multiple points can be queried at the same time.

radius_neighbors_graph (*X=None*, *radius=None*, *mode='connectivity'*)
Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

Parameters

X [array-like, shape = [n_samples, n_features], optional] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

radius [float] Radius of neighborhoods. (default is the value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse matrix in CSR format, shape = [n_samples, n_samples]] A[i, j] is assigned the weight of edge that connects i to j.

See also:

kneighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(radius=1.5)
>>> neigh.fit(X)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.radius_neighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 0.],
       [1., 0., 1.]])
```

score (*X*, *y*, *sample_weight=None*)

Return the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like of shape (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead, shape = (n_samples, n_samples_fitted), where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.14 skhubness.neighbors.RandomProjectionTree

class skhubness.neighbors.**RandomProjectionTree**(*n_candidates: int = 5*, *metric: str = 'euclidean'*, *n_trees: int = 10*, *search_k: int = - 1*, *mmap_dir: str = 'auto'*, *n_jobs: int = 1*, *verbose: int = 0*)
Wrapper for using annoy.AnnoyIndex

Annoy is an approximate nearest neighbor library, that builds a forest of random projections trees.

Parameters

n_candidates: int, default = 5 Number of neighbors to retrieve

metric: str, default = 'euclidean' Distance metric, allowed are "angular", "euclidean", "manhattan", "hamming", "dot"

n_trees: int, default = 10 Build a forest of n_trees trees. More trees gives higher precision when querying, but are more expensive in terms of build time and index size.

search_k: int, default = -1 Query will inspect search_k nodes. A larger value will give more accurate results, but will take longer time.

mmap_dir: str, default = 'auto' Memory-map the index to the given directory. This is required to make the the class pickleable. If None, keep everything in main memory (NON pickleable index), if mmap_dir is a string, it is interpreted as a directory to store the index into, if 'auto', create a temp dir for the index, preferably in /dev/shm on Linux.

n_jobs: int, default = 1 Number of parallel jobs

verbose: int, default = 0 Verbosity level. If verbose > 0, show tqdm progress bar on indexing and querying.

Attributes

valid_metrics: List of valid distance metrics/measures

__init__(*n_candidates: int = 5*, *metric: str = 'euclidean'*, *n_trees: int = 10*, *search_k: int = - 1*, *mmap_dir: str = 'auto'*, *n_jobs: int = 1*, *verbose: int = 0*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_candidates, metric, n_trees, ...])	Initialize self.
fit(X[, y])	Build the annoy.Index and insert data from X.
get_params([deep])	Get parameters for this estimator.
kneighbors([X, n_candidates, return_distance])	Retrieve k nearest neighbors.
set_params(**params)	Set the parameters of this estimator.

Attributes

valid_metrics

fit (*X*, *y=None*) → skhubness.neighbors.random_projection_trees.RandomProjectionTree
Build the annoy.Index and insert data from X.

Parameters

X: np.array Data to be indexed

y: any Ignored

Returns

self: RandomProjectionTree An instance of RandomProjectionTree with a built index

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 [mapping of string to any] Parameter names mapped to their values.

kneighbors (*X=None*, *n_candidates=None*, *return_distance=True*) → Union[Tuple[numpy.array, numpy.array], numpy.array]
Retrieve k nearest neighbors.

Parameters

X: np.array or None, optional, default = None Query objects. If None, search among the indexed objects.

n_candidates: int or None, optional, default = None Number of neighbors to retrieve. If None, use the value passed during construction.

return_distance: bool, default = True If return_distance, will return distances and indices to neighbors. Else, only return the indices.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.15 skhubness.neighbors.kneighbors_graph

skhubness.neighbors.**kneighbors_graph**(*X*, *n_neighbors*, *mode='connectivity'*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *metric='minkowski'*, *p=2*, *metric_params=None*, *include_self=False*, *n_jobs=None*)

Computes the (weighted) graph of k-Neighbors for points in X

Read more in the scikit-learn User Guide

Parameters

X: array-like or BallTree, shape = [n_samples, n_features] Sample data, in the form of a numpy array or a precomputed *BallTree*.

n_neighbors: int Number of neighbors for each sample.

mode: {'connectivity', 'distance'}, optional Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, and 'distance' will return the distances between neighbors according to the given metric.

algorithm: {'auto', 'hnsw', 'lsh', 'falconn_lsh', 'ball_tree', 'kd_tree', 'brute'}, optional Algorithm used to compute the nearest neighbors:

- 'hnsw' will use *HNSW*
- 'lsh' will use *PuffinnLSH*
- 'falconn_lsh' will use *FalconnLSH*
- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

algorithm_params: dict, optional Override default parameters of the NN algorithm. For example, with algorithm='lsh' and algorithm_params={n_candidates: 100} one hundred approximate neighbors are retrieved with LSH. If parameter hubness is set, the candidate neighbors are further reordered with hubness reduction. Finally, n_neighbors objects are used from the (optionally reordered) candidates.

hubness: {'mutual_proximity', 'local_scaling', 'dis_sim_local', None}, optional Hubness reduction algorithm

- 'mutual_proximity' or 'mp' will use `MutualProximity`
- 'local_scaling' or 'ls' will use `LocalScaling`
- 'dis_sim_local' or 'dsl' will use `DisSimLocal`

If None, no hubness reduction will be performed (=vanilla kNN).

hubness_params: dict, optional Override default parameters of the selected hubness reduction algorithm. For example, with hubness='mp' and hubness_params={'method': 'normal'} a mutual proximity variant is used, which models distance distributions with independent Gaussians.

metric: string, default 'minkowski' The distance metric used to calculate the k-Neighbors for each sample point. The DistanceMetric class gives a list of available metrics. The default distance is 'euclidean' ('minkowski' metric with the p param equal to 2.)

p: int, default 2 Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params: dict, optional additional keyword arguments for the metric function.

include_self: bool, default=False. Whether or not to mark each sample as the first nearest neighbor to itself. If *None*, then True is used for mode='connectivity' and False for mode='distance' as this will preserve backwards compatibility.

n_jobs: int or None, optional (default=None) The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

Returns

A: sparse matrix in CSR format, shape = [n_samples, n_samples] A[i, j] is assigned the weight of edge that connects i to j.

See also:

radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from skhubness.neighbors import kneighbors_graph
>>> A = kneighbors_graph(X, 2, mode='connectivity', include_self=True)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

4.2.16 skhubness.neighbors.radius_neighbors_graph

skhubness.neighbors.**radius_neighbors_graph**(*X*, *radius*, *mode='connectivity'*, *algorithm: str = 'auto'*, *algorithm_params: dict = None*, *hubness: str = None*, *hubness_params: dict = None*, *metric='minkowski'*, *p=2*, *metric_params=None*, *include_self=False*, *n_jobs=None*)

Computes the (weighted) graph of Neighbors for points in X

Neighborhoods are restricted the points at a distance lower than radius.

Read more in the scikit-learn User Guide

Parameters

X: array-like or BallTree, shape = [n_samples, n_features] Sample data, in the form of a numpy array or a precomputed *BallTree*.

radius: float Radius of neighborhoods.

mode: {'connectivity', 'distance'}, optional Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, and 'distance' will return the distances between neighbors according to the given metric.

algorithm: {'auto', 'falconn_lsh', 'ball_tree', 'kd_tree', 'brute'}, optional Algorithm used to compute the nearest neighbors:

- 'falconn_lsh' will use *FalconnLSH*
- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

algorithm_params: dict, optional Override default parameters of the NN algorithm. For example, with algorithm='lsh' and algorithm_params={n_candidates: 100} one hundred approximate neighbors are retrieved with LSH. If parameter hubness is set, the candidate neighbors are further reordered with hubness reduction. Finally, n_neighbors objects are used from the (optionally reordered) candidates.

hubness: {'mutual_proximity', 'local_scaling', 'dis_sim_local', None}, optional Hubness reduction algorithm

- 'mutual_proximity' or 'mp' will use `MutualProximity`
- 'local_scaling' or 'ls' will use `LocalScaling`
- 'dis_sim_local' or 'dsl' will use `DisSimLocal`

If None, no hubness reduction will be performed (=vanilla kNN).

hubness_params: dict, optional Override default parameters of the selected hubness reduction algorithm. For example, with hubness='mp' and hubness_params={'method': 'normal'} a mutual proximity variant is used, which models distance distributions with independent Gaussians.

metric: string, default 'minkowski' The distance metric used to calculate the neighbors within a given radius for each sample point. The DistanceMetric class gives a list of available metrics. The default distance is 'euclidean' ('minkowski' metric with the param equal to 2.)

p: int, default 2 Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params: dict, optional additional keyword arguments for the metric function.

include_self: bool, default=False Whether or not to mark each sample as the first nearest neighbor to itself. If *None*, then True is used for mode='connectivity' and False for mode='distance' as this will preserve backwards compatibility.

n_jobs: int or None, optional (default=None) The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

Returns

A: sparse matrix in CSR format, shape = [n_samples, n_samples] A[i, j] is assigned the weight of edge that connects i to j.

See also:

kneighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from skhubness.neighbors import radius_neighbors_graph
>>> A = radius_neighbors_graph(X, 1.5, mode='connectivity',
...                            include_self=True)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 0.],
       [1., 0., 1.]])
```

4.2.17 skhubness.neighbors.KernelDensity

class skhubness.neighbors.**KernelDensity**(***kwargs*)

Kernel Density Estimation.

Read more in the scikit-learn User Guide.

Parameters

bandwidth [float] The bandwidth of the kernel.

algorithm [str] The tree algorithm to use. Valid options are ['kd_tree'|'ball_tree'|'auto']. Default is 'auto'.

kernel [str] The kernel to use. Valid kernels are ['gaussian'|'tophat'|'epanechnikov'|'exponential'|'linear'|'cosine'] Default is 'gaussian'.

metric [str] The distance metric to use. Note that not all metrics are valid with all algorithms. Refer to the documentation of *BallTree* and *KDTree* for a description of available algorithms. Note that the normalization of the density output is correct only for the Euclidean distance metric. Default is 'euclidean'.

atol [float] The desired absolute tolerance of the result. A larger tolerance will generally lead to faster execution. Default is 0.

rtol [float] The desired relative tolerance of the result. A larger tolerance will generally lead to faster execution. Default is 1E-8.

breadth_first [bool] If true (default), use a breadth-first approach to the problem. Otherwise use a depth-first approach.

leaf_size [int] Specify the leaf size of the underlying tree. See *BallTree* or *KDTree* for details. Default is 40.

metric_params [dict] Additional parameters to be passed to the tree for use with the metric. For more information, see the documentation of *BallTree* or *KDTree*.

See also:

sklearn.neighbors.KDTree K-dimensional tree for fast generalized N-point problems.

sklearn.neighbors.BallTree Ball tree for fast generalized N-point problems.

Examples

Compute a gaussian kernel density estimate with a fixed bandwidth. >>> import numpy as np >>> rng = np.random.RandomState(42) >>> X = rng.random_sample((100, 3)) >>> kde = KernelDensity(kernel='gaussian', bandwidth=0.5).fit(X) >>> log_density = kde.score_samples(X[:3]) >>> log_density array([-1.52955942, -1.51462041, -1.60244657])

__init__ (*, *bandwidth=1.0*, *algorithm='auto'*, *kernel='gaussian'*, *metric='euclidean'*, *atol=0*, *rtol=0*, *breadth_first=True*, *leaf_size=40*, *metric_params=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*[, bandwidth, algorithm, kernel, ...])	Initialize self.
fit(X[, y, sample_weight])	Fit the Kernel Density model on the data.
get_params([deep])	Get parameters for this estimator.
sample([n_samples, random_state])	Generate random samples from the model.
score(X[, y])	Compute the total log probability density under the model.
score_samples(X)	Evaluate the log density model on the data.
set_params(**params)	Set the parameters of this estimator.

fit (*X*, *y=None*, *sample_weight=None*)
Fit the Kernel Density model on the data.

Parameters

X [array_like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

y [None] Ignored. This parameter exists only for compatibility with `sklearn.pipeline.Pipeline`.

sample_weight [array_like, shape (n_samples,), optional] List of sample weights attached to the data X.

New in version 0.20.

Returns

self [object] Returns instance of object.

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 [mapping of string to any] Parameter names mapped to their values.

sample(*n_samples=1*, *random_state=None*)

Generate random samples from the model.

Currently, this is implemented only for gaussian and tophat kernels.

Parameters

n_samples [int, optional] Number of samples to generate. Defaults to 1.

random_state [int, RandomState instance, default=None] Determines random number generation used to generate random samples. Pass an int for reproducible results across multiple function calls. See :term: *Glossary <random_state>*.

Returns

X [array_like, shape (n_samples, n_features)] List of samples.

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

Compute the total log probability density under the model.

Parameters

X [array_like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

y [None] Ignored. This parameter exists only for compatibility with `sklearn.pipeline.Pipeline`.

Returns

logprob [float] Total log-likelihood of the data in X. This is normalized to be a probability density, so the value will be low for high-dimensional data.

score_samples(*X*)

Evaluate the log density model on the data.

Parameters

X [array_like, shape (n_samples, n_features)] An array of points to query. Last dimension should match dimension of training data (n_features).

Returns

density [ndarray, shape (n_samples,)] The array of log(density) evaluations. These are normalized to be probability densities, so values will be low for high-dimensional data.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.18 skhubness.neighbors.LocalOutlierFactor

class skhubness.neighbors.**LocalOutlierFactor**(*n_neighbors=20*, *algorithm: str = 'auto'*, *algorithm_params: Optional[dict] = None*, *hubness: Optional[str] = None*, *hubness_params: Optional[dict] = None*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *contamination='auto'*, *novelty=False*, *n_jobs=None*)

Unsupervised Outlier Detection using Local Outlier Factor (LOF)

The anomaly score of each sample is called Local Outlier Factor. It measures the local deviation of density of a given sample with respect to its neighbors. It is local in that the anomaly score depends on how isolated the object is with respect to the surrounding neighborhood. More precisely, locality is given by k-nearest neighbors, whose distance is used to estimate the local density. By comparing the local density of a sample to the local densities of its neighbors, one can identify samples that have a substantially lower density than their neighbors. These are considered outliers.

Parameters

n_neighbors [int, optional (default=20)] Number of neighbors to use by default for *kneighbors()* queries. If n_neighbors is larger than the number of samples provided, all samples will be used.

algorithm [{'auto', 'hnsw', 'lsh', 'falconn_lsh', 'nng', 'rptree', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors:

- 'hnsw' will use *HNSW*
- 'lsh' will use *PuffinnLSH*
- 'falconn_lsh' will use *FalconnLSH*
- 'nng' will use *NNG*
- 'rptree' will use *RandomProjectionTree*
- 'ball_tree' will use *BallTree*
- 'kd_tree' will use *KDTree*
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate exact algorithm based on the values passed to *fit()* method. This will not select an approximate nearest neighbor algorithm.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

algorithm_params [dict, optional] Override default parameters of the NN algorithm. For example, with algorithm='lsh' and algorithm_params={n_candidates: 100} one hundred approximate neighbors are retrieved with LSH. If parameter hubness is set, the candidate neighbors are further reordered with hubness reduction. Finally, n_neighbors objects are used from the (optionally reordered) candidates.

hubness [{'mutual_proximity', 'local_scaling', 'dis_sim_local', None}, optional] Hubness reduction algorithm

- 'mutual_proximity' or 'mp' will use `MutualProximity`
- 'local_scaling' or 'ls' will use `LocalScaling`
- 'dis_sim_local' or 'dsl' will use `DisSimLocal`

If None, no hubness reduction will be performed (=vanilla kNN).

hubness_params: dict, optional Override default parameters of the selected hubness reduction algorithm. For example, with hubness='mp' and hubness_params={'method': 'normal'} a mutual proximity variant is used, which models distance distributions with independent Gaussians.

leaf_size: int, optional (default=30) Leaf size passed to *BallTree* or *KDTree*. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

metric: string or callable, default 'minkowski' metric used for the distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

If 'precomputed', the training input X is expected to be a distance matrix.

If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

Valid values for metric are:

- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

See the documentation for scipy.spatial.distance for details on these metrics: https://docs.scipy.org/doc/scipy/reference/spatial.distance.html

p: integer, optional (default=2) Parameter for the Minkowski metric from `sklearn.metrics.pairwise.pairwise_distances()`. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params: dict, optional (default=None) Additional keyword arguments for the metric function.

contamination: 'auto' or float, optional (default='auto') The amount of contamination of the data set, i.e. the proportion of outliers in the data set. When fitting this is used to define the threshold on the scores of the samples.

- if 'auto', the threshold is determined as in the original paper,
- if a float, the contamination should be in the range [0, 0.5].

Changed in version 0.22: The default value of `contamination` changed from 0.1 to `'auto'`.

novelty: boolean, default False By default, LocalOutlierFactor is only meant to be used for outlier detection (novelty=False). Set novelty to True if you want to use LocalOutlierFactor for novelty detection. In this case be aware that that you should only use predict, decision_function and score_samples on new unseen data and not on the training set.

n_jobs: int or None, optional (default=None) The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details. Affects only *kneighbors()* and *kneighbors_graph()* methods.

References

[1]

Attributes

negative_outlier_factor_: numpy array, shape (n_samples,) The opposite LOF of the training samples. The higher, the more normal. Inliers tend to have a LOF score close to 1 (`negative_outlier_factor_` close to -1), while outliers tend to have a larger LOF score.

The local outlier factor (LOF) of a sample captures its supposed 'degree of abnormality'. It is the average of the ratio of the local reachability density of a sample and those of its k-nearest neighbors.

n_neighbors_: integer The actual number of neighbors used for *kneighbors()* queries.

offset_: float Offset used to obtain binary labels from the raw scores. Observations having a negative_outlier_factor smaller than *offset_* are detected as abnormal. The offset is set to -1.5 (inliers score around -1), except when a contamination parameter different than "auto" is provided. In that case, the offset is defined in such a way we obtain the expected number of outliers in training.

__init__(*n_neighbors=20*, *algorithm: str = 'auto'*, *algorithm_params: Optional[dict] = None*, *hubness: Optional[str] = None*, *hubness_params: Optional[dict] = None*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *contamination='auto'*, *novelty=False*, *n_jobs=None*)

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

Methods

__init__([n_neighbors, algorithm, ...])	Initialize self.
fit(X[, y])	Fit the model using X as training data.
get_params([deep])	Get parameters for this estimator.
kcandidates([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
kneighbors([X, n_neighbors, return_distance])	TODO
kneighbors_graph([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
set_params(**params)	Set the parameters of this estimator.

Attributes

decision_function	Shifted opposite of the Local Outlier Factor of X.
fit_predict	"Fits the model to the training set X and returns the labels.
predict	Predict the labels (1 inlier, -1 outlier) of X according to LOF.
score_samples	Opposite of the Local Outlier Factor of X.

property decision_function

Shifted opposite of the Local Outlier Factor of X.

Bigger is better, i.e. large values correspond to inliers.

The shift offset allows a zero threshold for being an outlier. Only available for novelty detection (when novelty is set to True). The argument X is supposed to contain *new data*: if X contains a point from training, it considers the later in its own neighborhood. Also, the samples in X are not considered in the neighborhood of any point.

Parameters

X: array-like, shape (n_samples, n_features) The query sample or samples to compute the Local Outlier Factor w.r.t. the training samples.

Returns

shifted_opposite_lof_scores: array, shape (n_samples,) The shifted opposite of the Local Outlier Factor of each input samples. The lower, the more abnormal. Negative scores represent outliers, positive scores represent inliers.

fit (*X*, *y=None*) → skhubness.neighbors.lof.LocalOutlierFactor

Fit the model using X as training data.

Parameters

X: {array-like, sparse matrix, BallTree, KDTree} Training data. If array or matrix, shape [n_samples, n_features], or [n_samples, n_samples] if metric='precomputed'.

y: Ignored not used, present for API consistency by convention.

Returns

self: object

property fit_predict

"Fits the model to the training set X and returns the labels.

Label is 1 for an inlier and -1 for an outlier according to the LOF score and the contamination parameter.

Parameters

X: array-like, shape (n_samples, n_features), default=None The query sample or samples to compute the Local Outlier Factor w.r.t. to the training samples.

y: Ignored not used, present for API consistency by convention.

Returns

is_inlier: array, shape (n_samples,) Returns -1 for anomalies/outliers and 1 for inliers.

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 [mapping of string to any] Parameter names mapped to their values.

kcandidates (*X=None*, *n_neighbors=None*, *return_distance=True*) → numpy.ndarray

Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array] Array representing the lengths to points, only present if return_distance=True

ind [array] Indices of the nearest points in the population matrix.

Examples

In the following example, we construct a NeighborsClassifier class from an array representing our data set and ask who's the closest point to [1,1,1]

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from skhubness.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(algorithm='auto', leaf_size=30, ...)
>>> print(neigh.kneighbors([[1., 1., 1.]]))
(array([[0.5]]), array([[2]]))
```

As you can see, it returns [[0.5]], and [[2]], which means that the element is at distance 0.5 and is the third element of samples (indexes start at 0). You can also query for multiple points:

```
>>> X = [[0., 1., 0.], [1., 0., 1.]]
>>> neigh.kneighbors(X, return_distance=False)
array([[1],
       [2]]...)
```

kneighbors (*X=None*, *n_neighbors=None*, *return_distance=True*)
TODO

kneighbors_graph (*X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

Parameters

X [array-like, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors for each sample. (default is value passed to the constructor).

mode [{'connectivity', 'distance'}, optional] Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are Euclidean distance between points.

Returns

A [sparse graph in CSR format, shape = [n_queries, n_samples_fit]] n_samples_fit is the number of samples in the fitted data A[i, j] is assigned the weight of edge that connects i to j.

See also:

NearestNeighbors.radius_neighbors_graph

Examples

```
>>> X = [[0], [3], [1]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=2)
>>> neigh.fit(X)
NearestNeighbors(n_neighbors=2)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

property predict

Predict the labels (1 inlier, -1 outlier) of X according to LOF.

This method allows to generalize prediction to *new observations* (not in the training set). Only available for novelty detection (when novelty is set to True).

Parameters

X: array-like, shape (n_samples, n_features) The query sample or samples to compute the Local Outlier Factor w.r.t. to the training samples.

Returns

is_inlier: array, shape (n_samples,) Returns -1 for anomalies/outliers and +1 for inliers.

property score_samples

Opposite of the Local Outlier Factor of X.

It is the opposite as bigger is better, i.e. large values correspond to inliers.

Only available for novelty detection (when novelty is set to True). The argument X is supposed to contain *new data*: if X contains a point from training, it considers the later in its own neighborhood. Also, the samples in X are not considered in the neighborhood of any point. The score_samples on training data is available by considering the the `negative_outlier_factor_` attribute.

Parameters

X: array-like, shape (n_samples, n_features) The query sample or samples to compute the Local Outlier Factor w.r.t. the training samples.

Returns

opposite_lof_scores: array, shape (n_samples,) The opposite of the Local Outlier Factor of each input samples. The lower, the more abnormal.

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). 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 [object] Estimator instance.

4.2.19 skhubness.neighbors.NeighborhoodComponentsAnalysis

class skhubness.neighbors.**NeighborhoodComponentsAnalysis**(***kwargs*)
Neighborhood Components Analysis

Neighborhood Component Analysis (NCA) is a machine learning algorithm for metric learning. It learns a linear transformation in a supervised fashion to improve the classification accuracy of a stochastic nearest neighbors rule in the transformed space.

Read more in the scikit-learn User Guide.

Parameters

n_components [int, default=None] Preferred dimensionality of the projected space. If None it will be set to `n_features`.

init [{'auto', 'pca', 'lda', 'identity', 'random'} or ndarray of shape (n_features_a, n_features_b), default='auto'] Initialization of the linear transformation. Possible options are 'auto', 'pca', 'lda', 'identity', 'random', and a numpy array of shape (n_features_a, n_features_b).

'auto' Depending on `n_components`, the most reasonable initialization will be chosen. If `n_components <= n_classes` we use 'lda', as it uses labels information. If not, but `n_components < min(n_features, n_samples)`, we use 'pca', as it projects data in meaningful directions (those of higher variance). Otherwise, we just use 'identity'.

'pca' `n_components` principal components of the inputs passed to *`fit()`* will be used to initialize the transformation. (See `PCA`)

'lda' `min(n_components, n_classes)` most discriminative components of the inputs passed to *`fit()`* will be used to initialize the transformation. (If `n_components > n_classes`, the rest of the components will be zero.) (See `LinearDiscriminantAnalysis`)

'identity' If `n_components` is strictly smaller than the dimensionality of the inputs passed to *`fit()`*, the identity matrix will be truncated to the first `n_components` rows.

'random' The initial transformation will be a random array of shape *(n_components, n_features)*. Each value is sampled from the standard normal distribution.

numpy array n_features_b must match the dimensionality of the inputs passed to *`fit()`* and n_features_a must be less than or equal to that. If `n_components` is not None, n_features_a must match it.

warm_start [bool, default=False] If True and *`fit()`* has been called before, the solution of the previous call to *`fit()`* is used as the initial linear transformation (`n_components` and `init` will be ignored).

max_iter [int, default=50] Maximum number of iterations in the optimization.

tol [float, default=1e-5] Convergence tolerance for the optimization.

callback [callable, default=None] If not None, this function is called after every iteration of the optimizer, taking as arguments the current solution (flattened transformation matrix) and the number of iterations. This might be useful in case one wants to examine or store the transformation found after each iteration.

verbose [int, default=0] If 0, no progress messages will be printed. If 1, progress messages will be printed to stdout. If > 1, progress messages will be printed and the `disp` parameter of `scipy.optimize.minimize()` will be set to `verbose - 2`.

random_state [int or numpy.RandomState, default=None] A pseudo random number generator object or a seed for it if int. If `init='random'`, `random_state` is used to initialize the random transformation. If `init='pca'`, `random_state` is passed as an argument to PCA when initializing the transformation. Pass an int for reproducible results across multiple function calls. See :term: *Glossary* <*random_state*>.

References

[1], [2]

Examples

```
>>> from sklearn.neighbors import NeighborhoodComponentsAnalysis
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
... stratify=y, test_size=0.7, random_state=42)
>>> nca = NeighborhoodComponentsAnalysis(random_state=42)
>>> nca.fit(X_train, y_train)
NeighborhoodComponentsAnalysis(...)
>>> knn = KNeighborsClassifier(n_neighbors=3)
>>> knn.fit(X_train, y_train)
KNeighborsClassifier(...)
>>> print(knn.score(X_test, y_test))
0.933333...
>>> knn.fit(nca.transform(X_train), y_train)
KNeighborsClassifier(...)
>>> print(knn.score(nca.transform(X_test), y_test))
0.961904...
```

Attributes

components_ [ndarray of shape (n_components, n_features)] The linear transformation learned during fitting.

n_iter_ [int] Counts the number of iterations performed by the optimizer.

random_state_ [numpy.RandomState] Pseudo random number generator object used during initialization.

__init__(*n_components=None*, ***, *init='auto'*, *warm_start=False*, *max_iter=50*, *tol=1e-05*, *callback=None*, *verbose=0*, *random_state=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([n_components, init, warm_start, ...])	Initialize self.
fit(X, y)	Fit the model according to the given training data.
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.
transform(X)	Applies the learned transformation to the given data.

fit(*X*, *y*)
Fit the model according to the given training data.

Parameters

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

y [array-like of shape (n_samples,)] The corresponding training labels.

Returns

self [object] returns a trained NeighborhoodComponentsAnalysis model.

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, sparse matrix, dataframe} of shape (n_samples, n_features)]

y [ndarray of shape (n_samples,), default=None] Target values.

****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 [mapping of string to any] 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 pipelines). 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 [object] Estimator instance.

transform(*X*)
Applies the learned transformation to the given data.

Parameters

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

Returns

X_embedded: ndarray of shape (n_samples, n_components) The data samples transformed.

Raises

NotFittedError If *fit()* has not been called before.

4.3 Reduction: skhubness.reduction

The *skhubness.reduction* package provides methods for hubness reduction.

reduction.MutualProximity	Hubness reduction with Mutual Proximity [1].
reduction.LocalScaling	Hubness reduction with Local Scaling [1].
reduction.DisSimLocal	Hubness reduction with DisSimLocal [1].
reduction.hubness_algorithms	Supported hubness reduction algorithms

4.3.1 skhubness.reduction.MutualProximity

class skhubness.reduction.**MutualProximity**(*method: str = 'normal', verbose: int = 0, **kwargs*)
Hubness reduction with Mutual Proximity [1].

Parameters

method: 'normal' or 'empiric', default = 'normal' Model distance distribution with 'method'.

- 'normal' or 'gaussi' model distance distributions with independent Gaussians (fast)
- 'empiric' or 'exact' model distances with the empiric distributions (slow)

verbose: int, default = 0 If verbose > 0, show progress bar.

References

[1]

__init__(*method: str = 'normal', verbose: int = 0, **kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([method, verbose])	Initialize self.
fit(neigh_dist, neigh_ind[, X, assume_sorted])	Fit the model using neigh_dist and neigh_ind as training data.
fit_transform(neigh_dist, neigh_ind, X[, ...])	Equivalent to call .fit().transform()
transform(neigh_dist, neigh_ind[, X, ...])	Transform distance between test and training data with Mutual Proximity.

fit(*neigh_dist*, *neigh_ind*, *X=None*, *assume_sorted=None*, **args*, ***kwargs*) → skhubness.reduction.mutual_proximity.MutualProximity
Fit the model using neigh_dist and neigh_ind as training data.

Parameters

neigh_dist: np.ndarray, shape (n_samples, n_neighbors) Distance matrix of training objects (rows) against their individual k nearest neighbors (columns).

neigh_ind: np.ndarray, shape (n_samples, n_neighbors) Neighbor indices corresponding to the values in neigh_dist.

X: ignored

assume_sorted: ignored

fit_transform(*neigh_dist*, *neigh_ind*, *X*, *assume_sorted=True*, *return_distance=True*, **args*, ***kwargs*)
Equivalent to call .fit().transform()

transform(*neigh_dist*, *neigh_ind*, *X=None*, *assume_sorted=None*, **args*, ***kwargs*)
Transform distance between test and training data with Mutual Proximity.

Parameters

neigh_dist: np.ndarray Distance matrix of test objects (rows) against their individual k nearest neighbors among the training data (columns).

neigh_ind: np.ndarray Neighbor indices corresponding to the values in neigh_dist

X: ignored

assume_sorted: ignored

Returns

hub_reduced_dist, neigh_ind Mutual Proximity distances, and corresponding neighbor indices

Notes

The returned distances are NOT sorted! If you use this class directly, you will need to sort the returned matrices according to hub_reduced_dist. Classes from *skhubness.neighbors* do this automatically.

4.3.2 skhubness.reduction.LocalScaling

class skhubness.reduction.**LocalScaling**(*k: int = 5*, *method: str = 'standard'*, *verbose: int = 0*, ***kwargs*)

Hubness reduction with Local Scaling [1].

Parameters

k: int, default = 5 Number of neighbors to consider for the rescaling

method: 'standard' or 'nicdm', default = 'standard' Perform local scaling with the specified variant:

- 'standard' or 'ls' rescale distances using the distance to the k-th neighbor
- 'nicdm' rescales distances using a statistic over distances to k neighbors

verbose: int, default = 0 If verbose > 0, show progress bar.

References

[1]

__init__(*k: int = 5*, *method: str = 'standard'*, *verbose: int = 0*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([k, method, verbose])	Initialize self.
fit(neigh_dist, neigh_ind[, X, assume_sorted])	Fit the model using neigh_dist and neigh_ind as training data.
fit_transform(neigh_dist, neigh_ind, X[, ...])	Equivalent to call .fit().transform()
transform(neigh_dist, neigh_ind[, X, ...])	Transform distance between test and training data with Mutual Proximity.

fit(*neigh_dist*, *neigh_ind*, *X=None*, *assume_sorted: bool = True*, **args*, ***kwargs*) → skhubness.reduction.local_scaling.LocalScaling
Fit the model using neigh_dist and neigh_ind as training data.

Parameters

neigh_dist: np.ndarray, shape (n_samples, n_neighbors) Distance matrix of training objects (rows) against their individual k nearest neighbors (colums).

neigh_ind: np.ndarray, shape (n_samples, n_neighbors) Neighbor indices corresponding to the values in neigh_dist.

X: ignored

assume_sorted: bool, default = True Assume input matrices are sorted according to neigh_dist. If False, these are sorted here.

fit_transform(*neigh_dist*, *neigh_ind*, *X*, *assume_sorted=True*, *return_distance=True*, **args*, ***kwargs*)
Equivalent to call .fit().transform()

transform(*neigh_dist*, *neigh_ind*, *X=None*, *assume_sorted: bool = True*, **args*, ***kwargs*)
Transform distance between test and training data with Mutual Proximity.

Parameters

neigh_dist: np.ndarray, shape (n_query, n_neighbors) Distance matrix of test objects (rows) against their individual k nearest neighbors among the training data (columns).

neigh_ind: np.ndarray, shape (n_query, n_neighbors) Neighbor indices corresponding to the values in neigh_dist

X: ignored

assume_sorted: bool, default = True Assume input matrices are sorted according to neigh_dist. If False, these are partitioned here.

NOTE: The returned matrices are never sorted.

Returns

hub_reduced_dist, neigh_ind Local scaling distances, and corresponding neighbor indices

Notes

The returned distances are NOT sorted! If you use this class directly, you will need to sort the returned matrices according to hub_reduced_dist. Classes from *skhubness.neighbors* do this automatically.

4.3.3 skhubness.reduction.DisSimLocal

class skhubness.reduction.**DisSimLocal**(*k: int = 5*, *squared: bool = True*, **args*, ***kwargs*)

Hubness reduction with DisSimLocal [1].

Parameters

k: int, default = 5 Number of neighbors to consider for the local centroids

squared: bool, default = True DisSimLocal operates on squared Euclidean distances. If True, return (quasi) squared Euclidean distances; if False, return (quasi) Eucldean distances.

References

[1]

__init__(*k: int = 5*, *squared: bool = True*, **args*, ***kwargs*)

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

Methods

__init__([k, squared])	Initialize self.
fit(neigh_dist, neigh_ind, X[, assume_sorted])	Fit the model using X, neigh_dist, and neigh_ind as training data.
fit_transform(neigh_dist, neigh_ind, X[, ...])	Equivalent to call .fit().transform()
transform(neigh_dist, neigh_ind, X[, ...])	Transform distance between test and training data with DisSimLocal.

fit(*neigh_dist: numpy.ndarray*, *neigh_ind: numpy.ndarray*, *X: numpy.ndarray*, *assume_sorted: bool = True*, **args*, ***kwargs*) → skhubness.reduction.dis_sim.DisSimLocal

Fit the model using X, neigh_dist, and neigh_ind as training data.

Parameters

neigh_dist: np.ndarray, shape (n_samples, n_neighbors) Distance matrix of training objects (rows) against their individual k nearest neighbors (colums).

neigh_ind: np.ndarray, shape (n_samples, n_neighbors) Neighbor indices corresponding to the values in neigh_dist.

X: np.ndarray, shape (n_samples, n_features) Training data, where n_samples is the number of vectors, and n_features their dimensionality (number of features).

assume_sorted: bool, default = True Assume input matrices are sorted according to neigh_dist. If False, these are sorted here.

fit_transform(*neigh_dist*, *neigh_ind*, *X*, *assume_sorted=True*, *return_distance=True*, **args*, ***kwargs*)

Equivalent to call .fit().transform()

transform(*neigh_dist: np.ndarray*, *neigh_ind: np.ndarray*, *X: np.ndarray*, *assume_sorted: bool = True*, **args*, ***kwargs*)

Transform distance between test and training data with DisSimLocal.

Parameters

neigh_dist: np.ndarray, shape (n_query, n_neighbors) Distance matrix of test objects (rows) against their individual k nearest neighbors among the training data (columns).

neigh_ind: np.ndarray, shape (n_query, n_neighbors) Neighbor indices corresponding to the values in neigh_dist

X: np.ndarray, shape (n_query, n_features) Test data, where n_query is the number of vectors, and n_features their dimensionality (number of features).

assume_sorted: ignored

Returns

hub_reduced_dist, neigh_ind DisSimLocal distances, and corresponding neighbor indices

Notes

The returned distances are NOT sorted! If you use this class directly, you will need to sort the returned matrices according to hub_reduced_dist. Classes from *skhubness.neighbors* do this automatically.

4.3.4 skhubness.reduction.hubness_algorithms

skhubness.reduction.**hubness_algorithms = ['mp', 'ls', 'dsl']**

Supported hubness reduction algorithms

CHAPTER

FIVE

HISTORY OF SCIKIT-HUBNESS

scikit-hubness builds upon previous software: the Hub-Toolbox. The original Hub-Toolbox was written for Matlab, and released in parallel with the release of the first hubness reduction methods in JMLR. In essence, it comprises methods to reduce hubness in distance matrices.

The Hub-Toolbox for Python3 is a port from Matlab to Python, which over the years got several extensions and additional functionality, such as more hubness reduction methods (Localized Centering, DisSimLocal, mp-dissim, etc.), approximate hubness reduction, and more. The software was developed by hubness researchers for hubness research.

The new scikit-hubness package is rewritten from scratch with a different goal in mind: Providing easy-to-use neighborhood-based data mining methods (classification, regression, etc.) with transparent hubness reduction. Building upon scikit-learn's neighbors package, we provide a drop-in replacement called skhubness.neighbors, which offers all the functionality of sklearn.neighbors, but adds additional functionality (approximate nearest neighbor search, hubness reduction).

This way, we think that machine learning researchers and practitioners (many of which will be fluent in scikit-learn) can quickly and effectively employ scikit-hubness in their existing workflows, and improve learning in their high-dimensional data.

CHAPTER

SIX

CONTRIBUTING

scikit-hubness is free open source software. Contributions from the community are highly appreciated. Even small contributions improve the software's quality.

Even if you are not familiar with programming languages and tools, you may contribute by filing bugs or any problems as a GitHub issue.

6.1 Git and branching model

We use *git* for version control (CVS), as do most projects nowadays. If you are not familiar with git, there are lots of tutorials on GitHub Guide. All the important basics are covered in the GitHub Git handbook.

Development of *scikit-hubness* (mostly) follows the git flow branching model. There are two main branches: master and develop. For any changes, a new branch should be created. If you want to add a new feature, fix a noncritical bug, etc. one should branch off *develop*. Only if you want to fix a critical bug, branch off *master*.

6.2 Workflow

In case of large changes to the software, please first get in contact with the authors for coordination, for example by filing an issue. If you want to fix small issues (typos in the docs, obvious errors, etc.) you can - of course - directly submit a pull request (PR).

1. **Create a fork of *scikit-hubness* in your GitHub account.** Simply click "Fork" button on https://github.com/VarIr/scikit-hubness.
2. **Clone your fork on your computer.** $ `git clone git@github.com:YOUR-ACCOUNT-GOES-HERE/scikit-hubness.git && cd scikit-hubness`
3. **Add remote upstream.** $ `git remote add upstream git@github.com:VarIr/scikit-hubness.git`
4. **Create feature/bugfix branch.** In case of feature or noncritical bugfix: $ `git checkout develop && git checkout -b featureXYZ develop`

   In case of critical bug: $ `git checkout -b bugfix123 master`
5. **Implement feature/fix bug/fix typo/...** Happy coding!
6. **Create a commit with meaningful message** If you only modified existing files, simply `$ git commit -am "descriptive message what this commit does (in present tense) here"`
7. **Push to GitHub** e.g. $ `git push origin featureXYZ`

8. **Create pull request (PR)** Git will likely provide a link to directly create the PR. If not, click "New pull request" on your fork on GitHub.
9. **Wait...** Several devops checks will be performed automatically (e.g. continuous integration (CI) with Travis, AppVeyor).

   The authors will get in contact with you, and may ask for changes.
10. **Respond to code review.** If there were issues with continous integration, or the authors asked for changes, please create a new commit locally, and simply push again to GitHub as you did before. The PR will be updated automatically.
11. **Maintainers merge PR, when all issues are resolved.** Thanks a lot for your contribution!

6.3 Code style and further guidelines

- Please make sure all code complies with PEP 8
- All code should be documented sufficiently (functions, classes, etc. must have docstrings with general description, parameters, ideally return values, raised exceptions, notes, etc.)
- Documentation style is NumPy format.
- New code must be covered by unit tests using pytest.
- If you fix a bug, please provide regression tests (fail on old code, succeed on new code).
- It may be helpful to install *scikit-hubness* in editable mode for development. When you have already cloned the package, switch into the corresponding directory, and

```
pip install -e .
```

  (don't omit the trailing period). This way, any changes to the code are reflected immediately. That is, you don't need to install the package each and every time, when you make changes while developing code.

6.4 Testing

In *scikit-hubness*, we aim for high code coverage. As of September 2019, between 98% and 99% of all code lines are visited at least once when running the complete test suite. This is primarily to ensure:

- correctness of the code (to some extent) and
- maintainability (new changes don't break old code).

Creating a new PR, ideally all code would be covered by tests. Sometimes, this is not feasible or only with large effort. Pull requests will likely be accepted, if the overall code coverage at least does not decrease.

Unit tests are automatically performed for each PR using CI tools online. This may take some time, however. To run the tests locally, you need *pytest* installed. From the scikit-hubness directory, call

```
pytest skhubness/
```

to run all the tests. You can also restrict the tests to the subpackage you are working on, down to single tests. For example

```
pytest skhubness/reduction/tests/test_local_scaling.py --showlocals -v
```

only runs tests for hubness reduction with local scaling.

In order to check code coverage locally, you need the pytest-cov plugin.

```
pytest skhubness/reduction/ --cov=skhubness/reduction/
```

CHAPTER

SEVEN

CHANGELOG

7.1 Next release

...

7.2 0.21.1 - 2019-12-10

This is a bugfix release due to the recent update of scikit-learn to v0.22.

7.2.1 Fixes

- Require scikit-learn v0.21.3.

  Until the necessary adaptions for v0.22 are completed, scikit-hubness will require scikit-learn v0.21.3.

7.3 0.21.0 - 2019-11-25

This is the first major release of scikit-hubness.

7.3.1 Added

- Enable ONNG provided by NGT (optimized ANNG). Pass `optimize=True` to `NNG`.
- User Guide: Description of all subpackages and common usage scenarios.
- Examples: Various usage examples
- Several tests
- Classes inheriting from `SupervisedIntegerMixin` can be fit with an `ApproximateNearestNeighbor` or `NearestNeighbors` instance, thus reuse precomputed indices.

7.3.2 Changes

- Use argument `algorithm='nng'` for ANNG/ONNG provided by NGT instead of `'onng'`. Also set `optimize=True` in order to use ONNG.

7.3.3 Fixes

- DisSimLocal would previously fail when invoked as `hubness='dis_sim_local'`.
- Hubness reduction would previously ignore `verbose` arguments under certain circumstances.
- `HNSW` would previously ignore `n_jobs` on index creation.
- Fix installation instructions for puffinn.

7.4 0.21.0a9 - 2019-10-30

7.4.1 Added

- General structure for docs
- Enable NGT OpenMP support on MacOS (in addition to Linux)
- Enable Puffinn LSH also on MacOS

7.4.2 Fixes

- Correct mutual proximity (empiric) calculation
- Better handling of optional packages (ANN libraries)

7.4.3 Maintenance

- streamlined CI builds
- several minor code improvements

7.4.4 New contributors

- Silvan David Peter

7.5 0.21.0a8 - 2019-09-12

7.5.1 Added

- Approximate nearest neighbor search
  - LSH by an additional provider, `puffinn` (Linux only, atm)
  - ANNG provided by `ngtpy` (Linux, MacOS)
  - Random projection forests provided by `annoy` (Linux, MacOS, Windows)

7.5.2 Fixes

- Several minor issues
- Several documentations issues

7.6 0.21.0a7 - 2019-07-17

The first alpha release of `scikit-hubness` to appear in this changelog. It already contains the following features:

- Hubness estimation (exact or approximate)
- Hubness reduction (exact or approximate)
  - Mutual proximity
  - Local scaling
  - DisSim Local
- Approximate nearest neighbor search
  - HNSW provided by nmslib
  - LSH provided by falconn

CHAPTER

EIGHT

GETTING STARTED

Get started with `scikit-hubness` in a breeze. Find how to install the package and see all core functionality applied in a single quick start example.

CHAPTER

NINE

USER GUIDE

The User Guide introduces the main concepts of `scikit-hubness`. It explains, how to analyze your data sets for hubness, and how to use the package to lift this *curse of dimensionality*. You will also find examples how to use `skhubness.neighbors` for approximate nearest neighbor search (with or without hubness reduction).

CHAPTER

TEN

API DOCUMENTATION

The API Documentation provides detailed information of the implemented methods. This information includes method descriptions, parameters, references, examples, etc. Find all the information about specific modules and functions of `scikit-hubness` in this section.

CHAPTER

ELEVEN

HISTORY

A brief history of the package, and how it relates to the `Hub-Toolbox`'es.

CHAPTER

TWELVE

DEVELOPMENT

There are several possibilities to contribute to this free open source software. We highly appreciate all input from the community, be it bug reports or code contributions.

Source code, issue tracking, discussion, and continuous integration appear on our GitHub page.

CHAPTER

THIRTEEN

WHAT'S NEW

To see what's new in the latest version of `scikit-hubness`, have a look at the changelog.

BIBLIOGRAPHY

[1] *Radovanović, M.; Nanopoulos, A. & Ivanovic, M. Hubs in space: Popular nearest neighbors in high-dimensional data. Journal of Machine Learning Research, 2010, 11, 2487-2531*

[2] *Feldbauer, R.; Leodolter, M.; Plant, C. & Flexer, A. Fast approximate hubness reduction for large high-dimensional data. IEEE International Conference of Big Knowledge (2018).*

[1] Breunig, M. M., Kriegel, H. P., Ng, R. T., & Sander, J. (2000, May). LOF: identifying density-based local outliers. In ACM sigmod record.

[1] J. Goldberger, G. Hinton, S. Roweis, R. Salakhutdinov. "Neighbourhood Components Analysis". Advances in Neural Information Processing Systems. 17, 513-520, 2005. http://www.cs.nyu.edu/~roweis/papers/ncanips.pdf

[2] Wikipedia entry on Neighborhood Components Analysis https://en.wikipedia.org/wiki/Neighbourhood_components_analysis

[1] Schnitzer, D., Flexer, A., Schedl, M., & Widmer, G. (2012). Local and global scaling reduce hubs in space. The Journal of Machine Learning Research, 13(1), 2871–2902.

[1] Schnitzer, D., Flexer, A., Schedl, M., & Widmer, G. (2012). Local and global scaling reduce hubs in space. The Journal of Machine Learning Research, 13(1), 2871–2902.

[1] Hara K, Suzuki I, Kobayashi K, Fukumizu K, Radovanović M (2016) Flattening the density gradient for eliminating spatial centrality to reduce hubness. In: Proceedings of the 30th AAAI conference on artificial intelligence, pp 1659–1665. https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/viewPaper/12055

PYTHON MODULE INDEX

S

INDEX

Symbols

B

D

F

G

H

K

L

M

N

P

Q

R

S

T

V