`sklearn.neighbors`.KNeighborsClassifier

class `sklearn.neighbors.`**`KNeighborsClassifier`**(*n_neighbors=5*, ***, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*) [source]

Classifier implementing the k-nearest neighbors vote.

Read more in the User Guide.

Toggle Menu

Parameters:

n_neighbors : *int, default=5*

Number of neighbors to use by default for `kneighbors` queries.

weights : *{'uniform', 'distance'}, callable or None, 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.

Refer to the example entitled Nearest Neighbors Classification showing the impact of the `weights` parameter on the decision boundary.

algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*

Algorithm used to compute the nearest neighbors:

- '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.

leaf_size : *int, 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 : *float, 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. This parameter is expected to be positive.

metric : *str or callable, default='minkowski'*

Metric to use for distance computation. Default is "minkowski", which results in the standard Euclidean distance when p = 2. See the documentation of scipy.spatial.distance and the metrics listed in `distance_metrics` for valid metric values.

If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a sparse graph, in which case only "nonzero" elements may be considered neighbors.

If metric is a callable function, it takes two arrays representing 1D vectors as inputs and must return one value indicating the distance between those vectors. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

metric_params : *dict, default=None*

Additional keyword arguments for the metric function.

n_jobs : *int, 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.

Attributes:

classes_ : *array of shape (n_classes,)*

Class labels known to the classifier

effective_metric_ : *str or callble*

The distance metric used. It will be same as the `metric` parameter or a synonym of it, e.g. 'euclidean' if the `metric` parameter set to 'minkowski' and `p` parameter set to 2.

effective_metric_params_ : *dict*

Additional keyword arguments for the metric function. For most metrics will be same with `metric_params` parameter, but may also contain the `p` parameter value if the `effective_metric_` attribute is set to 'minkowski'.

n_features_in_ : *int*

Number of features seen during fit.

New in version 0.24.

Toggle Menu **mes_in_ : *ndarray of shape (`n_features_in_`,)*** Names of features seen during fit. Defined only when `X` has feature names that are all strings.

New in version 1.0.

n_samples_fit_ : *int*
Number of samples in the fitted data.

outputs_2d_ : *bool*
False when `y`'s shape is (n_samples,) or (n_samples, 1) during fit otherwise True.

See also:

`RadiusNeighborsClassifier`
Classifier based on neighbors within a fixed radius.

`KNeighborsRegressor`
Regression based on k-nearest neighbors.

`RadiusNeighborsRegressor`
Regression based on neighbors within a fixed radius.

`NearestNeighbors`
Unsupervised learner for implementing neighbor searches.

Notes

See Nearest Neighbors in the 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 sklearn.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]]))
[[0.666... 0.333...]]
```

Methods

`fit`(X, y)	Fit the k-nearest neighbors classifier from the training dataset.
`get_metadata_routing`()	Get metadata routing of this object.
`get_params`([deep])	Get parameters for this estimator.
`kneighbors`([X, n_neighbors, return_distance])	Find the K-neighbors of a point.
`kneighbors_graph`([X, n_neighbors, mode])	Compute 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.
`set_score_request`(*[, sample_weight])	Request metadata passed to the `score` method.

fit(*X*, *y*) [source]

Fit the k-nearest neighbors classifier from the training dataset.

Toggle Menu

Parameters:

X : *{array-like, sparse matrix} of shape (n_samples, n_features) or (n_samples, n_samples) if metric='precomputed'*
Training data.

y : *{array-like, sparse matrix} of shape (n_samples,) or (n_samples, n_outputs)*
Target values.

Returns:

self : *KNeighborsClassifier*
The fitted k-nearest neighbors classifier.

`get_metadata_routing`() [source]

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns:

routing : *MetadataRequest*
A `MetadataRequest` encapsulating routing information.

`get_params`(*deep=True*) [source]

Get parameters for this estimator.

Parameters:

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

Returns:

params : *dict*
Parameter names mapped to their values.

`kneighbors`(*X=None*, *n_neighbors=None*, *return_distance=True*) [source]

Find the K-neighbors of a point.

Returns indices of and distances to the neighbors of each point.

Parameters:

X : *{array-like, sparse matrix}, shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed', default=None*
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, default=None*
Number of neighbors required for each sample. The default is the value passed to the constructor.

return_distance : *bool, default=True*
Whether or not to return the distances.

Returns:

neigh_dist : *ndarray of shape (n_queries, n_neighbors)*
Array representing the lengths to points, only present if return_distance=True.

neigh_ind : *ndarray of shape (n_queries, n_neighbors)*
Indices of the nearest points in the population matrix.

Examples

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

Toggle Menu

```
>>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
>>> from sklearn.neighbors import NearestNeighbors
>>> neigh = NearestNeighbors(n_neighbors=1)
>>> neigh.fit(samples)
NearestNeighbors(n_neighbors=1)
>>> 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_graph(*X=None*, *n_neighbors=None*, *mode='connectivity'*) [source]

Compute the (weighted) graph of k-Neighbors for points in X.

Parameters:

X : *{array-like, sparse matrix} of shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed', default=None*

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. For `metric='precomputed'` the shape should be (n_queries, n_indexed). Otherwise the shape should be (n_queries, n_features).

n_neighbors : *int, default=None*

Number of neighbors for each sample. The default is the value passed to the constructor.

mode : *{'connectivity', 'distance'}, default='connectivity'*

Type of returned matrix: 'connectivity' will return the connectivity matrix with ones and zeros, in 'distance' the edges are distances between points, type of distance depends on the selected metric parameter in NearestNeighbors class.

Returns:

A : *sparse-matrix of shape (n_queries, n_samples_fit)*

`n_samples_fit` is the number of samples in the fitted data. `A[i, j]` gives the weight of the edge connecting `i` to `j`. The matrix is of CSR format.

See also:

`NearestNeighbors.radius_neighbors_graph`

Compute the (weighted) graph of Neighbors for points in X.

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*) [source]

Predict the class labels for the provided data.

Toggle Menu

Parameters:

X : *{array-like, sparse matrix} of shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'*

Test samples.

Returns:

y : *ndarray of shape (n_queries,) or (n_queries, n_outputs)*

Class labels for each data sample.

`predict_proba`(*X*) [source]

Return probability estimates for the test data X.

Parameters:

X : *{array-like, sparse matrix} of shape (n_queries, n_features), or (n_queries, n_indexed) if metric == 'precomputed'*

Test samples.

Returns:

p : *ndarray of shape (n_queries, 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*) [source]

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)` w.r.t. `y`.

`set_params`(***params*) [source]

Set the parameters of this estimator.

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

Parameters:

****params : *dict***

Estimator parameters.

Returns:

self : *estimator instance*

Estimator instance.

`set_score_request`(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → KNeighborsClassifier [source]

Request metadata passed to the `score` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config`). Please see User Guide on how the

Toggle Menu hanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `score` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `score`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters:

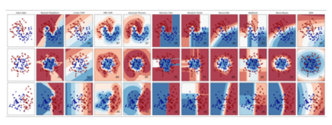
sample_weight : *str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*
Metadata routing for `sample_weight` parameter in `score`.

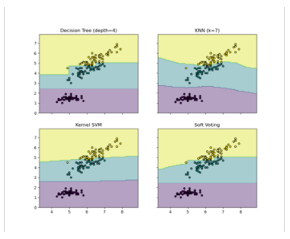
Returns:

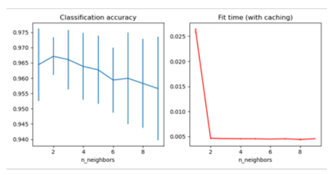
self : *object*
The updated object.

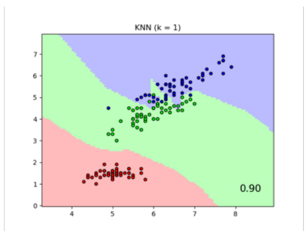
Examples using `sklearn.neighbors.KNeighborsClassifier`

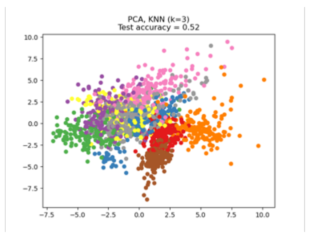
Release Highlights for scikit-learn 0.24

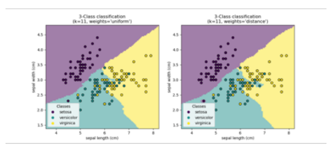

Classifier comparison

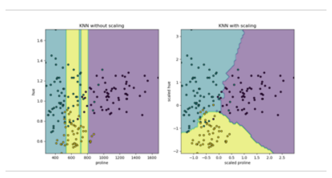

Plot the decision boundaries of a VotingClassifier

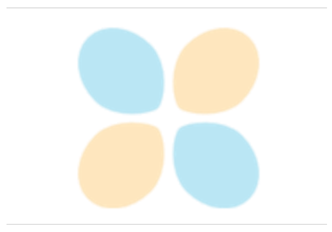

Caching nearest neighbors

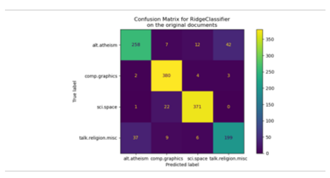

Comparing Nearest Neighbors with and without Neighborhood Components Analysis


Dimensionality Reduction with Neighborhood Components Analysis


Nearest Neighbors Classification


Importance of Feature Scaling


Digits Classification Exercise


Classification of text documents using sparse features

Toggle Menu