CHAPTER

SIX

API REFERENCE

This is the class and function reference of scikit-learn. Please refer to the *full user guide* for further details, as the class and function raw specifications may not be enough to give full guidelines on their uses. For reference on concepts repeated across the API, see *Glossary of Common Terms and API Elements*.

6.1 `sklearn.base`: Base classes and utility functions

Base classes for all estimators.

6.1.1 Base classes

base.BaseEstimator	Base class for all estimators in scikit-learn
base.BiclusterMixin	Mixin class for all bicluster estimators in scikit-learn
base.ClassifierMixin	Mixin class for all classifiers in scikit-learn.
base.ClusterMixin	Mixin class for all cluster estimators in scikit-learn.
base.DensityMixin	Mixin class for all density estimators in scikit-learn.
base.RegressorMixin	Mixin class for all regression estimators in scikit-learn.
base.TransformerMixin	Mixin class for all transformers in scikit-learn.

`sklearn.base.BaseEstimator`

class `sklearn.base.`**`BaseEstimator`**

Base class for all estimators in scikit-learn

Notes

All estimators should specify all the parameters that can be set at the class level in their `__init__` as explicit keyword arguments (no `*args` or `**kwargs`).

Methods

get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

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

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, ***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.

Returns

self

Examples using `sklearn.base.BaseEstimator`

- *Inductive Clustering*
- *Column Transformer with Heterogeneous Data Sources*

sklearn.base.BiclusterMixin

class sklearn.base.**BiclusterMixin**
Mixin class for all bicluster estimators in scikit-learn

Attributes

biclusters_ Convenient way to get row and column indicators together.

Methods

get_indices(self, i)	Row and column indices of the i'th bicluster.
get_shape(self, i)	Shape of the i'th bicluster.
get_submatrix(self, i, data)	Returns the submatrix corresponding to bicluster `i`.

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

biclusters_
Convenient way to get row and column indicators together.

Returns the `rows_` and `columns_` members.

get_indices (*self*, *i*)
Row and column indices of the i'th bicluster.

Only works if `rows_` and `columns_` attributes exist.

Parameters

i [int] The index of the cluster.

Returns

row_ind [np.array, dtype=np.intp] Indices of rows in the dataset that belong to the bicluster.

col_ind [np.array, dtype=np.intp] Indices of columns in the dataset that belong to the bicluster.

get_shape(*self*, *i*)
Shape of the i'th bicluster.

Parameters

i [int] The index of the cluster.

Returns

shape [(int, int)] Number of rows and columns (resp.) in the bicluster.

get_submatrix(*self*, *i*, *data*)
Returns the submatrix corresponding to bicluster `i`.

Parameters

i [int] The index of the cluster.

data [array] The data.

Returns

submatrix [array] The submatrix corresponding to bicluster i.

Notes

Works with sparse matrices. Only works if `rows_` and `columns_` attributes exist.

sklearn.base.ClassifierMixin

class sklearn.base.**ClassifierMixin**
Mixin class for all classifiers in scikit-learn.

Methods

score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.

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

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

sklearn.base.ClusterMixin

class sklearn.base.**ClusterMixin**

Mixin class for all cluster estimators in scikit-learn.

Methods

fit_predict(self, X[, y])	Performs clustering on X and returns cluster labels.

__init__ (*self*, */*, **args*, ***kwargs*)

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

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

Performs clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

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

Returns

labels [ndarray, shape (n_samples,)] cluster labels

sklearn.base.DensityMixin

class sklearn.base.**DensityMixin**

Mixin class for all density estimators in scikit-learn.

Methods

score(self, X[, y])	Returns the score of the model on the data X

__init__ (*self*, */*, **args*, ***kwargs*)

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

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

Returns the score of the model on the data X

Parameters

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

Returns

score [float]

sklearn.base.RegressorMixin

class sklearn.base.**RegressorMixin**
Mixin class for all regression estimators in scikit-learn.

Methods

score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.

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

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

sklearn.base.TransformerMixin

class sklearn.base.**TransformerMixin**
Mixin class for all transformers in scikit-learn.

Methods

fit_transform(self, X[, y])	Fit to data, then transform it.

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

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

Examples using `sklearn.base.TransformerMixin`

- *Column Transformer with Heterogeneous Data Sources*

6.1.2 Functions

base.clone(estimator[, safe])	Constructs a new estimator with the same parameters.
base.is_classifier(estimator)	Returns True if the given estimator is (probably) a classifier.
base.is_regressor(estimator)	Returns True if the given estimator is (probably) a regressor.
config_context(**new_config)	Context manager for global scikit-learn configuration
get_config()	Retrieve current values for configuration set by *set_config*
set_config([assume_finite, working_memory, ...])	Set global scikit-learn configuration
show_versions()	Print useful debugging information

sklearn.base.clone

sklearn.base.**clone** (*estimator*, *safe=True*)
Constructs a new estimator with the same parameters.

Clone does a deep copy of the model in an estimator without actually copying attached data. It yields a new estimator with the same parameters that has not been fit on any data.

Parameters

estimator [estimator object, or list, tuple or set of objects] The estimator or group of estimators to be cloned

safe [boolean, optional] If safe is false, clone will fall back to a deep copy on objects that are not estimators.

sklearn.base.is_classifier

sklearn.base.**is_classifier**(*estimator*)
Returns True if the given estimator is (probably) a classifier.

Parameters

estimator [object] Estimator object to test.

Returns

out [bool] True if estimator is a classifier and False otherwise.

sklearn.base.is_regressor

sklearn.base.**is_regressor**(*estimator*)
Returns True if the given estimator is (probably) a regressor.

Parameters

estimator [object] Estimator object to test.

Returns

out [bool] True if estimator is a regressor and False otherwise.

sklearn.config_context

sklearn.**config_context**(***new_config*)
Context manager for global scikit-learn configuration

Parameters

assume_finite [bool, optional] If True, validation for finiteness will be skipped, saving time, but leading to potential crashes. If False, validation for finiteness will be performed, avoiding error. Global default: False.

working_memory [int, optional] If set, scikit-learn will attempt to limit the size of temporary arrays to this number of MiB (per job when parallelised), often saving both computation time and memory on expensive operations that can be performed in chunks. Global default: 1024.

See also:

set_config Set global scikit-learn configuration

get_config Retrieve current values of the global configuration

Notes

All settings, not just those presently modified, will be returned to their previous values when the context manager is exited. This is not thread-safe.

Examples

```
>>> import sklearn
>>> from sklearn.utils.validation import assert_all_finite
>>> with sklearn.config_context(assume_finite=True):
...     assert_all_finite([float('nan')])
>>> with sklearn.config_context(assume_finite=True):
...     with sklearn.config_context(assume_finite=False):
...         assert_all_finite([float('nan')])
...
Traceback (most recent call last):
...
ValueError: Input contains NaN, ...
```

sklearn.get_config

sklearn.**get_config**()

Retrieve current values for configuration set by *set_config*

Returns

config [dict] Keys are parameter names that can be passed to *set_config*.

See also:

config_context Context manager for global scikit-learn configuration

set_config Set global scikit-learn configuration

sklearn.set_config

sklearn.**set_config**(*assume_finite=None*, *working_memory=None*, *print_changed_only=None*)

Set global scikit-learn configuration

New in version 0.19.

Parameters

assume_finite [bool, optional] If True, validation for finiteness will be skipped, saving time, but leading to potential crashes. If False, validation for finiteness will be performed, avoiding error. Global default: False.

New in version 0.19.

working_memory [int, optional] If set, scikit-learn will attempt to limit the size of temporary arrays to this number of MiB (per job when parallelised), often saving both computation time and memory on expensive operations that can be performed in chunks. Global default: 1024.

New in version 0.20.

print_changed_only [bool, optional] If True, only the parameters that were set to non-default values will be printed when printing an estimator. For example, `print(SVC())` while True will only print 'SVC()' while the default behaviour would be to print 'SVC(C=1.0, cache_size=200, ...)' with all the non-changed parameters.

New in version 0.21.

See also:

config_context Context manager for global scikit-learn configuration

get_config Retrieve current values of the global configuration

Examples using sklearn.set_config

- *Compact estimator representations*

sklearn.show_versions

sklearn.**show_versions**()
Print useful debugging information

6.2 sklearn.calibration: Probability Calibration

Calibration of predicted probabilities.

User guide: See the *Probability calibration* section for further details.

calibration.CalibratedClassifierCV([...])	Probability calibration with isotonic regression or sigmoid.

6.2.1 sklearn.calibration.CalibratedClassifierCV

class sklearn.calibration.**CalibratedClassifierCV**(*base_estimator=None*, *method='sigmoid'*, *cv='warn'*)

Probability calibration with isotonic regression or sigmoid.

See glossary entry for *cross-validation estimator*.

With this class, the base_estimator is fit on the train set of the cross-validation generator and the test set is used for calibration. The probabilities for each of the folds are then averaged for prediction. In case that cv="prefit" is passed to __init__, it is assumed that base_estimator has been fitted already and all data is used for calibration. Note that data for fitting the classifier and for calibrating it must be disjoint.

Read more in the *User Guide*.

Parameters

base_estimator [instance BaseEstimator] The classifier whose output decision function needs to be calibrated to offer more accurate predict_proba outputs. If cv=prefit, the classifier must have been fit already on data.

method ['sigmoid' or 'isotonic'] The method to use for calibration. Can be 'sigmoid' which corresponds to Platt's method or 'isotonic' which is a non-parametric approach. It is not advised to use isotonic calibration with too few calibration samples (<<1000) since it tends to overfit. Use sigmoids (Platt's calibration) in this case.

cv [integer, cross-validation generator, iterable or "prefit", optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if `y` is binary or multiclass, *sklearn.model_selection.StratifiedKFold* is used. If `y` is neither binary nor multiclass, *sklearn.model_selection.KFold* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

If "prefit" is passed, it is assumed that base_estimator has been fitted already and all data is used for calibration.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

Attributes

classes_ [array, shape (n_classes)] The class labels.

calibrated_classifiers_ [list (len() equal to cv or 1 if cv == "prefit")] The list of calibrated classifiers, one for each crossvalidation fold, which has been fitted on all but the validation fold and calibrated on the validation fold.

References

[R57cf438d7060-1], *[R57cf438d7060-2]*, *[R57cf438d7060-3]*, *[R57cf438d7060-4]*

Methods

fit(self, X, y[, sample_weight])	Fit the calibrated model
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the target of new samples.
predict_proba(self, X)	Posterior probabilities of classification
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *base_estimator=None*, *method='sigmoid'*, *cv='warn'*)

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the calibrated model

Parameters

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

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

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted.

Returns

self [object] Returns an instance of self.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)

Predict the target of new samples. Can be different from the prediction of the uncalibrated classifier.

Parameters

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

Returns

C [array, shape (n_samples,)] The predicted class.

predict_proba(*self*, *X*)

Posterior probabilities of classification

This function returns posterior probabilities of classification according to each class on an array of test vectors X.

Parameters

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

Returns

C [array, shape (n_samples, n_classes)] The predicted probas.

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

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.calibration.CalibratedClassifierCV`

- *Probability Calibration curves*
- *Probability calibration of classifiers*
- *Probability Calibration for 3-class classification*

calibration.calibration_curve(y_true, y_prob)	Compute true and predicted probabilities for a calibration curve.

6.2.2 sklearn.calibration.calibration_curve

sklearn.calibration.**calibration_curve**(*y_true*, *y_prob*, *normalize=False*, *n_bins=5*, *strategy='uniform'*)

Compute true and predicted probabilities for a calibration curve.

The method assumes the inputs come from a binary classifier.

Calibration curves may also be referred to as reliability diagrams.

Read more in the *User Guide*.

Parameters

y_true [array, shape (n_samples,)] True targets.

y_prob [array, shape (n_samples,)] Probabilities of the positive class.

normalize [bool, optional, default=False] Whether y_prob needs to be normalized into the bin [0, 1], i.e. is not a proper probability. If True, the smallest value in y_prob is mapped onto 0 and the largest one onto 1.

n_bins [int] Number of bins. A bigger number requires more data. Bins with no data points (i.e. without corresponding values in y_prob) will not be returned, thus there may be fewer than n_bins in the return value.

strategy [{'uniform', 'quantile'}, (default='uniform')] Strategy used to define the widths of the bins.

uniform All bins have identical widths.

quantile All bins have the same number of points.

Returns

prob_true [array, shape (n_bins,) or smaller] The true probability in each bin (fraction of positives).

prob_pred [array, shape (n_bins,) or smaller] The mean predicted probability in each bin.

References

Alexandru Niculescu-Mizil and Rich Caruana (2005) Predicting Good Probabilities With Supervised Learning, in Proceedings of the 22nd International Conference on Machine Learning (ICML). See section 4 (Qualitative Analysis of Predictions).

Examples using sklearn.calibration.calibration_curve

- *Comparison of Calibration of Classifiers*
- *Probability Calibration curves*

6.3 sklearn.cluster: Clustering

The *sklearn.cluster* module gathers popular unsupervised clustering algorithms.

User guide: See the *Clustering* section for further details.

6.3.1 Classes

cluster.AffinityPropagation([damping, ...])	Perform Affinity Propagation Clustering of data.
cluster.AgglomerativeClustering([...])	Agglomerative Clustering
cluster.Birch([threshold, branching_factor, ...])	Implements the Birch clustering algorithm.
cluster.DBSCAN([eps, min_samples, metric, ...])	Perform DBSCAN clustering from vector array or distance matrix.
cluster.OPTICS([min_samples, max_eps, ...])	Estimate clustering structure from vector array
cluster.FeatureAgglomeration([n_clusters, ...])	Agglomerate features.
cluster.KMeans([n_clusters, init, n_init, ...])	K-Means clustering
cluster.MiniBatchKMeans([n_clusters, init, ...])	Mini-Batch K-Means clustering
cluster.MeanShift([bandwidth, seeds, ...])	Mean shift clustering using a flat kernel.
cluster.SpectralClustering([n_clusters, ...])	Apply clustering to a projection of the normalized Laplacian.

sklearn.cluster.AffinityPropagation

class sklearn.cluster.**AffinityPropagation**(*damping=0.5*, *max_iter=200*, *convergence_iter=15*, *copy=True*, *preference=None*, *affinity='euclidean'*, *verbose=False*)

Perform Affinity Propagation Clustering of data.

Read more in the *User Guide*.

Parameters

damping [float, optional, default: 0.5] Damping factor (between 0.5 and 1) is the extent to which the current value is maintained relative to incoming values (weighted 1 - damping). This in order to avoid numerical oscillations when updating these values (messages).

max_iter [int, optional, default: 200] Maximum number of iterations.

convergence_iter [int, optional, default: 15] Number of iterations with no change in the number of estimated clusters that stops the convergence.

copy [boolean, optional, default: True] Make a copy of input data.

preference [array-like, shape (n_samples,) or float, optional] Preferences for each point - points with larger values of preferences are more likely to be chosen as exemplars. The number of exemplars, ie of clusters, is influenced by the input preferences value. If the preferences are not passed as arguments, they will be set to the median of the input similarities.

affinity [string, optional, default=``euclidean``] Which affinity to use. At the moment `precomputed` and `euclidean` are supported. `euclidean` uses the negative squared euclidean distance between points.

verbose [boolean, optional, default: False] Whether to be verbose.

Attributes

cluster_centers_indices_ [array, shape (n_clusters,)] Indices of cluster centers

cluster_centers_ [array, shape (n_clusters, n_features)] Cluster centers (if affinity != `precomputed`).

labels_ [array, shape (n_samples,)] Labels of each point

affinity_matrix_ [array, shape (n_samples, n_samples)] Stores the affinity matrix used in `fit`.

n_iter_ [int] Number of iterations taken to converge.

Notes

For an example, see *examples/cluster/plot_affinity_propagation.py*.

The algorithmic complexity of affinity propagation is quadratic in the number of points.

When `fit` does not converge, `cluster_centers_` becomes an empty array and all training samples will be labelled as `-1`. In addition, `predict` will then label every sample as `-1`.

When all training samples have equal similarities and equal preferences, the assignment of cluster centers and labels depends on the preference. If the preference is smaller than the similarities, `fit` will result in a single cluster center and label `0` for every sample. Otherwise, every training sample becomes its own cluster center and is assigned a unique label.

References

Brendan J. Frey and Delbert Dueck, "Clustering by Passing Messages Between Data Points", Science Feb. 2007

Examples

```
>>> from sklearn.cluster import AffinityPropagation
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [4, 2], [4, 4], [4, 0]])
>>> clustering = AffinityPropagation().fit(X)
>>> clustering
AffinityPropagation(affinity='euclidean', convergence_iter=15, copy=True,
          damping=0.5, max_iter=200, preference=None, verbose=False)
>>> clustering.labels_
array([0, 0, 0, 1, 1, 1])
>>> clustering.predict([[0, 0], [4, 4]])
array([0, 1])
>>> clustering.cluster_centers_
array([[1, 2],
       [4, 2]])
```

Methods

fit(self, X[, y])	Create affinity matrix from negative euclidean distances, then apply affinity propagation clustering.
fit_predict(self, X[, y])	Performs clustering on X and returns cluster labels.

Continued on next page

Table 6.14 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the closest cluster each sample in X belongs to.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *damping=0.5*, *max_iter=200*, *convergence_iter=15*, *copy=True*, *preference=None*, *affinity='euclidean'*, *verbose=False*)

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

Create affinity matrix from negative euclidean distances, then apply affinity propagation clustering.

Parameters

X [array-like, shape (n_samples, n_features) or (n_samples, n_samples)] Data matrix or, if affinity is `precomputed`, matrix of similarities / affinities.

y [Ignored]

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

Performs clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

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

Returns

labels [ndarray, shape (n_samples,)] cluster labels

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)

Predict the closest cluster each sample in X belongs to.

Parameters

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

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.cluster.AffinityPropagation`

- *Demo of affinity propagation clustering algorithm*
- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster`.AgglomerativeClustering

class `sklearn.cluster.`**AgglomerativeClustering**(*n_clusters=2*, *affinity='euclidean'*, *memory=None*, *connectivity=None*, *compute_full_tree='auto'*, *linkage='ward'*, *pooling_func='deprecated'*, *distance_threshold=None*)

Agglomerative Clustering

Recursively merges the pair of clusters that minimally increases a given linkage distance.

Read more in the *User Guide*.

Parameters

n_clusters [int or None, optional (default=2)] The number of clusters to find. It must be `None` if `distance_threshold` is not `None`.

affinity [string or callable, default: "euclidean"] Metric used to compute the linkage. Can be "euclidean", "l1", "l2", "manhattan", "cosine", or "precomputed". If linkage is "ward", only "euclidean" is accepted. If "precomputed", a distance matrix (instead of a similarity matrix) is needed as input for the fit method.

memory [None, str or object with the joblib.Memory interface, optional] Used to cache the output of the computation of the tree. By default, no caching is done. If a string is given, it is the path to the caching directory.

connectivity [array-like or callable, optional] Connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. This can be a connectivity matrix itself or a callable that transforms the data into a connectivity matrix, such as derived from kneighbors_graph. Default is None, i.e, the hierarchical clustering algorithm is unstructured.

compute_full_tree [bool or 'auto' (optional)] Stop early the construction of the tree at n_clusters. This is useful to decrease computation time if the number of clusters is not small compared to the number of samples. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of clusters and using caching, it may be advantageous to compute the full tree. It must be `True` if `distance_threshold` is not `None`.

linkage [{"ward", "complete", "average", "single"}, optional (default="ward")] Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.

- ward minimizes the variance of the clusters being merged.
- average uses the average of the distances of each observation of the two sets.
- complete or maximum linkage uses the maximum distances between all observations of the two sets.
- single uses the minimum of the distances between all observations of the two sets.

pooling_func [callable, default='deprecated'] Ignored.

Deprecated since version 0.20: `pooling_func` has been deprecated in 0.20 and will be removed in 0.22.

distance_threshold [float, optional (default=None)] The linkage distance threshold above which, clusters will not be merged. If not `None`, `n_clusters` must be `None` and `compute_full_tree` must be `True`.

New in version 0.21.

Attributes

n_clusters_ [int] The number of clusters found by the algorithm. If `distance_threshold=None`, it will be equal to the given `n_clusters`.

labels_ [array [n_samples]] cluster labels for each point

n_leaves_ [int] Number of leaves in the hierarchical tree.

n_connected_components_ [int] The estimated number of connected components in the graph.

children_ [array-like, shape (n_samples-1, 2)] The children of each non-leaf node. Values less than *n_samples* correspond to leaves of the tree which are the original samples. A node `i` greater than or equal to *n_samples* is a non-leaf node and has children `children_[i - n_samples]`. Alternatively at the i-th iteration, children[i][0] and children[i][1] are merged to form node `n_samples + i`

Examples

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [4, 2], [4, 4], [4, 0]])
>>> clustering = AgglomerativeClustering().fit(X)
>>> clustering
AgglomerativeClustering(affinity='euclidean', compute_full_tree='auto',
                        connectivity=None, distance_threshold=None,
                        linkage='ward', memory=None, n_clusters=2,
                        pooling_func='deprecated')
>>> clustering.labels_
array([1, 1, 1, 0, 0, 0])
```

Methods

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

__init__(*self*, *n_clusters=2*, *affinity='euclidean'*, *memory=None*, *connectivity=None*, *compute_full_tree='auto'*, *linkage='ward'*, *pooling_func='deprecated'*, *distance_threshold=None*)

fit(*self*, *X*, *y=None*)
Fit the hierarchical clustering on the data

Parameters

X [array-like, shape = [n_samples, n_features]] Training data. Shape [n_samples, n_features], or [n_samples, n_samples] if affinity=='precomputed'.

y [Ignored]

Returns

self

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

Performs clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

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

Returns

labels [ndarray, shape (n_samples,)] cluster labels

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, ***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.

Returns

self

Examples using `sklearn.cluster.AgglomerativeClustering`

- *Agglomerative clustering with and without structure*
- *Various Agglomerative Clustering on a 2D embedding of digits*
- *A demo of structured Ward hierarchical clustering on an image of coins*
- *Hierarchical clustering: structured vs unstructured ward*
- *Agglomerative clustering with different metrics*
- *Inductive Clustering*
- *Comparing different hierarchical linkage methods on toy datasets*
- *Comparing different clustering algorithms on toy datasets*

sklearn.cluster.Birch

class sklearn.cluster.**Birch**(*threshold=0.5*, *branching_factor=50*, *n_clusters=3*, *compute_labels=True*, *copy=True*)

Implements the Birch clustering algorithm.

It is a memory-efficient, online-learning algorithm provided as an alternative to *MiniBatchKMeans*. It constructs a tree data structure with the cluster centroids being read off the leaf. These can be either the final cluster centroids or can be provided as input to another clustering algorithm such as *AgglomerativeClustering*.

Read more in the *User Guide*.

Parameters

threshold [float, default 0.5] The radius of the subcluster obtained by merging a new sample and the closest subcluster should be lesser than the threshold. Otherwise a new subcluster is started. Setting this value to be very low promotes splitting and vice-versa.

branching_factor [int, default 50] Maximum number of CF subclusters in each node. If a new samples enters such that the number of subclusters exceed the branching_factor then that node is split into two nodes with the subclusters redistributed in each. The parent subcluster of that node is removed and two new subclusters are added as parents of the 2 split nodes.

n_clusters [int, instance of sklearn.cluster model, default 3] Number of clusters after the final clustering step, which treats the subclusters from the leaves as new samples.

- `None` : the final clustering step is not performed and the subclusters are returned as they are.
- *sklearn.cluster* Estimator : If a model is provided, the model is fit treating the subclusters as new samples and the initial data is mapped to the label of the closest subcluster.
- `int` : the model fit is *AgglomerativeClustering* with `n_clusters` set to be equal to the int.

compute_labels [bool, default True] Whether or not to compute labels for each fit.

copy [bool, default True] Whether or not to make a copy of the given data. If set to False, the initial data will be overwritten.

Attributes

root_ [_CFNode] Root of the CFTree.

dummy_leaf_ [_CFNode] Start pointer to all the leaves.

subcluster_centers_ [ndarray,] Centroids of all subclusters read directly from the leaves.

subcluster_labels_ [ndarray,] Labels assigned to the centroids of the subclusters after they are clustered globally.

labels_ [ndarray, shape (n_samples,)] Array of labels assigned to the input data. if partial_fit is used instead of fit, they are assigned to the last batch of data.

Notes

The tree data structure consists of nodes with each node consisting of a number of subclusters. The maximum number of subclusters in a node is determined by the branching factor. Each subcluster maintains a linear sum, squared sum and the number of samples in that subcluster. In addition, each subcluster can also have a node as its child, if the subcluster is not a member of a leaf node.

For a new point entering the root, it is merged with the subcluster closest to it and the linear sum, squared sum and the number of samples of that subcluster are updated. This is done recursively till the properties of the leaf node are updated.

References

- Tian Zhang, Raghu Ramakrishnan, Maron Livny BIRCH: An efficient data clustering method for large databases. https://www.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf
- Roberto Perdisci JBirch - Java implementation of BIRCH clustering algorithm https://code.google.com/archive/p/jbirch

Examples

```
>>> from sklearn.cluster import Birch
>>> X = [[0, 1], [0.3, 1], [-0.3, 1], [0, -1], [0.3, -1], [-0.3, -1]]
>>> brc = Birch(branching_factor=50, n_clusters=None, threshold=0.5,
... compute_labels=True)
>>> brc.fit(X)
Birch(branching_factor=50, compute_labels=True, copy=True, n_clusters=None,
   threshold=0.5)
>>> brc.predict(X)
array([0, 0, 0, 1, 1, 1])
```

Methods

fit(self, X[, y])	Build a CF Tree for the input data.
fit_predict(self, X[, y])	Performs clustering on X and returns cluster labels.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self[, X, y])	Online learning.
predict(self, X)	Predict data using the `centroids_` of subclusters.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X into subcluster centroids dimension.

__init__(*self*, *threshold=0.5*, *branching_factor=50*, *n_clusters=3*, *compute_labels=True*, *copy=True*)

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

Build a CF Tree for the input data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data.

y [Ignored]

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

Performs clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

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

Returns

labels [ndarray, shape (n_samples,)] cluster labels

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X=None*, *y=None*)
Online learning. Prevents rebuilding of CFTree from scratch.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features), None] Input data. If X is not provided, only the global clustering step is done.

y [Ignored]

predict(*self*, *X*)
Predict data using the `centroids_` of subclusters.

Avoid computation of the row norms of X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data.

Returns

labels [ndarray, shape(n_samples)] Labelled data.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform X into subcluster centroids dimension.

Each dimension represents the distance from the sample point to each cluster centroid.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data.

Returns

X_trans [{array-like, sparse matrix}, shape (n_samples, n_clusters)] Transformed data.

Examples using `sklearn.cluster.Birch`

- *Compare BIRCH and MiniBatchKMeans*
- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster`.DBSCAN

class sklearn.cluster.**DBSCAN**(*eps=0.5*, *min_samples=5*, *metric='euclidean'*, *metric_params=None*, *algorithm='auto'*, *leaf_size=30*, *p=None*, *n_jobs=None*)
Perform DBSCAN clustering from vector array or distance matrix.

DBSCAN - Density-Based Spatial Clustering of Applications with Noise. Finds core samples of high density and expands clusters from them. Good for data which contains clusters of similar density.

Read more in the *User Guide*.

Parameters

eps [float, optional] The maximum distance between two samples for one to be considered as in the neighborhood of the other. This is not a maximum bound on the distances of points within a cluster. This is the most important DBSCAN parameter to choose appropriately for your data set and distance function.

min_samples [int, optional] The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself.

metric [string, 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 `sklearn.metrics.pairwise_distances` for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square. X may be a sparse matrix, in which case only "nonzero" elements may be considered neighbors for DBSCAN.

New in version 0.17: metric *precomputed* to accept precomputed sparse matrix.

metric_params [dict, optional] Additional keyword arguments for the metric function.

New in version 0.19.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] The algorithm to be used by the NearestNeighbors module to compute pointwise distances and find nearest neighbors. See NearestNeighbors module documentation for details.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or cKDTree. 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, optional] The power of the Minkowski metric to be used to calculate distance between points.

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

Attributes

core_sample_indices_ [array, shape = [n_core_samples]] Indices of core samples.

components_ [array, shape = [n_core_samples, n_features]] Copy of each core sample found by training.

labels_ [array, shape = [n_samples]] Cluster labels for each point in the dataset given to fit(). Noisy samples are given the label -1.

See also:

OPTICS A similar clustering at multiple values of eps. Our implementation is optimized for memory usage.

Notes

For an example, see *examples/cluster/plot_dbscan.py*.

This implementation bulk-computes all neighborhood queries, which increases the memory complexity to O(n.d) where d is the average number of neighbors, while original DBSCAN had memory complexity O(n). It may attract a higher memory complexity when querying these nearest neighborhoods, depending on the `algorithm`.

One way to avoid the query complexity is to pre-compute sparse neighborhoods in chunks using *NearestNeighbors.radius_neighbors_graph* with `mode='distance'`, then using `metric='precomputed'` here.

Another way to reduce memory and computation time is to remove (near-)duplicate points and use `sample_weight` instead.

`cluster.OPTICS` provides a similar clustering with lower memory usage.

References

Ester, M., H. P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226-231. 1996

Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017). DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. ACM Transactions on Database Systems (TODS), 42(3), 19.

Examples

```
>>> from sklearn.cluster import DBSCAN
>>> import numpy as np
>>> X = np.array([[1, 2], [2, 2], [2, 3],
...               [8, 7], [8, 8], [25, 80]])
>>> clustering = DBSCAN(eps=3, min_samples=2).fit(X)
>>> clustering.labels_
```

```
array([ 0,  0,  0,  1,  1, -1])
>>> clustering
DBSCAN(algorithm='auto', eps=3, leaf_size=30, metric='euclidean',
    metric_params=None, min_samples=2, n_jobs=None, p=None)
```

Methods

fit(self, X[, y, sample_weight])	Perform DBSCAN clustering from features or distance matrix.
fit_predict(self, X[, y, sample_weight])	Performs clustering on X and returns cluster labels.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *eps=0.5*, *min_samples=5*, *metric='euclidean'*, *metric_params=None*, *algorithm='auto'*, *leaf_size=30*, *p=None*, *n_jobs=None*)

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

Perform DBSCAN clustering from features or distance matrix.

Parameters

X [array or sparse (CSR) matrix of shape (n_samples, n_features), or array of shape (n_samples, n_samples)] A feature array, or array of distances between samples if `metric='precomputed'`.

sample_weight [array, shape (n_samples,), optional] Weight of each sample, such that a sample with a weight of at least `min_samples` is by itself a core sample; a sample with negative weight may inhibit its eps-neighbor from being core. Note that weights are absolute, and default to 1.

y [Ignored]

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

Performs clustering on X and returns cluster labels.

Parameters

X [array or sparse (CSR) matrix of shape (n_samples, n_features), or array of shape (n_samples, n_samples)] A feature array, or array of distances between samples if `metric='precomputed'`.

sample_weight [array, shape (n_samples,), optional] Weight of each sample, such that a sample with a weight of at least `min_samples` is by itself a core sample; a sample with negative weight may inhibit its eps-neighbor from being core. Note that weights are absolute, and default to 1.

y [Ignored]

Returns

y [ndarray, shape (n_samples,)] cluster labels

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

Examples using `sklearn.cluster.DBSCAN`

- *Demo of DBSCAN clustering algorithm*
- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster`.OPTICS

class `sklearn.cluster.`**OPTICS**(*min_samples=5*, *max_eps=inf*, *metric='minkowski'*, *p=2*, *metric_params=None*, *cluster_method='xi'*, *eps=None*, *xi=0.05*, *predecessor_correction=True*, *min_cluster_size=None*, *algorithm='auto'*, *leaf_size=30*, *n_jobs=None*)
Estimate clustering structure from vector array

OPTICS (Ordering Points To Identify the Clustering Structure), closely related to DBSCAN, finds core sample of high density and expands clusters from them *[R2c55e37003fe-1]*. Unlike DBSCAN, keeps cluster hierarchy for a variable neighborhood radius. Better suited for usage on large datasets than the current sklearn implementation of DBSCAN.

Clusters are then extracted using a DBSCAN-like method (cluster_method = 'dbscan') or an automatic technique proposed in *[R2c55e37003fe-1]* (cluster_method = 'xi').

This implementation deviates from the original OPTICS by first performing k-nearest-neighborhood searches on all points to identify core sizes, then computing only the distances to unprocessed points when constructing the cluster order. Note that we do not employ a heap to manage the expansion candidates, so the time complexity will be O(n^2).

Read more in the *User Guide*.

Parameters

min_samples [int > 1 or float between 0 and 1 (default=5)] The number of samples in a neighborhood for a point to be considered as a core point. Also, up and down steep regions can't have more then `min_samples` consecutive non-steep points. Expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2).

max_eps [float, optional (default=np.inf)] The maximum distance between two samples for one to be considered as in the neighborhood of the other. Default value of `np.inf` will identify clusters across all scales; reducing `max_eps` will result in shorter run times.

metric [string or callable, optional (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. If metric is "precomputed", X is assumed to be a distance matrix and must be square.

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_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.

cluster_method [string, optional (default='xi')] The extraction method used to extract clusters using the calculated reachability and ordering. Possible values are "xi" and "dbscan".

eps [float, optional (default=None)] The maximum distance between two samples for one to be considered as in the neighborhood of the other. By default it assumes the same value as `max_eps`. Used only when `cluster_method='dbscan'`.

xi [float, between 0 and 1, optional (default=0.05)] Determines the minimum steepness on the reachability plot that constitutes a cluster boundary. For example, an upwards point in the reachability plot is defined by the ratio from one point to its successor being at most 1-xi. Used only when `cluster_method='xi'`.

predecessor_correction [bool, optional (default=True)] Correct clusters according to the predecessors calculated by OPTICS *[R2c55e37003fe-2]*. This parameter has minimal effect on most datasets. Used only when `cluster_method='xi'`.

min_cluster_size [int > 1 or float between 0 and 1 (default=None)] Minimum number of samples in an OPTICS cluster, expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2). If `None`, the value of `min_samples` is used instead. Used only when `cluster_method='xi'`.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] 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. (default)

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

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.

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.

Attributes

labels_ [array, shape (n_samples,)] Cluster labels for each point in the dataset given to fit(). Noisy samples and points which are not included in a leaf cluster of `cluster_hierarchy_` are labeled as -1.

reachability_ [array, shape (n_samples,)] Reachability distances per sample, indexed by object order. Use `clust.reachability_[clust.ordering_]` to access in cluster order.

ordering_ [array, shape (n_samples,)] The cluster ordered list of sample indices.

core_distances_ [array, shape (n_samples,)] Distance at which each sample becomes a core point, indexed by object order. Points which will never be core have a distance of inf. Use `clust.core_distances_[clust.ordering_]` to access in cluster order.

predecessor_ [array, shape (n_samples,)] Point that a sample was reached from, indexed by object order. Seed points have a predecessor of -1.

cluster_hierarchy_ [array, shape (n_clusters, 2)] The list of clusters in the form of `[start, end]` in each row, with all indices inclusive. The clusters are ordered according to `(end, -start)` (ascending) so that larger clusters encompassing smaller clusters come after those smaller ones. Since `labels_` does not reflect the hierarchy, usually `len(cluster_hierarchy_) > np.unique(optics.labels_)`. Please also note that these indices are of the `ordering_`, i.e. `X[ordering_][start:end + 1]` form a cluster. Only available when `cluster_method='xi'`.

See also:

DBSCAN A similar clustering for a specified neighborhood radius (eps). Our implementation is optimized for runtime.

References

[R2c55e37003fe-1], *[R2c55e37003fe-2]*

Methods

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

__init__(*self*, *min_samples=5*, *max_eps=inf*, *metric='minkowski'*, *p=2*, *metric_params=None*, *cluster_method='xi'*, *eps=None*, *xi=0.05*, *predecessor_correction=True*, *min_cluster_size=None*, *algorithm='auto'*, *leaf_size=30*, *n_jobs=None*)

fit(*self*, *X*, *y=None*)
Perform OPTICS clustering

Extracts an ordered list of points and reachability distances, and performs initial clustering using `max_eps` distance specified at OPTICS object instantiation.

Parameters

X [array, shape (n_samples, n_features), or (n_samples, n_samples) if metric='precomputed'.] A feature array, or array of distances between samples if metric='precomputed'.

y [ignored]

Returns

self [instance of OPTICS] The instance.

fit_predict(*self*, *X*, *y=None*)
Performs clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

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

Returns

labels [ndarray, shape (n_samples,)] cluster labels

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

Examples using `sklearn.cluster.OPTICS`

- *Demo of OPTICS clustering algorithm*
- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster`.FeatureAgglomeration

class sklearn.cluster.**FeatureAgglomeration**(*n_clusters=2*, *affinity='euclidean'*, *memory=None*, *connectivity=None*, *compute_full_tree='auto'*, *linkage='ward'*, *pooling_func=<function mean>*, *distance_threshold=None*)
Agglomerate features.

Similar to AgglomerativeClustering, but recursively merges features instead of samples.

Read more in the *User Guide*.

Parameters

n_clusters [int or None, optional (default=2)] The number of clusters to find. It must be `None` if `distance_threshold` is not `None`.

affinity [string or callable, default "euclidean"] Metric used to compute the linkage. Can be "euclidean", "l1", "l2", "manhattan", "cosine", or 'precomputed'. If linkage is "ward", only "euclidean" is accepted.

memory [None, str or object with the joblib.Memory interface, optional] Used to cache the output of the computation of the tree. By default, no caching is done. If a string is given, it is the path to the caching directory.

connectivity [array-like or callable, optional] Connectivity matrix. Defines for each feature the neighboring features following a given structure of the data. This can be a connectivity matrix itself or a callable that transforms the data into a connectivity matrix, such as derived from kneighbors_graph. Default is None, i.e, the hierarchical clustering algorithm is unstructured.

compute_full_tree [bool or 'auto', optional, default "auto"] Stop early the construction of the tree at n_clusters. This is useful to decrease computation time if the number of clusters is not small compared to the number of features. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of clusters and using caching, it may be advantageous to compute the full tree. It must be `True` if `distance_threshold` is not `None`.

linkage [{"ward", "complete", "average", "single"}, optional (default="ward")] Which linkage criterion to use. The linkage criterion determines which distance to use between sets of features. The algorithm will merge the pairs of cluster that minimize this criterion.

- ward minimizes the variance of the clusters being merged.
- average uses the average of the distances of each feature of the two sets.
- complete or maximum linkage uses the maximum distances between all features of the two sets.
- single uses the minimum of the distances between all observations of the two sets.

pooling_func [callable, default np.mean] This combines the values of agglomerated features into a single value, and should accept an array of shape [M, N] and the keyword argument `axis=1`, and reduce it to an array of size [M].

distance_threshold [float, optional (default=None)] The linkage distance threshold above which, clusters will not be merged. If not `None`, `n_clusters` must be `None` and `compute_full_tree` must be `True`.

New in version 0.21.

Attributes

n_clusters_ [int] The number of clusters found by the algorithm. If `distance_threshold=None`, it will be equal to the given `n_clusters`.

labels_ [array-like, (n_features,)] cluster labels for each feature.

n_leaves_ [int] Number of leaves in the hierarchical tree.

n_connected_components_ [int] The estimated number of connected components in the graph.

children_ [array-like, shape (n_nodes-1, 2)] The children of each non-leaf node. Values less than *n_features* correspond to leaves of the tree which are the original samples. A node

`i` greater than or equal to *n_features* is a non-leaf node and has children `children_[i - n_features]`. Alternatively at the i-th iteration, children[i][0] and children[i][1] are merged to form node `n_features + i`

Examples

```
>>> import numpy as np
>>> from sklearn import datasets, cluster
>>> digits = datasets.load_digits()
>>> images = digits.images
>>> X = np.reshape(images, (len(images), -1))
>>> agglo = cluster.FeatureAgglomeration(n_clusters=32)
>>> agglo.fit(X)
FeatureAgglomeration(affinity='euclidean', compute_full_tree='auto',
            connectivity=None, distance_threshold=None, linkage='ward',
            memory=None, n_clusters=32,
            pooling_func=...)
>>> X_reduced = agglo.transform(X)
>>> X_reduced.shape
(1797, 32)
```

Methods

fit(self, X[, y])	Fit the hierarchical clustering on the data
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Xred)	Inverse the transformation.
pooling_func(a[, axis, dtype, out, keepdims])	Compute the arithmetic mean along the specified axis.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform a new matrix using the built clustering

__init__(*self*, *n_clusters=2*, *affinity='euclidean'*, *memory=None*, *connectivity=None*, *compute_full_tree='auto'*, *linkage='ward'*, *pooling_func=<function mean at 0x7f3c23df3400>*, *distance_threshold=None*)

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

Fit the hierarchical clustering on the data

Parameters

X [array-like, shape = [n_samples, n_features]] The data

y [Ignored]

Returns

self

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *Xred*)
Inverse the transformation. Return a vector of size nb_features with the values of Xred assigned to each group of features

Parameters

Xred [array-like, shape=[n_samples, n_clusters] or [n_clusters,]] The values to be assigned to each cluster of samples

Returns

X [array, shape=[n_samples, n_features] or [n_features]] A vector of size n_samples with the values of Xred assigned to each of the cluster of samples.

pooling_func(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*)
Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. `float64` intermediate and return values are used for integer inputs.

Parameters

a [array_like] Array containing numbers whose mean is desired. If `a` is not an array, a conversion is attempted.

axis [None or int or tuple of ints, optional] Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

New in version 1.7.0.

If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as before.

dtype [data-type, optional] Type to use in computing the mean. For integer inputs, the default is `float64`; for floating point inputs, it is the same as the input dtype.

out [ndarray, optional] Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See `doc.ufuncs` for details.

keepdims [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then `keepdims` will not be passed through to the `mean` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement `keepdims` any exceptions will be raised.

Returns

m [ndarray, see dtype parameter above] If `out=None`, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See also:

`average` Weighted average

`std`, `var`, `nanmean`, `nanstd`, `nanvar`

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `float32` (see example below). Specifying a higher-precision accumulator using the *dtype* keyword can alleviate this issue.

By default, `float16` results are computed using `float32` intermediates for extra precision.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([ 2.,  3.])
>>> np.mean(a, axis=1)
array([ 1.5,  3.5])
```

In single precision, `mean` can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.54999924
```

Computing the mean in float64 is more accurate:

```
>>> np.mean(a, dtype=np.float64)
0.55000000074505806
```

`set_params`(*self*, ***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.

Returns

self

`transform`(*self*, *X*)
Transform a new matrix using the built clustering

Parameters

X [array-like, shape = [n_samples, n_features] or [n_features]] A M by N array of M observations in N dimensions or a length M array of M one-dimensional observations.

Returns

Y [array, shape = [n_samples, n_clusters] or [n_clusters]] The pooled values for each feature cluster.

Examples using `sklearn.cluster.FeatureAgglomeration`

- *Feature agglomeration*
- *Feature agglomeration vs. univariate selection*

`sklearn.cluster.KMeans`

class `sklearn.cluster.`**`KMeans`**(*n_clusters=8*, *init='k-means++'*, *n_init=10*, *max_iter=300*, *tol=0.0001*, *precompute_distances='auto'*, *verbose=0*, *random_state=None*, *copy_x=True*, *n_jobs=None*, *algorithm='auto'*)

K-Means clustering

Read more in the *User Guide*.

Parameters

n_clusters [int, optional, default: 8] The number of clusters to form as well as the number of centroids to generate.

init [{'k-means++', 'random' or an ndarray}] Method for initialization, defaults to 'k-means++':

'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

'random': choose k observations (rows) at random from data for the initial centroids.

If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

n_init [int, default: 10] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

max_iter [int, default: 300] Maximum number of iterations of the k-means algorithm for a single run.

tol [float, default: 1e-4] Relative tolerance with regards to inertia to declare convergence

precompute_distances [{'auto', True, False}] Precompute distances (faster but takes more memory).

'auto' : do not precompute distances if n_samples * n_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.

True : always precompute distances

False : never precompute distances

verbose [int, default 0] Verbosity mode.

random_state [int, RandomState instance or None (default)] Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. See *Glossary*.

copy_x [boolean, optional] When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True (default), then the original data is not modified, ensuring X is C-contiguous. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean, in this case it will also not ensure that data is C-contiguous which may cause a significant slowdown.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

algorithm ["auto", "full" or "elkan", default="auto"] K-means algorithm to use. The classical EM-style algorithm is "full". The "elkan" variation is more efficient by using the triangle inequality, but currently doesn't support sparse data. "auto" chooses "elkan" for dense data and "full" for sparse data.

Attributes

cluster_centers_ [array, [n_clusters, n_features]] Coordinates of cluster centers. If the algorithm stops before fully converging (see `tol` and `max_iter`), these will not be consistent with `labels_`.

labels_ : Labels of each point

inertia_ [float] Sum of squared distances of samples to their closest cluster center.

n_iter_ [int] Number of iterations run.

See also:

MiniBatchKMeans Alternative online implementation that does incremental updates of the centers positions using mini-batches. For large scale learning (say n_samples > 10k) MiniBatchKMeans is probably much faster than the default batch implementation.

Notes

The k-means problem is solved using either Lloyd's or Elkan's algorithm.

The average complexity is given by O(k n T), were n is the number of samples and T is the number of iteration.

The worst case complexity is given by O(n^(k+2/p)) with n = n_samples, p = n_features. (D. Arthur and S. Vassilvitskii, 'How slow is the k-means method?' SoCG2006)

In practice, the k-means algorithm is very fast (one of the fastest clustering algorithms available), but it falls in local minima. That's why it can be useful to restart it several times.

If the algorithm stops before fully converging (because of `tol` or `max_iter`), `labels_` and `cluster_centers_` will not be consistent, i.e. the `cluster_centers_` will not be the means of the points in each cluster. Also, the estimator will reassign `labels_` after the last iteration to make `labels_` consistent with `predict` on the training set.

Examples

```
>>> from sklearn.cluster import KMeans
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [10, 2], [10, 4], [10, 0]])
>>> kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
>>> kmeans.labels_
array([1, 1, 1, 0, 0, 0], dtype=int32)
>>> kmeans.predict([[0, 0], [12, 3]])
array([1, 0], dtype=int32)
>>> kmeans.cluster_centers_
array([[10.,  2.],
       [ 1.,  2.]])
```

Methods

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

__init__(*self*, *n_clusters=8*, *init='k-means++'*, *n_init=10*, *max_iter=300*, *tol=0.0001*, *precompute_distances='auto'*, *verbose=0*, *random_state=None*, *copy_x=True*, *n_jobs=None*, *algorithm='auto'*)

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

Compute k-means clustering.

Parameters

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

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

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

Compute cluster centers and predict cluster index for each sample.

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

Parameters

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

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

Returns

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

fit_transform(*self*, *X*, *y=None*, *sample_weight=None*)
Compute clustering and transform X to cluster-distance space.

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

Parameters

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

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

Returns

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

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*, *sample_weight=None*)
Predict the closest cluster each sample in X belongs to.

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

Parameters

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

Returns

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

score(*self*, *X*, *y=None*, *sample_weight=None*)
Opposite of the value of X on the K-means objective.

Parameters

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

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

Returns

score [float] Opposite of the value of X on the K-means objective.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform X to a cluster-distance space.

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

Parameters

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

Returns

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

Examples using sklearn.cluster.KMeans

- *Demonstration of k-means assumptions*
- *Vector Quantization Example*
- *K-means Clustering*
- *Color Quantization using K-Means*
- *Empirical evaluation of the impact of k-means initialization*
- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*
- *A demo of K-Means clustering on the handwritten digits data*
- *Selecting the number of clusters with silhouette analysis on KMeans clustering*
- *Clustering text documents using k-means*

sklearn.cluster.MiniBatchKMeans

class sklearn.cluster.**MiniBatchKMeans**(*n_clusters=8*, *init='k-means++'*, *max_iter=100*, *batch_size=100*, *verbose=0*, *compute_labels=True*, *random_state=None*, *tol=0.0*, *max_no_improvement=10*, *init_size=None*, *n_init=3*, *reassignment_ratio=0.01*)

Mini-Batch K-Means clustering

Read more in the *User Guide*.

Parameters

n_clusters [int, optional, default: 8] The number of clusters to form as well as the number of centroids to generate.

init [{'k-means++', 'random' or an ndarray}, default: 'k-means++'] Method for initialization, defaults to 'k-means++':

'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

'random': choose k observations (rows) at random from data for the initial centroids.

If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

max_iter [int, optional] Maximum number of iterations over the complete dataset before stopping independently of any early stopping criterion heuristics.

batch_size [int, optional, default: 100] Size of the mini batches.

verbose [boolean, optional] Verbosity mode.

compute_labels [boolean, default=True] Compute label assignment and inertia for the complete dataset once the minibatch optimization has converged in fit.

random_state [int, RandomState instance or None (default)] Determines random number generation for centroid initialization and random reassignment. Use an int to make the randomness deterministic. See *Glossary*.

tol [float, default: 0.0] Control early stopping based on the relative center changes as measured by a smoothed, variance-normalized of the mean center squared position changes. This early stopping heuristics is closer to the one used for the batch variant of the algorithms but induces a slight computational and memory overhead over the inertia heuristic.

To disable convergence detection based on normalized center change, set tol to 0.0 (default).

max_no_improvement [int, default: 10] Control early stopping based on the consecutive number of mini batches that does not yield an improvement on the smoothed inertia.

To disable convergence detection based on inertia, set max_no_improvement to None.

init_size [int, optional, default: 3 * batch_size] Number of samples to randomly sample for speeding up the initialization (sometimes at the expense of accuracy): the only algorithm is initialized by running a batch KMeans on a random subset of the data. This needs to be larger than n_clusters.

n_init [int, default=3] Number of random initializations that are tried. In contrast to KMeans, the algorithm is only run once, using the best of the `n_init` initializations as measured by inertia.

reassignment_ratio [float, default: 0.01] Control the fraction of the maximum number of counts for a center to be reassigned. A higher value means that low count centers are more easily reassigned, which means that the model will take longer to converge, but should converge in a better clustering.

Attributes

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

labels_ : Labels of each point (if compute_labels is set to True).

inertia_ [float] The value of the inertia criterion associated with the chosen partition (if compute_labels is set to True). The inertia is defined as the sum of square distances of samples to their nearest neighbor.

See also:

KMeans The classic implementation of the clustering method based on the Lloyd's algorithm. It consumes the whole set of input data at each iteration.

Notes

See https://www.eecs.tufts.edu/~dsculley/papers/fastkmeans.pdf

Examples

```
>>> from sklearn.cluster import MiniBatchKMeans
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [4, 2], [4, 0], [4, 4],
...               [4, 5], [0, 1], [2, 2],
...               [3, 2], [5, 5], [1, -1]])
>>> # manually fit on batches
>>> kmeans = MiniBatchKMeans(n_clusters=2,
...                          random_state=0,
...                          batch_size=6)
>>> kmeans = kmeans.partial_fit(X[0:6,:])
>>> kmeans = kmeans.partial_fit(X[6:12,:])
>>> kmeans.cluster_centers_
array([[1, 1],
       [3, 4]])
>>> kmeans.predict([[0, 0], [4, 4]])
array([0, 1], dtype=int32)
>>> # fit on the whole data
>>> kmeans = MiniBatchKMeans(n_clusters=2,
...                          random_state=0,
...                          batch_size=6,
...                          max_iter=10).fit(X)
>>> kmeans.cluster_centers_
array([[3.95918367, 2.40816327],
       [1.12195122, 1.3902439 ]])
>>> kmeans.predict([[0, 0], [4, 4]])
array([1, 0], dtype=int32)
```

Methods

fit(self, X[, y, sample_weight])	Compute the centroids on X by chunking it into mini-batches.
fit_predict(self, X[, y, sample_weight])	Compute cluster centers and predict cluster index for each sample.
fit_transform(self, X[, y, sample_weight])	Compute clustering and transform X to cluster-distance space.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X[, y, sample_weight])	Update k means estimate on a single mini-batch X.
predict(self, X[, sample_weight])	Predict the closest cluster each sample in X belongs to.
score(self, X[, y, sample_weight])	Opposite of the value of X on the K-means objective.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X to a cluster-distance space.

__init__(*self*, *n_clusters=8*, *init='k-means++'*, *max_iter=100*, *batch_size=100*, *verbose=0*, *compute_labels=True*, *random_state=None*, *tol=0.0*, *max_no_improvement=10*, *init_size=None*, *n_init=3*, *reassignment_ratio=0.01*)

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

Compute the centroids on X by chunking it into mini-batches.

Parameters

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

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

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

Compute cluster centers and predict cluster index for each sample.

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

Parameters

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

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

Returns

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

fit_transform(*self*, *X*, *y=None*, *sample_weight=None*)

Compute clustering and transform X to cluster-distance space.

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

Parameters

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

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

Returns

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

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X*, *y=None*, *sample_weight=None*)

Update k means estimate on a single mini-batch X.

Parameters

X [array-like, shape = [n_samples, n_features]] Coordinates of the data points to cluster. It must be noted that X will be copied if it is not C-contiguous.

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

predict (*self*, *X*, *sample_weight=None*)

Predict the closest cluster each sample in X belongs to.

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

Parameters

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

Returns

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

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

Opposite of the value of X on the K-means objective.

Parameters

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

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

Returns

score [float] Opposite of the value of X on the K-means objective.

set_params (*self*, ***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.

Returns

self

transform (*self*, *X*)

Transform X to a cluster-distance space.

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

Parameters

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

Returns

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

Examples using `sklearn.cluster.MiniBatchKMeans`

- *Biclustering documents with the Spectral Co-clustering algorithm*
- *Online learning of a dictionary of parts of faces*
- *Compare BIRCH and MiniBatchKMeans*
- *Empirical evaluation of the impact of k-means initialization*
- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*
- *Comparing different clustering algorithms on toy datasets*
- *Faces dataset decompositions*
- *Clustering text documents using k-means*

`sklearn.cluster`.MeanShift

class `sklearn.cluster.`**MeanShift**(*bandwidth=None*, *seeds=None*, *bin_seeding=False*, *min_bin_freq=1*, *cluster_all=True*, *n_jobs=None*)

Mean shift clustering using a flat kernel.

Mean shift clustering aims to discover "blobs" in a smooth density of samples. It is a centroid-based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.

Seeding is performed using a binning technique for scalability.

Read more in the *User Guide*.

Parameters

bandwidth [float, optional] Bandwidth used in the RBF kernel.

If not given, the bandwidth is estimated using sklearn.cluster.estimate_bandwidth; see the documentation for that function for hints on scalability (see also the Notes, below).

seeds [array, shape=[n_samples, n_features], optional] Seeds used to initialize kernels. If not set, the seeds are calculated by clustering.get_bin_seeds with bandwidth as the grid size and default values for other parameters.

bin_seeding [boolean, optional] If true, initial kernel locations are not locations of all points, but rather the location of the discretized version of points, where points are binned onto a grid whose coarseness corresponds to the bandwidth. Setting this option to True will speed up the algorithm because fewer seeds will be initialized. default value: False Ignored if seeds argument is not None.

min_bin_freq [int, optional] To speed up the algorithm, accept only those bins with at least min_bin_freq points as seeds. If not defined, set to 1.

cluster_all [boolean, default True] If true, then all points are clustered, even those orphans that are not within any kernel. Orphans are assigned to the nearest kernel. If false, then orphans are given cluster label -1.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

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

labels_ : Labels of each point.

Notes

Scalability:

Because this implementation uses a flat kernel and a Ball Tree to look up members of each kernel, the complexity will tend towards O(T*n*log(n)) in lower dimensions, with n the number of samples and T the number of points. In higher dimensions the complexity will tend towards O(T*n^2).

Scalability can be boosted by using fewer seeds, for example by using a higher value of min_bin_freq in the get_bin_seeds function.

Note that the estimate_bandwidth function is much less scalable than the mean shift algorithm and will be the bottleneck if it is used.

References

Dorin Comaniciu and Peter Meer, "Mean Shift: A robust approach toward feature space analysis". IEEE Transactions on Pattern Analysis and Machine Intelligence. 2002. pp. 603-619.

Examples

```
>>> from sklearn.cluster import MeanShift
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [1, 0],
...               [4, 7], [3, 5], [3, 6]])
>>> clustering = MeanShift(bandwidth=2).fit(X)
>>> clustering.labels_
array([1, 1, 1, 0, 0, 0])
>>> clustering.predict([[0, 0], [5, 5]])
array([1, 0])
>>> clustering
MeanShift(bandwidth=2, bin_seeding=False, cluster_all=True, min_bin_freq=1,
     n_jobs=None, seeds=None)
```

Methods

fit(self, X[, y])	Perform clustering.
fit_predict(self, X[, y])	Performs clustering on X and returns cluster labels.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the closest cluster each sample in X belongs to.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *bandwidth=None*, *seeds=None*, *bin_seeding=False*, *min_bin_freq=1*, *cluster_all=True*, *n_jobs=None*)

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

Perform clustering.

Parameters

X [array-like, shape=[n_samples, n_features]] Samples to cluster.

y [Ignored]

fit_predict(*self*, *X*, *y=None*)
Performs clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

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

Returns

labels [ndarray, shape (n_samples,)] cluster labels

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)
Predict the closest cluster each sample in X belongs to.

Parameters

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

Returns

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

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.cluster.MeanShift`

- *A demo of the mean-shift clustering algorithm*
- *Comparing different clustering algorithms on toy datasets*

sklearn.cluster.SpectralClustering

class sklearn.cluster.**SpectralClustering**(*n_clusters=8*, *eigen_solver=None*, *random_state=None*, *n_init=10*, *gamma=1.0*, *affinity='rbf'*, *n_neighbors=10*, *eigen_tol=0.0*, *assign_labels='kmeans'*, *degree=3*, *coef0=1*, *kernel_params=None*, *n_jobs=None*)

Apply clustering to a projection of the normalized Laplacian.

In practice Spectral Clustering is very useful when the structure of the individual clusters is highly non-convex or more generally when a measure of the center and spread of the cluster is not a suitable description of the complete cluster. For instance when clusters are nested circles on the 2D plane.

If affinity is the adjacency matrix of a graph, this method can be used to find normalized graph cuts.

When calling `fit`, an affinity matrix is constructed using either kernel function such the Gaussian (aka RBF) kernel of the euclidean distanced `d(X, X)`:

```
np.exp(-gamma * d(X,X) ** 2)
```

or a k-nearest neighbors connectivity matrix.

Alternatively, using `precomputed`, a user-provided affinity matrix can be used.

Read more in the *User Guide*.

Parameters

n_clusters [integer, optional] The dimension of the projection subspace.

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

random_state [int, RandomState instance or None (default)] A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when `eigen_solver='amg'` and by the K-Means initialization. Use an int to make the randomness deterministic. See *Glossary*.

n_init [int, optional, default: 10] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

gamma [float, default=1.0] Kernel coefficient for rbf, poly, sigmoid, laplacian and chi2 kernels. Ignored for `affinity='nearest_neighbors'`.

affinity [string, array-like or callable, default 'rbf'] If a string, this may be one of 'nearest_neighbors', 'precomputed', 'rbf' or one of the kernels supported by `sklearn.metrics.pairwise_kernels`.

Only kernels that produce similarity scores (non-negative values that increase with similarity) should be used. This property is not checked by the clustering algorithm.

n_neighbors [integer] Number of neighbors to use when constructing the affinity matrix using the nearest neighbors method. Ignored for `affinity='rbf'`.

eigen_tol [float, optional, default: 0.0] Stopping criterion for eigendecomposition of the Laplacian matrix when `eigen_solver='arpack'`.

assign_labels [{'kmeans', 'discretize'}, default: 'kmeans'] The strategy to use to assign labels in the embedding space. There are two ways to assign labels after the laplacian embedding.

k-means can be applied and is a popular choice. But it can also be sensitive to initialization. Discretization is another approach which is less sensitive to random initialization.

degree [float, default=3] Degree of the polynomial kernel. Ignored by other kernels.

coef0 [float, default=1] Zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels.

kernel_params [dictionary of string to any, optional] Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

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

Attributes

affinity_matrix_ [array-like, shape (n_samples, n_samples)] Affinity matrix used for clustering. Available only if after calling `fit`.

labels_ : Labels of each point

Notes

If you have an affinity matrix, such as a distance matrix, for which 0 means identical elements, and high values means very dissimilar elements, it can be transformed in a similarity matrix that is well suited for the algorithm by applying the Gaussian (RBF, heat) kernel:

```
np.exp(- dist_matrix ** 2 / (2. * delta ** 2))
```

Where `delta` is a free parameter representing the width of the Gaussian kernel.

Another alternative is to take a symmetric version of the k nearest neighbors connectivity matrix of the points.

If the pyamg package is installed, it is used: this greatly speeds up computation.

References

- Normalized cuts and image segmentation, 2000 Jianbo Shi, Jitendra Malik http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.160.2324
- A Tutorial on Spectral Clustering, 2007 Ulrike von Luxburg http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.9323
- Multiclass spectral clustering, 2003 Stella X. Yu, Jianbo Shi https://www1.icsi.berkeley.edu/~stellayu/publication/doc/2003kwayICCV.pdf

Examples

```
>>> from sklearn.cluster import SpectralClustering
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [1, 0],
...               [4, 7], [3, 5], [3, 6]])
>>> clustering = SpectralClustering(n_clusters=2,
...         assign_labels="discretize",
...         random_state=0).fit(X)
>>> clustering.labels_
```

```
array([1, 1, 1, 0, 0, 0])
>>> clustering
SpectralClustering(affinity='rbf', assign_labels='discretize', coef0=1,
          degree=3, eigen_solver=None, eigen_tol=0.0, gamma=1.0,
          kernel_params=None, n_clusters=2, n_init=10, n_jobs=None,
          n_neighbors=10, random_state=0)
```

Methods

fit(self, X[, y])	Creates an affinity matrix for X using the selected affinity, then applies spectral clustering to this affinity matrix.
fit_predict(self, X[, y])	Performs clustering on X and returns cluster labels.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *n_clusters=8*, *eigen_solver=None*, *random_state=None*, *n_init=10*, *gamma=1.0*, *affinity='rbf'*, *n_neighbors=10*, *eigen_tol=0.0*, *assign_labels='kmeans'*, *degree=3*, *coef0=1*, *kernel_params=None*, *n_jobs=None*)

fit (*self*, *X*, *y=None*)
Creates an affinity matrix for X using the selected affinity, then applies spectral clustering to this affinity matrix.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] OR, if affinity==‘precomputed‘, a precomputed affinity matrix of shape (n_samples, n_samples)

y [Ignored]

fit_predict (*self*, *X*, *y=None*)
Performs clustering on X and returns cluster labels.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

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

Returns

labels [ndarray, shape (n_samples,)] cluster labels

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, ***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.

Returns

self

Examples using `sklearn.cluster.SpectralClustering`

- *Comparing different clustering algorithms on toy datasets*

6.3.2 Functions

cluster.affinity_propagation(S[, ...])	Perform Affinity Propagation Clustering of data
cluster.cluster_optics_dbscan(reachability, ...)	Performs DBSCAN extraction for an arbitrary epsilon.
cluster.cluster_optics_xi(reachability, ...)	Automatically extract clusters according to the Xi-steep method.
cluster.compute_optics_graph(X, min_samples, ...)	Computes the OPTICS reachability graph.
cluster.dbscan(X[, eps, min_samples, ...])	Perform DBSCAN clustering from vector array or distance matrix.
cluster.estimate_bandwidth(X[, quantile, ...])	Estimate the bandwidth to use with the mean-shift algorithm.
cluster.k_means(X, n_clusters[, ...])	K-means clustering algorithm.
cluster.mean_shift(X[, bandwidth, seeds, ...])	Perform mean shift clustering of data using a flat kernel.
cluster.spectral_clustering(affinity[, ...])	Apply clustering to a projection of the normalized Laplacian.
cluster.ward_tree(X[, connectivity, ...])	Ward clustering based on a Feature matrix.

sklearn.cluster.affinity_propagation

sklearn.cluster.**affinity_propagation**(*S*, *preference=None*, *convergence_iter=15*, *max_iter=200*, *damping=0.5*, *copy=True*, *verbose=False*, *return_n_iter=False*)

Perform Affinity Propagation Clustering of data

Read more in the *User Guide*.

Parameters

S [array-like, shape (n_samples, n_samples)] Matrix of similarities between points

preference [array-like, shape (n_samples,) or float, optional] Preferences for each point - points with larger values of preferences are more likely to be chosen as exemplars. The number of exemplars, i.e. of clusters, is influenced by the input preferences value. If the preferences are not passed as arguments, they will be set to the median of the input similarities (resulting in a moderate number of clusters). For a smaller amount of clusters, this can be set to the minimum value of the similarities.

convergence_iter [int, optional, default: 15] Number of iterations with no change in the number of estimated clusters that stops the convergence.

max_iter [int, optional, default: 200] Maximum number of iterations

damping [float, optional, default: 0.5] Damping factor between 0.5 and 1.

copy [boolean, optional, default: True] If copy is False, the affinity matrix is modified inplace by the algorithm, for memory efficiency

verbose [boolean, optional, default: False] The verbosity level

return_n_iter [bool, default False] Whether or not to return the number of iterations.

Returns

cluster_centers_indices [array, shape (n_clusters,)] index of clusters centers

labels [array, shape (n_samples,)] cluster labels for each point

n_iter [int] number of iterations run. Returned only if `return_n_iter` is set to True.

Notes

For an example, see *examples/cluster/plot_affinity_propagation.py*.

When the algorithm does not converge, it returns an empty array as `cluster_center_indices` and `-1` as label for each training sample.

When all training samples have equal similarities and equal preferences, the assignment of cluster centers and labels depends on the preference. If the preference is smaller than the similarities, a single cluster center and label `0` for every sample will be returned. Otherwise, every training sample becomes its own cluster center and is assigned a unique label.

References

Brendan J. Frey and Delbert Dueck, "Clustering by Passing Messages Between Data Points", Science Feb. 2007

Examples using `sklearn.cluster.affinity_propagation`

- *Visualizing the stock market structure*

`sklearn.cluster`.cluster_optics_dbscan

`sklearn.cluster.`**`cluster_optics_dbscan`**(*reachability*, *core_distances*, *ordering*, *eps*)

Performs DBSCAN extraction for an arbitrary epsilon.

Extracting the clusters runs in linear time. Note that this results in `labels_` which are close to a *DBSCAN* with similar settings and `eps`, only if `eps` is close to `max_eps`.

Parameters

reachability [array, shape (n_samples,)] Reachability distances calculated by OPTICS (`reachability_`)

core_distances [array, shape (n_samples,)] Distances at which points become core (`core_distances_`)

ordering [array, shape (n_samples,)] OPTICS ordered point indices (`ordering_`)

eps [float] DBSCAN `eps` parameter. Must be set to < `max_eps`. Results will be close to DBSCAN algorithm if `eps` and `max_eps` are close to one another.

Returns

labels_ [array, shape (n_samples,)] The estimated labels.

Examples using `sklearn.cluster.cluster_optics_dbscan`

- *Demo of OPTICS clustering algorithm*

`sklearn.cluster.cluster_optics_xi`

`sklearn.cluster.cluster_optics_xi(reachability, predecessor, ordering, min_samples, min_cluster_size=None, xi=0.05, predecessor_correction=True)`

Automatically extract clusters according to the Xi-steep method.

Parameters

reachability [array, shape (n_samples,)] Reachability distances calculated by OPTICS (`reachability_`)

predecessor [array, shape (n_samples,)] Predecessors calculated by OPTICS.

ordering [array, shape (n_samples,)] OPTICS ordered point indices (`ordering_`)

min_samples [int > 1 or float between 0 and 1] The same as the min_samples given to OPTICS. Up and down steep regions can't have more then `min_samples` consecutive non-steep points. Expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2).

min_cluster_size [int > 1 or float between 0 and 1 (default=None)] Minimum number of samples in an OPTICS cluster, expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2). If `None`, the value of `min_samples` is used instead.

xi [float, between 0 and 1, optional (default=0.05)] Determines the minimum steepness on the reachability plot that constitutes a cluster boundary. For example, an upwards point in the reachability plot is defined by the ratio from one point to its successor being at most 1-xi.

predecessor_correction [bool, optional (default=True)] Correct clusters based on the calculated predecessors.

Returns

labels [array, shape (n_samples)] The labels assigned to samples. Points which are not included in any cluster are labeled as -1.

clusters [array, shape (n_clusters, 2)] The list of clusters in the form of `[start, end]` in each row, with all indices inclusive. The clusters are ordered according to `(end, -start)` (ascending) so that larger clusters encompassing smaller clusters come after such nested smaller clusters. Since `labels` does not reflect the hierarchy, usually `len(clusters) > np.unique(labels)`.

`sklearn.cluster.compute_optics_graph`

`sklearn.cluster.compute_optics_graph(X, min_samples, max_eps, metric, p, metric_params, algorithm, leaf_size, n_jobs)`

Computes the OPTICS reachability graph.

Read more in the *User Guide*.

Parameters

X [array, shape (n_samples, n_features), or (n_samples, n_samples) if metric='precomputed'.] A feature array, or array of distances between samples if metric='precomputed'

min_samples [int > 1 or float between 0 and 1] The number of samples in a neighborhood for a point to be considered as a core point. Expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2).

max_eps [float, optional (default=np.inf)] The maximum distance between two samples for one to be considered as in the neighborhood of the other. Default value of `np.inf` will identify clusters across all scales; reducing `max_eps` will result in shorter run times.

metric [string or callable, optional (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. If metric is "precomputed", X is assumed to be a distance matrix and must be square.

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_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.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] 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. (default)

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

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.

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

ordering_ [array, shape (n_samples,)] The cluster ordered list of sample indices.

core_distances_ [array, shape (n_samples,)] Distance at which each sample becomes a core point, indexed by object order. Points which will never be core have a distance of inf. Use `clust.core_distances_[clust.ordering_]` to access in cluster order.

reachability_ [array, shape (n_samples,)] Reachability distances per sample, indexed by object order. Use `clust.reachability_[clust.ordering_]` to access in cluster order.

predecessor_ [array, shape (n_samples,)] Point that a sample was reached from, indexed by object order. Seed points have a predecessor of -1.

References

[1]

sklearn.cluster.dbscan

sklearn.cluster.**dbscan**(*X*, *eps=0.5*, *min_samples=5*, *metric='minkowski'*, *metric_params=None*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *sample_weight=None*, *n_jobs=None*)

Perform DBSCAN clustering from vector array or distance matrix.

Read more in the *User Guide*.

Parameters

X [array or sparse (CSR) matrix of shape (n_samples, n_features), or array of shape (n_samples, n_samples)] A feature array, or array of distances between samples if `metric='precomputed'`.

eps [float, optional] The maximum distance between two samples for one to be considered as in the neighborhood of the other. This is not a maximum bound on the distances of points within a cluster. This is the most important DBSCAN parameter to choose appropriately for your data set and distance function.

min_samples [int, optional] The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself.

metric [string, 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 *sklearn.metrics.pairwise_distances* for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square. X may be a sparse matrix, in which case only "nonzero" elements may be considered neighbors for DBSCAN.

metric_params [dict, optional] Additional keyword arguments for the metric function.

New in version 0.19.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] The algorithm to be used by the NearestNeighbors module to compute pointwise distances and find nearest neighbors. See NearestNeighbors module documentation for details.

leaf_size [int, optional (default = 30)] Leaf size passed to BallTree or cKDTree. 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, optional] The power of the Minkowski metric to be used to calculate distance between points.

sample_weight [array, shape (n_samples,), optional] Weight of each sample, such that a sample with a weight of at least `min_samples` is by itself a core sample; a sample with negative weight may inhibit its eps-neighbor from being core. Note that weights are absolute, and default to 1.

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

core_samples [array [n_core_samples]] Indices of core samples.

labels [array [n_samples]] Cluster labels for each point. Noisy samples are given the label -1.

See also:

DBSCAN An estimator interface for this clustering algorithm.

OPTICS A similar estimator interface clustering at multiple values of eps. Our implementation is optimized for memory usage.

Notes

For an example, see *examples/cluster/plot_dbscan.py*.

This implementation bulk-computes all neighborhood queries, which increases the memory complexity to O(n.d) where d is the average number of neighbors, while original DBSCAN had memory complexity O(n). It may attract a higher memory complexity when querying these nearest neighborhoods, depending on the `algorithm`.

One way to avoid the query complexity is to pre-compute sparse neighborhoods in chunks using *NearestNeighbors.radius_neighbors_graph* with `mode='distance'`, then using `metric='precomputed'` here.

Another way to reduce memory and computation time is to remove (near-)duplicate points and use `sample_weight` instead.

`cluster.optics` provides a similar clustering with lower memory usage.

References

Ester, M., H. P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226-231. 1996

Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017). DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. ACM Transactions on Database Systems (TODS), 42(3), 19.

sklearn.cluster.estimate_bandwidth

sklearn.cluster.**estimate_bandwidth**(*X*, *quantile=0.3*, *n_samples=None*, *random_state=0*, *n_jobs=None*)

Estimate the bandwidth to use with the mean-shift algorithm.

That this function takes time at least quadratic in n_samples. For large datasets, it's wise to set that parameter to a small value.

Parameters

X [array-like, shape=[n_samples, n_features]] Input points.

quantile [float, default 0.3] should be between [0, 1] 0.5 means that the median of all pairwise distances is used.

n_samples [int, optional] The number of samples to use. If not given, all samples are used.

random_state [int, RandomState instance or None (default)] The generator used to randomly select the samples from input points for bandwidth estimation. Use an int to make the randomness deterministic. See *Glossary*.

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

bandwidth [float] The bandwidth parameter.

Examples using `sklearn.cluster.estimate_bandwidth`

- *A demo of the mean-shift clustering algorithm*
- *Comparing different clustering algorithms on toy datasets*

`sklearn.cluster.k_means`

`sklearn.cluster.k_means(X, n_clusters, sample_weight=None, init='k-means++', precompute_distances='auto', n_init=10, max_iter=300, verbose=False, tol=0.0001, random_state=None, copy_x=True, n_jobs=None, algorithm='auto', return_n_iter=False)`

K-means clustering algorithm.

Read more in the *User Guide*.

Parameters

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

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

sample_weight [array-like, shape (n_samples,), optional] The weights for each observation in X. If None, all observations are assigned equal weight (default: None)

init [{'k-means++', 'random', or ndarray, or a callable}, optional] Method for initialization, default to 'k-means++':

'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

'random': choose k observations (rows) at random from data for the initial centroids.

If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

If a callable is passed, it should take arguments X, k and and a random state and return an initialization.

precompute_distances [{'auto', True, False}] Precompute distances (faster but takes more memory).

'auto' : do not precompute distances if n_samples * n_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.

True : always precompute distances

False : never precompute distances

n_init [int, optional, default: 10] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

max_iter [int, optional, default 300] Maximum number of iterations of the k-means algorithm to run.

verbose [boolean, optional] Verbosity mode.

tol [float, optional] The relative increment in the results before declaring convergence.

random_state [int, RandomState instance or None (default)] Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. See *Glossary*.

copy_x [boolean, optional] When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True (default), then the original data is not modified, ensuring X is C-contiguous. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean, in this case it will also not ensure that data is C-contiguous which may cause a significant slowdown.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

algorithm ["auto", "full" or "elkan", default="auto"] K-means algorithm to use. The classical EM-style algorithm is "full". The "elkan" variation is more efficient by using the triangle inequality, but currently doesn't support sparse data. "auto" chooses "elkan" for dense data and "full" for sparse data.

return_n_iter [bool, optional] Whether or not to return the number of iterations.

Returns

centroid [float ndarray with shape (k, n_features)] Centroids found at the last iteration of k-means.

label [integer ndarray with shape (n_samples,)] label[i] is the code or index of the centroid the i'th observation is closest to.

inertia [float] The final value of the inertia criterion (sum of squared distances to the closest centroid for all observations in the training set).

best_n_iter [int] Number of iterations corresponding to the best results. Returned only if `return_n_iter` is set to True.

sklearn.cluster.mean_shift

sklearn.cluster.**mean_shift**(*X*, *bandwidth=None*, *seeds=None*, *bin_seeding=False*, *min_bin_freq=1*, *cluster_all=True*, *max_iter=300*, *n_jobs=None*)

Perform mean shift clustering of data using a flat kernel.

Read more in the *User Guide*.

Parameters

X [array-like, shape=[n_samples, n_features]] Input data.

bandwidth [float, optional] Kernel bandwidth.

If bandwidth is not given, it is determined using a heuristic based on the median of all pairwise distances. This will take quadratic time in the number of samples. The sklearn.cluster.estimate_bandwidth function can be used to do this more efficiently.

seeds [array-like, shape=[n_seeds, n_features] or None] Point used as initial kernel locations. If None and bin_seeding=False, each data point is used as a seed. If None and bin_seeding=True, see bin_seeding.

bin_seeding [boolean, default=False] If true, initial kernel locations are not locations of all points, but rather the location of the discretized version of points, where points are binned onto a grid whose coarseness corresponds to the bandwidth. Setting this option to True will speed up the algorithm because fewer seeds will be initialized. Ignored if seeds argument is not None.

min_bin_freq [int, default=1] To speed up the algorithm, accept only those bins with at least min_bin_freq points as seeds.

cluster_all [boolean, default True] If true, then all points are clustered, even those orphans that are not within any kernel. Orphans are assigned to the nearest kernel. If false, then orphans are given cluster label -1.

max_iter [int, default 300] Maximum number of iterations, per seed point before the clustering operation terminates (for that seed point), if has not converged yet.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

New in version 0.17: Parallel Execution using *n_jobs*.

Returns

cluster_centers [array, shape=[n_clusters, n_features]] Coordinates of cluster centers.

labels [array, shape=[n_samples]] Cluster labels for each point.

Notes

For an example, see *examples/cluster/plot_mean_shift.py*.

sklearn.cluster.spectral_clustering

sklearn.cluster.**spectral_clustering**(*affinity*, *n_clusters=8*, *n_components=None*, *eigen_solver=None*, *random_state=None*, *n_init=10*, *eigen_tol=0.0*, *assign_labels='kmeans'*)

Apply clustering to a projection of the normalized Laplacian.

In practice Spectral Clustering is very useful when the structure of the individual clusters is highly non-convex or more generally when a measure of the center and spread of the cluster is not a suitable description of the complete cluster. For instance, when clusters are nested circles on the 2D plane.

If affinity is the adjacency matrix of a graph, this method can be used to find normalized graph cuts.

Read more in the *User Guide*.

Parameters

affinity [array-like or sparse matrix, shape: (n_samples, n_samples)] The affinity matrix describing the relationship of the samples to embed. **Must be symmetric**.

Possible examples:

- adjacency matrix of a graph,
- heat kernel of the pairwise distance matrix of the samples,
- symmetric k-nearest neighbours connectivity matrix of the samples.

n_clusters [integer, optional] Number of clusters to extract.

n_components [integer, optional, default is n_clusters] Number of eigen vectors to use for the spectral embedding

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

random_state [int, RandomState instance or None (default)] A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when eigen_solver == 'amg' and by the K-Means initialization. Use an int to make the randomness deterministic. See *Glossary*.

n_init [int, optional, default: 10] Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

eigen_tol [float, optional, default: 0.0] Stopping criterion for eigendecomposition of the Laplacian matrix when using arpack eigen_solver.

assign_labels [{'kmeans', 'discretize'}, default: 'kmeans'] The strategy to use to assign labels in the embedding space. There are two ways to assign labels after the laplacian embedding. k-means can be applied and is a popular choice. But it can also be sensitive to initialization. Discretization is another approach which is less sensitive to random initialization. See the 'Multiclass spectral clustering' paper referenced below for more details on the discretization approach.

Returns

labels [array of integers, shape: n_samples] The labels of the clusters.

Notes

The graph should contain only one connect component, elsewhere the results make little sense.

This algorithm solves the normalized cut for k=2: it is a normalized spectral clustering.

References

- Normalized cuts and image segmentation, 2000 Jianbo Shi, Jitendra Malik http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.160.2324
- A Tutorial on Spectral Clustering, 2007 Ulrike von Luxburg http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.9323
- Multiclass spectral clustering, 2003 Stella X. Yu, Jianbo Shi https://www1.icsi.berkeley.edu/~stellayu/publication/doc/2003kwayICCV.pdf

Examples using `sklearn.cluster.spectral_clustering`

- *Segmenting the picture of greek coins in regions*
- *Spectral clustering for image segmentation*

`sklearn.cluster.ward_tree`

`sklearn.cluster.`**`ward_tree`**(*X*, *connectivity=None*, *n_clusters=None*, *return_distance=False*)

Ward clustering based on a Feature matrix.

Recursively merges the pair of clusters that minimally increases within-cluster variance.

The inertia matrix uses a Heapq-based representation.

This is the structured version, that takes into account some topological structure between samples.

Read more in the *User Guide*.

Parameters

X [array, shape (n_samples, n_features)] feature matrix representing n_samples samples to be clustered

connectivity [sparse matrix (optional).] connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. The matrix is assumed to be symmetric and only the upper triangular half is used. Default is None, i.e, the Ward algorithm is unstructured.

n_clusters [int (optional)] Stop early the construction of the tree at n_clusters. This is useful to decrease computation time if the number of clusters is not small compared to the number of samples. In this case, the complete tree is not computed, thus the 'children' output is of limited use, and the 'parents' output should rather be used. This option is valid only when specifying a connectivity matrix.

return_distance [bool (optional)] If True, return the distance between the clusters.

Returns

children [2D array, shape (n_nodes-1, 2)] The children of each non-leaf node. Values less than *n_samples* correspond to leaves of the tree which are the original samples. A node `i` greater than or equal to *n_samples* is a non-leaf node and has children `children_[i - n_samples]`. Alternatively at the i-th iteration, children[i][0] and children[i][1] are merged to form node `n_samples + i`

n_connected_components [int] The number of connected components in the graph.

n_leaves [int] The number of leaves in the tree

parents [1D array, shape (n_nodes,) or None] The parent of each node. Only returned when a connectivity matrix is specified, elsewhere 'None' is returned.

distances [1D array, shape (n_nodes-1,)] Only returned if return_distance is set to True (for compatibility). The distances between the centers of the nodes. `distances[i]` corresponds to a weighted euclidean distance between the nodes `children[i, 1]` and `children[i, 2]`. If the nodes refer to leaves of the tree, then `distances[i]` is their unweighted euclidean distance. Distances are updated in the following way (from scipy.hierarchy.linkage):

The new entry $d(u, v)$ is computed as follows,

$$d(u,v) = \sqrt{\frac{|v|+|s|}{T}d(v,s)^2 + \frac{|v|+|t|}{T}d(v,t)^2 - \frac{|v|}{T}d(s,t)^2}$$

where u is the newly joined cluster consisting of clusters s and t, v is an unused cluster in the forest, $T = |v| + |s| + |t|$, and $|*|$ is the cardinality of its argument. This is also known as the incremental algorithm.

6.4 `sklearn.cluster.bicluster`: Biclustering

Spectral biclustering algorithms.

Authors : Kemal Eren License: BSD 3 clause

User guide: See the *Biclustering* section for further details.

6.4.1 Classes

SpectralBiclustering([n_clusters, method, ...])	Spectral biclustering (Kluger, 2003).
SpectralCoclustering([n_clusters, ...])	Spectral Co-Clustering algorithm (Dhillon, 2001).

`sklearn.cluster.bicluster.SpectralBiclustering`

class `sklearn.cluster.bicluster.`**`SpectralBiclustering`**(*n_clusters=3*, *method='bistochastic'*, *n_components=6*, *n_best=3*, *svd_method='randomized'*, *n_svd_vecs=None*, *mini_batch=False*, *init='k-means++'*, *n_init=10*, *n_jobs=None*, *random_state=None*)

Spectral biclustering (Kluger, 2003).

Partitions rows and columns under the assumption that the data has an underlying checkerboard structure. For instance, if there are two row partitions and three column partitions, each row will belong to three biclusters, and each column will belong to two biclusters. The outer product of the corresponding row and column label vectors gives this checkerboard structure.

Read more in the *User Guide*.

Parameters

n_clusters [integer or tuple (n_row_clusters, n_column_clusters)] The number of row and column clusters in the checkerboard structure.

method [string, optional, default: 'bistochastic'] Method of normalizing and converting singular vectors into biclusters. May be one of 'scale', 'bistochastic', or 'log'. The authors recommend using 'log'. If the data is sparse, however, log normalization will not work, which is why the default is 'bistochastic'. CAUTION: if `method='log'`, the data must not be sparse.

n_components [integer, optional, default: 6] Number of singular vectors to check.

n_best [integer, optional, default: 3] Number of best singular vectors to which to project the data for clustering.

svd_method [string, optional, default: 'randomized'] Selects the algorithm for finding singular vectors. May be 'randomized' or 'arpack'. If 'randomized', uses *sklearn.utils.extmath.randomized_svd*, which may be faster for large matrices. If 'arpack', uses `scipy.sparse.linalg.svds`, which is more accurate, but possibly slower in some cases.

n_svd_vecs [int, optional, default: None] Number of vectors to use in calculating the SVD. Corresponds to `ncv` when `svd_method=arpack` and `n_oversamples` when `svd_method` is 'randomized'.

mini_batch [bool, optional, default: False] Whether to use mini-batch k-means, which is faster but may get different results.

init [{'k-means++', 'random' or an ndarray}] Method for initialization of k-means algorithm; defaults to 'k-means++'.

n_init [int, optional, default: 10] Number of random initializations that are tried with the k-means algorithm.

If mini-batch k-means is used, the best initialization is chosen and the algorithm runs once. Otherwise, the algorithm is run for each initialization and the best solution chosen.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None (default)] Used for randomizing the singular value decomposition and the k-means initialization. Use an int to make the randomness deterministic. See *Glossary*.

Attributes

rows_ [array-like, shape (n_row_clusters, n_rows)] Results of the clustering. `rows[i, r]` is True if cluster `i` contains row `r`. Available only after calling `fit`.

columns_ [array-like, shape (n_column_clusters, n_columns)] Results of the clustering, like `rows`.

row_labels_ [array-like, shape (n_rows,)] Row partition labels.

column_labels_ [array-like, shape (n_cols,)] Column partition labels.

References

- Kluger, Yuval, et. al., 2003. Spectral biclustering of microarray data: coclustering genes and conditions.

Examples

```
>>> from sklearn.cluster import SpectralBiclustering
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [1, 0],
...               [4, 7], [3, 5], [3, 6]])
>>> clustering = SpectralBiclustering(n_clusters=2, random_state=0).fit(X)
>>> clustering.row_labels_
array([1, 1, 1, 0, 0, 0], dtype=int32)
>>> clustering.column_labels_
array([0, 1], dtype=int32)
>>> clustering
SpectralBiclustering(init='k-means++', method='bistochastic',
           mini_batch=False, n_best=3, n_clusters=2, n_components=6,
           n_init=10, n_jobs=None, n_svd_vecs=None, random_state=0,
           svd_method='randomized')
```

Methods

fit(self, X[, y])	Creates a biclustering for X.
get_indices(self, i)	Row and column indices of the i'th bicluster.
get_params(self[, deep])	Get parameters for this estimator.
get_shape(self, i)	Shape of the i'th bicluster.
get_submatrix(self, i, data)	Returns the submatrix corresponding to bicluster i.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *n_clusters=3*, *method='bistochastic'*, *n_components=6*, *n_best=3*, *svd_method='randomized'*, *n_svd_vecs=None*, *mini_batch=False*, *init='k-means++'*, *n_init=10*, *n_jobs=None*, *random_state=None*)

biclusters_
Convenient way to get row and column indicators together.

Returns the rows_ and columns_ members.

fit (*self*, *X*, *y=None*)
Creates a biclustering for X.

Parameters

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

y [Ignored]

get_indices (*self*, *i*)
Row and column indices of the i'th bicluster.

Only works if `rows_` and `columns_` attributes exist.

Parameters

i [int] The index of the cluster.

Returns

row_ind [np.array, dtype=np.intp] Indices of rows in the dataset that belong to the bicluster.

col_ind [np.array, dtype=np.intp] Indices of columns in the dataset that belong to the bicluster.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_shape(*self*, *i*)
Shape of the i'th bicluster.

Parameters

i [int] The index of the cluster.

Returns

shape [(int, int)] Number of rows and columns (resp.) in the bicluster.

get_submatrix(*self*, *i*, *data*)
Returns the submatrix corresponding to bicluster `i`.

Parameters

i [int] The index of the cluster.

data [array] The data.

Returns

submatrix [array] The submatrix corresponding to bicluster i.

Notes

Works with sparse matrices. Only works if `rows_` and `columns_` attributes exist.

set_params(*self*, ***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.

Returns

self

sklearn.cluster.bicluster.SpectralCoclustering

class sklearn.cluster.bicluster.**SpectralCoclustering**(*n_clusters=3*, *svd_method='randomized'*, *n_svd_vecs=None*, *mini_batch=False*, *init='k-means++'*, *n_init=10*, *n_jobs=None*, *random_state=None*)

Spectral Co-Clustering algorithm (Dhillon, 2001).

Clusters rows and columns of an array *X* to solve the relaxed normalized cut of the bipartite graph created from *X* as follows: the edge between row vertex `i` and column vertex `j` has weight `X[i, j]`.

The resulting bicluster structure is block-diagonal, since each row and each column belongs to exactly one bicluster.

Supports sparse matrices, as long as they are nonnegative.

Read more in the *User Guide*.

Parameters

n_clusters [integer, optional, default: 3] The number of biclusters to find.

svd_method [string, optional, default: 'randomized'] Selects the algorithm for finding singular vectors. May be 'randomized' or 'arpack'. If 'randomized', use *sklearn.utils.extmath.randomized_svd*, which may be faster for large matrices. If 'arpack', use `scipy.sparse.linalg.svds`, which is more accurate, but possibly slower in some cases.

n_svd_vecs [int, optional, default: None] Number of vectors to use in calculating the SVD. Corresponds to `ncv` when `svd_method=arpack` and `n_oversamples` when `svd_method` is 'randomized'.

mini_batch [bool, optional, default: False] Whether to use mini-batch k-means, which is faster but may get different results.

init [{'k-means++', 'random' or an ndarray}] Method for initialization of k-means algorithm; defaults to 'k-means++'.

n_init [int, optional, default: 10] Number of random initializations that are tried with the k-means algorithm.

If mini-batch k-means is used, the best initialization is chosen and the algorithm runs once. Otherwise, the algorithm is run for each initialization and the best solution chosen.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None (default)] Used for randomizing the singular value decomposition and the k-means initialization. Use an int to make the randomness deterministic. See *Glossary*.

Attributes

rows_ [array-like, shape (n_row_clusters, n_rows)] Results of the clustering. `rows[i, r]` is True if cluster `i` contains row `r`. Available only after calling `fit`.

columns_ [array-like, shape (n_column_clusters, n_columns)] Results of the clustering, like `rows`.

row_labels_ [array-like, shape (n_rows,)] The bicluster label of each row.

column_labels_ [array-like, shape (n_cols,)] The bicluster label of each column.

References

- Dhillon, Inderjit S, 2001. Co-clustering documents and words using bipartite spectral graph partitioning.

Examples

```
>>> from sklearn.cluster import SpectralCoclustering
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [1, 0],
...               [4, 7], [3, 5], [3, 6]])
>>> clustering = SpectralCoclustering(n_clusters=2, random_state=0).fit(X)
>>> clustering.row_labels_
array([0, 1, 1, 0, 0, 0], dtype=int32)
>>> clustering.column_labels_
array([0, 0], dtype=int32)
>>> clustering
SpectralCoclustering(init='k-means++', mini_batch=False, n_clusters=2,
           n_init=10, n_jobs=None, n_svd_vecs=None, random_state=0,
           svd_method='randomized')
```

Methods

fit(self, X[, y])	Creates a biclustering for X.
get_indices(self, i)	Row and column indices of the i'th bicluster.
get_params(self[, deep])	Get parameters for this estimator.
get_shape(self, i)	Shape of the i'th bicluster.
get_submatrix(self, i, data)	Returns the submatrix corresponding to bicluster `i`.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *n_clusters=3*, *svd_method='randomized'*, *n_svd_vecs=None*, *mini_batch=False*, *init='k-means++'*, *n_init=10*, *n_jobs=None*, *random_state=None*)

biclusters_
Convenient way to get row and column indicators together.

Returns the `rows_` and `columns_` members.

fit (*self*, *X*, *y=None*)
Creates a biclustering for X.

Parameters

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

y [Ignored]

get_indices (*self*, *i*)
Row and column indices of the i'th bicluster.

Only works if `rows_` and `columns_` attributes exist.

Parameters

i [int] The index of the cluster.

Returns

row_ind [np.array, dtype=np.intp] Indices of rows in the dataset that belong to the bicluster.

col_ind [np.array, dtype=np.intp] Indices of columns in the dataset that belong to the bicluster.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_shape(*self*, *i*)
Shape of the i'th bicluster.

Parameters

i [int] The index of the cluster.

Returns

shape [(int, int)] Number of rows and columns (resp.) in the bicluster.

get_submatrix(*self*, *i*, *data*)
Returns the submatrix corresponding to bicluster `i`.

Parameters

i [int] The index of the cluster.

data [array] The data.

Returns

submatrix [array] The submatrix corresponding to bicluster i.

Notes

Works with sparse matrices. Only works if `rows_` and `columns_` attributes exist.

set_params(*self*, ***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.

Returns

self

6.5 sklearn.compose: Composite Estimators

Meta-estimators for building composite models with transformers

In addition to its current contents, this module will eventually be home to refurbished versions of Pipeline and FeatureUnion.

User guide: See the *Pipelines and composite estimators* section for further details.

compose.ColumnTransformer(transformers[, ...])	Applies transformers to columns of an array or pandas DataFrame.
compose.TransformedTargetRegressor([...])	Meta-estimator to regress on a transformed target.

6.5.1 sklearn.compose.ColumnTransformer

class sklearn.compose.**ColumnTransformer**(*transformers*, *remainder='drop'*, *sparse_threshold=0.3*, *n_jobs=None*, *transformer_weights=None*, *verbose=False*)

Applies transformers to columns of an array or pandas DataFrame.

This estimator allows different columns or column subsets of the input to be transformed separately and the features generated by each transformer will be concatenated to form a single feature space. This is useful for heterogeneous or columnar data, to combine several feature extraction mechanisms or transformations into a single transformer.

Read more in the *User Guide*.

New in version 0.20.

Parameters

transformers [list of tuples] List of (name, transformer, column(s)) tuples specifying the transformer objects to be applied to subsets of the data.

name [string] Like in Pipeline and FeatureUnion, this allows the transformer and its parameters to be set using `set_params` and searched in grid search.

transformer [estimator or {'passthrough', 'drop'}] Estimator must support *fit* and *transform*. Special-cased strings 'drop' and 'passthrough' are accepted as well, to indicate to drop the columns or to pass them through untransformed, respectively.

column(s) [string or int, array-like of string or int, slice, boolean mask array or callable] Indexes the data on its second axis. Integers are interpreted as positional columns, while strings can reference DataFrame columns by name. A scalar string or int should be used where `transformer` expects X to be a 1d array-like (vector), otherwise a 2d array will be passed to the transformer. A callable is passed the input data *X* and can return any of the above.

remainder [{'drop', 'passthrough'} or estimator, default 'drop'] By default, only the specified columns in *transformers* are transformed and combined in the output, and the non-specified columns are dropped. (default of `'drop'`). By specifying `remainder='passthrough'`, all remaining columns that were not specified in *transformers* will be automatically passed through. This subset of columns is concatenated with the output of the transformers. By setting `remainder` to be an estimator, the remaining non-specified columns will use the `remainder` estimator. The estimator must support *fit* and *transform*. Note that using this feature requires that the DataFrame columns input at *fit* and *transform* have identical order.

sparse_threshold [float, default = 0.3] If the output of the different transformers contains sparse matrices, these will be stacked as a sparse matrix if the overall density is lower than this value. Use `sparse_threshold=0` to always return dense. When the transformed output consists of all dense data, the stacked result will be dense, and this keyword will be ignored.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

transformer_weights [dict, optional] Multiplicative weights for features per transformer. The output of the transformer is multiplied by these weights. Keys are transformer names, values the weights.

verbose [boolean, optional(default=False)] If True, the time elapsed while fitting each transformer will be printed as it is completed.

Attributes

transformers_ [list] The collection of fitted transformers as tuples of (name, fitted_transformer, column). `fitted_transformer` can be an estimator, 'drop', or 'passthrough'. In case there were no columns selected, this will be the unfitted transformer. If there are remaining columns, the final element is a tuple of the form: ('remainder', transformer, remaining_columns) corresponding to the `remainder` parameter. If there are remaining columns, then `len(transformers_)==len(transformers)+1`, otherwise `len(transformers_)==len(transformers)`.

named_transformers_ [Bunch object, a dictionary with attribute access] Access the fitted transformer by name.

sparse_output_ [boolean] Boolean flag indicating wether the output of `transform` is a sparse matrix or a dense numpy array, which depends on the output of the individual transformers and the `sparse_threshold` keyword.

See also:

sklearn.compose.make_column_transformer convenience function for combining the outputs of multiple transformer objects applied to column subsets of the original feature space.

Notes

The order of the columns in the transformed feature matrix follows the order of how the columns are specified in the *transformers* list. Columns of the original feature matrix that are not specified are dropped from the resulting transformed feature matrix, unless specified in the `passthrough` keyword. Those columns specified with `passthrough` are added at the right to the output of the transformers.

Examples

```
>>> import numpy as np
>>> from sklearn.compose import ColumnTransformer
>>> from sklearn.preprocessing import Normalizer
>>> ct = ColumnTransformer(
...     [("norm1", Normalizer(norm='l1'), [0, 1]),
...      ("norm2", Normalizer(norm='l1'), slice(2, 4))])
>>> X = np.array([[0., 1., 2., 2.],
...               [1., 1., 0., 1.]])
>>> # Normalizer scales each row of X to unit norm. A separate scaling
```

```
>>> # is applied for the two first and two last elements of each
>>> # row independently.
>>> ct.fit_transform(X)
array([[0. , 1. , 0.5, 0.5],
       [0.5, 0.5, 0. , 1. ]])
```

Methods

fit(self, X[, y])	Fit all transformers using X.
fit_transform(self, X[, y])	Fit all transformers, transform the data and concatenate results.
get_feature_names(self)	Get feature names from all transformers.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **kwargs)	Set the parameters of this estimator.
transform(self, X)	Transform X separately by each transformer, concatenate results.

__init__(*self*, *transformers*, *remainder='drop'*, *sparse_threshold=0.3*, *n_jobs=None*, *transformer_weights=None*, *verbose=False*)

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

Fit all transformers using X.

Parameters

X [array-like or DataFrame of shape [n_samples, n_features]] Input data, of which specified subsets are used to fit the transformers.

y [array-like, shape (n_samples, ...), optional] Targets for supervised learning.

Returns

self [ColumnTransformer] This estimator

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

Fit all transformers, transform the data and concatenate results.

Parameters

X [array-like or DataFrame of shape [n_samples, n_features]] Input data, of which specified subsets are used to fit the transformers.

y [array-like, shape (n_samples, ...), optional] Targets for supervised learning.

Returns

X_t [array-like or sparse matrix, shape (n_samples, sum_n_components)] hstack of results of transformers. sum_n_components is the sum of n_components (output dimension) over transformers. If any result is a sparse matrix, everything will be converted to sparse matrices.

get_feature_names(*self*)

Get feature names from all transformers.

Returns

feature_names [list of strings] Names of the features produced by transform.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

named_transformers_
Access the fitted transformer by name.

Read-only attribute to access any transformer by given name. Keys are transformer names and values are the fitted transformer objects.

set_params(*self*, ***kwargs*)
Set the parameters of this estimator.

Valid parameter keys can be listed with `get_params()`.

Returns

self

transform(*self*, *X*)
Transform X separately by each transformer, concatenate results.

Parameters

X [array-like or DataFrame of shape [n_samples, n_features]] The data to be transformed by subset.

Returns

X_t [array-like or sparse matrix, shape (n_samples, sum_n_components)] hstack of results of transformers. sum_n_components is the sum of n_components (output dimension) over transformers. If any result is a sparse matrix, everything will be converted to sparse matrices.

Examples using `sklearn.compose.ColumnTransformer`

- *Column Transformer with Mixed Types*
- *Column Transformer with Heterogeneous Data Sources*

6.5.2 `sklearn.compose`.TransformedTargetRegressor

class `sklearn.compose.`**TransformedTargetRegressor**(*regressor=None*, *transformer=None*, *func=None*, *inverse_func=None*, *check_inverse=True*)
Meta-estimator to regress on a transformed target.

Useful for applying a non-linear transformation in regression problems. This transformation can be given as a Transformer such as the QuantileTransformer or as a function and its inverse such as `log` and `exp`.

The computation during `fit` is:

```
regressor.fit(X, func(y))
```

or:

```
regressor.fit(X, transformer.transform(y))
```

The computation during `predict` is:

```
inverse_func(regressor.predict(X))
```

or:

```
transformer.inverse_transform(regressor.predict(X))
```

Read more in the *User Guide*.

Parameters

regressor [object, default=LinearRegression()] Regressor object such as derived from `RegressorMixin`. This regressor will automatically be cloned each time prior to fitting.

transformer [object, default=None] Estimator object such as derived from `TransformerMixin`. Cannot be set at the same time as `func` and `inverse_func`. If `transformer` is `None` as well as `func` and `inverse_func`, the transformer will be an identity transformer. Note that the transformer will be cloned during fitting. Also, the transformer is restricting `y` to be a numpy array.

func [function, optional] Function to apply to `y` before passing to `fit`. Cannot be set at the same time as `transformer`. The function needs to return a 2-dimensional array. If `func` is `None`, the function used will be the identity function.

inverse_func [function, optional] Function to apply to the prediction of the regressor. Cannot be set at the same time as `transformer` as well. The function needs to return a 2-dimensional array. The inverse function is used to return predictions to the same space of the original training labels.

check_inverse [bool, default=True] Whether to check that `transform` followed by `inverse_transform` or `func` followed by `inverse_func` leads to the original targets.

Attributes

regressor_ [object] Fitted regressor.

transformer_ [object] Transformer used in `fit` and `predict`.

Notes

Internally, the target `y` is always converted into a 2-dimensional array to be used by scikit-learn transformers. At the time of prediction, the output will be reshaped to a have the same number of dimensions as `y`.

See *examples/compose/plot_transformed_target.py*.

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.compose import TransformedTargetRegressor
>>> tt = TransformedTargetRegressor(regressor=LinearRegression(),
...                                 func=np.log, inverse_func=np.exp)
```

```
>>> X = np.arange(4).reshape(-1, 1)
>>> y = np.exp(2 * X).ravel()
>>> tt.fit(X, y)
TransformedTargetRegressor(...)
>>> tt.score(X, y)
1.0
>>> tt.regressor_.coef_
array([2.])
```

Methods

fit(self, X, y[, sample_weight])	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the base regressor, applying inverse.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *regressor=None*, *transformer=None*, *func=None*, *inverse_func=None*, *check_inverse=True*)

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

Fit the model according to the given training data.

Parameters

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

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

sample_weight [array-like, shape (n_samples,) optional] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

Returns

self [object]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)

Predict using the base regressor, applying inverse.

The regressor is used to predict and the `inverse_func` or `inverse_transform` is applied before returning the prediction.

Parameters

X [{array-like, sparse matrix}, shape = (n_samples, n_features)] Samples.

Returns

y_hat [array, shape = (n_samples,)] Predicted values.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.compose.TransformedTargetRegressor`

- *Effect of transforming the targets in regression model*

compose.make_column_transformer(...)	Construct a ColumnTransformer from the given transformers.

6.5.3 `sklearn.compose`.make_column_transformer

`sklearn.compose.`**make_column_transformer**(**transformers*, ***kwargs*)

Construct a ColumnTransformer from the given transformers.

This is a shorthand for the ColumnTransformer constructor; it does not require, and does not permit, naming the transformers. Instead, they will be given names automatically based on their types. It also does not allow weighting with `transformer_weights`.

Parameters

***transformers** [tuples of transformers and column selections]

remainder [{'drop', 'passthrough'} or estimator, default 'drop'] By default, only the specified columns in *transformers* are transformed and combined in the output, and the non-specified columns are dropped. (default of `'drop'`). By specifying `remainder='passthrough'`, all remaining columns that were not specified in *transformers* will be automatically passed through. This subset of columns is concatenated with the output of the transformers. By setting `remainder` to be an estimator, the remaining non-specified columns will use the `remainder` estimator. The estimator must support *fit* and *transform*.

sparse_threshold [float, default = 0.3] If the transformed output consists of a mix of sparse and dense data, it will be stacked as a sparse matrix if the density is lower than this value. Use `sparse_threshold=0` to always return dense. When the transformed output consists of all sparse or all dense data, the stacked result will be sparse or dense, respectively, and this keyword will be ignored.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [boolean, optional(default=False)] If True, the time elapsed while fitting each transformer will be printed as it is completed.

Returns

ct [ColumnTransformer]

See also:

sklearn.compose.ColumnTransformer Class that allows combining the outputs of multiple transformer objects used on column subsets of the data into a single feature space.

Examples

```
>>> from sklearn.preprocessing import StandardScaler, OneHotEncoder
>>> from sklearn.compose import make_column_transformer
>>> make_column_transformer(
...     (StandardScaler(), ['numerical_column']),
...     (OneHotEncoder(), ['categorical_column']))
...
ColumnTransformer(n_jobs=None, remainder='drop', sparse_threshold=0.3,
         transformer_weights=None,
         transformers=[('standardscaler',
                        StandardScaler(...),
                        ['numerical_column']),
                       ('onehotencoder',
                        OneHotEncoder(...),
                        ['categorical_column'])], verbose=False)
```

6.6 sklearn.covariance: Covariance Estimators

The *sklearn.covariance* module includes methods and algorithms to robustly estimate the covariance of features given a set of points. The precision matrix defined as the inverse of the covariance is also estimated. Covariance estimation is closely related to the theory of Gaussian Graphical Models.

User guide: See the *Covariance estimation* section for further details.

covariance.EmpiricalCovariance([...])	Maximum likelihood covariance estimator
covariance.EllipticEnvelope([...])	An object for detecting outliers in a Gaussian distributed dataset.
covariance.GraphicalLasso([alpha, mode, ...])	Sparse inverse covariance estimation with an l1-penalized estimator.
covariance.GraphicalLassoCV([alphas, ...])	Sparse inverse covariance w/ cross-validated choice of the l1 penalty.
covariance.LedoitWolf([store_precision, ...])	LedoitWolf Estimator
covariance.MinCovDet([store_precision, ...])	Minimum Covariance Determinant (MCD): robust estimator of covariance.
covariance.OAS([store_precision, ...])	Oracle Approximating Shrinkage Estimator
covariance.ShrunkCovariance([...])	Covariance estimator with shrinkage

6.6.1 sklearn.covariance.EmpiricalCovariance

class sklearn.covariance.**EmpiricalCovariance**(*store_precision=True*, *assume_centered=False*)

Maximum likelihood covariance estimator

Read more in the *User Guide*.

Parameters

store_precision [bool] Specifies if the estimated precision is stored.

assume_centered [bool] If True, data are not centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False (default), data are centered before computation.

Attributes

location_ [array-like, shape (n_features,)] Estimated location, i.e. the estimated mean.

covariance_ [2D ndarray, shape (n_features, n_features)] Estimated covariance matrix

precision_ [2D ndarray, shape (n_features, n_features)] Estimated pseudo-inverse matrix. (stored only if store_precision is True)

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import EmpiricalCovariance
>>> from sklearn.datasets import make_gaussian_quantiles
>>> real_cov = np.array([[.8, .3],
...                      [.3, .4]])
>>> rng = np.random.RandomState(0)
>>> X = rng.multivariate_normal(mean=[0, 0],
```

```
...                                 cov=real_cov,
...                                 size=500)
>>> cov = EmpiricalCovariance().fit(X)
>>> cov.covariance_
array([[0.7569..., 0.2818...],
       [0.2818..., 0.3928...]])
>>> cov.location_
array([0.0622..., 0.0193...])
```

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the Maximum Likelihood Estimator covariance model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*)

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like, shape = [n_features, n_features]] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

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

Fits the Maximum Likelihood Estimator covariance model according to the given training data and parameters.

Parameters

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

y not used, present for API consistence purpose.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like, shape = [n_samples, n_features]] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

score(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like, shape = [n_samples, n_features]] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.covariance.EmpiricalCovariance`

- *Robust covariance estimation and Mahalanobis distances relevance*
- *Robust vs Empirical covariance estimate*

6.6.2 `sklearn.covariance`.EllipticEnvelope

class `sklearn.covariance.`**`EllipticEnvelope`**(*store_precision=True*, *assume_centered=False*, *support_fraction=None*, *contamination=0.1*, *random_state=None*)

An object for detecting outliers in a Gaussian distributed dataset.

Read more in the *User Guide*.

Parameters

store_precision [boolean, optional (default=True)] Specify if the estimated precision is stored.

assume_centered [boolean, optional (default=False)] If True, the support of robust location and covariance estimates is computed, and a covariance estimate is recomputed from it, without centering the data. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, the robust location and covariance are directly computed with the FastMCD algorithm without additional treatment.

support_fraction [float in (0., 1.), optional (default=None)] The proportion of points to be included in the support of the raw MCD estimate. If None, the minimum value of support_fraction will be used within the algorithm: `[n_sample + n_features + 1] / 2`.

contamination [float in (0., 0.5), optional (default=0.1)] The amount of contamination of the data set, i.e. the proportion of outliers in the data set.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. 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`.

Attributes

location_ [array-like, shape (n_features,)] Estimated robust location

covariance_ [array-like, shape (n_features, n_features)] Estimated robust covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix. (stored only if store_precision is True)

support_ [array-like, shape (n_samples,)] A mask of the observations that have been used to compute the robust estimates of location and shape.

offset_ [float] Offset used to define the decision function from the raw scores. We have the relation: `decision_function = score_samples - offset_`. The offset depends on the contamination parameter and is defined in such a way we obtain the expected number of outliers (samples with decision function < 0) in training.

See also:

EmpiricalCovariance, *MinCovDet*

Notes

Outlier detection from covariance estimation may break or not perform well in high-dimensional settings. In particular, one will always take care to work with `n_samples > n_features ** 2`.

References

[R68ae096da0e4-1]

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import EllipticEnvelope
>>> true_cov = np.array([[.8, .3],
...                      [.3, .4]])
>>> X = np.random.RandomState(0).multivariate_normal(mean=[0, 0],
...                                                  cov=true_cov,
...                                                  size=500)
>>> cov = EllipticEnvelope(random_state=0).fit(X)
>>> # predict returns 1 for an inlier and -1 for an outlier
>>> cov.predict([[0, 0],
...              [3, 3]])
array([ 1, -1])
>>> cov.covariance_
array([[0.7411..., 0.2535...],
       [0.2535..., 0.3053...]])
>>> cov.location_
array([0.0813... , 0.0427...])
```

Methods

correct_covariance(self, data)	Apply a correction to raw Minimum Covariance Determinant estimates.
decision_function(self, X[, raw_values])	Compute the decision function of the given observations.
error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fit the EllipticEnvelope model.
fit_predict(self, X[, y])	Performs fit on X and returns labels for X.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
predict(self, X)	Predict the labels (1 inlier, -1 outlier) of X according to the fitted model.
reweight_covariance(self, data)	Re-weight raw Minimum Covariance Determinant estimates.

Continued on next page

Table 6.34 – continued from previous page

score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
score_samples(self, X)	Compute the negative Mahalanobis distances.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *store_precision=True*, *assume_centered=False*, *support_fraction=None*, *contamination=0.1*, *random_state=None*)

correct_covariance (*self*, *data*)

Apply a correction to raw Minimum Covariance Determinant estimates.

Correction using the empirical correction factor suggested by Rousseeuw and Van Driessen in *[RVD]*.

Parameters

data [array-like, shape (n_samples, n_features)] The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

Returns

covariance_corrected [array-like, shape (n_features, n_features)] Corrected robust covariance estimate.

References

[RVD]

decision_function (*self*, *X*, *raw_values=None*)

Compute the decision function of the given observations.

Parameters

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

raw_values [bool, optional] Whether or not to consider raw Mahalanobis distances as the decision function. Must be False (default) for compatibility with the others outlier detection tools.

Deprecated since version 0.20: `raw_values` has been deprecated in 0.20 and will be removed in 0.22.

Returns

decision [array-like, shape (n_samples,)] Decision function of the samples. It is equal to the shifted Mahalanobis distances. The threshold for being an outlier is 0, which ensures a compatibility with other outlier detection algorithms.

error_norm (*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like, shape = [n_features, n_features]] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

`fit`(*self*, *X*, *y=None*)
Fit the EllipticEnvelope model.

Parameters

X [numpy array or sparse matrix, shape (n_samples, n_features).] Training data

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

`fit_predict`(*self*, *X*, *y=None*)
Performs fit on X and returns labels for X.

Returns -1 for outliers and 1 for inliers.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

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

Returns

y [ndarray, shape (n_samples,)] 1 for inliers, -1 for outliers.

`get_params`(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

`get_precision`(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

`mahalanobis`(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like, shape = [n_samples, n_features]] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

`predict`(*self*, *X*)
Predict the labels (1 inlier, -1 outlier) of X according to the fitted model.

Parameters

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

Returns

is_inlier [array, shape (n_samples,)] Returns -1 for anomalies/outliers and +1 for inliers.

reweight_covariance(*self*, *data*)
Re-weight raw Minimum Covariance Determinant estimates.

Re-weight observations using Rousseeuw's method (equivalent to deleting outlying observations from the data set before computing location and covariance estimates) described in *[RVDriessen]*.

Parameters

data [array-like, shape (n_samples, n_features)] The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

Returns

location_reweighted [array-like, shape (n_features,)] Re-weighted robust location estimate.

covariance_reweighted [array-like, shape (n_features, n_features)] Re-weighted robust covariance estimate.

support_reweighted [array-like, type boolean, shape (n_samples,)] A mask of the observations that have been used to compute the re-weighted robust location and covariance estimates.

References

[RVDriessen]

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape (n_samples,), optional] Sample weights.

Returns

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

score_samples(*self*, *X*)
Compute the negative Mahalanobis distances.

Parameters

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

Returns

negative_mahal_distances [array-like, shape (n_samples,)] Opposite of the Mahalanobis distances.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.covariance.EllipticEnvelope`

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *Outlier detection on a real data set*

6.6.3 sklearn.covariance.GraphicalLasso

class sklearn.covariance.**GraphicalLasso**(*alpha=0.01*, *mode='cd'*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *verbose=False*, *assume_centered=False*)
Sparse inverse covariance estimation with an l1-penalized estimator.

Read more in the *User Guide*.

Parameters

alpha [positive float, default 0.01] The regularization parameter: the higher alpha, the more regularization, the sparser the inverse covariance.

mode [{'cd', 'lars'}, default 'cd'] The Lasso solver to use: coordinate descent or LARS. Use LARS for very sparse underlying graphs, where p > n. Elsewhere prefer cd which is more numerically stable.

tol [positive float, default 1e-4] The tolerance to declare convergence: if the dual gap goes below this value, iterations are stopped.

enet_tol [positive float, optional] The tolerance for the elastic net solver used to calculate the descent direction. This parameter controls the accuracy of the search direction for a given column update, not of the overall parameter estimate. Only used for mode='cd'.

max_iter [integer, default 100] The maximum number of iterations.

verbose [boolean, default False] If verbose is True, the objective function and dual gap are plotted at each iteration.

assume_centered [boolean, default False] If True, data are not centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False, data are centered before computation.

Attributes

location_ [array-like, shape (n_features,)] Estimated location, i.e. the estimated mean.

covariance_ [array-like, shape (n_features, n_features)] Estimated covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix.

n_iter_ [int] Number of iterations run.

See also:

graphical_lasso, *GraphicalLassoCV*

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import GraphicalLasso
>>> true_cov = np.array([[0.8, 0.0, 0.2, 0.0],
...                      [0.0, 0.4, 0.0, 0.0],
...                      [0.2, 0.0, 0.3, 0.1],
...                      [0.0, 0.0, 0.1, 0.7]])
>>> np.random.seed(0)
>>> X = np.random.multivariate_normal(mean=[0, 0, 0, 0],
...                                   cov=true_cov,
...                                   size=200)
>>> cov = GraphicalLasso().fit(X)
>>> np.around(cov.covariance_, decimals=3)
array([[0.816, 0.049, 0.218, 0.019],
       [0.049, 0.364, 0.017, 0.034],
       [0.218, 0.017, 0.322, 0.093],
       [0.019, 0.034, 0.093, 0.69 ]])
>>> np.around(cov.location_, decimals=3)
array([0.073, 0.04 , 0.038, 0.143])
```

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the GraphicalLasso model to X.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=0.01*, *mode='cd'*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *verbose=False*, *assume_centered=False*)

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like, shape = [n_features, n_features]] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

`fit`(*self*, *X*, *y=None*)
Fits the GraphicalLasso model to X.

Parameters

X [ndarray, shape (n_samples, n_features)] Data from which to compute the covariance estimate

y [(ignored)]

`get_params`(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

`get_precision`(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

`mahalanobis`(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like, shape = [n_samples, n_features]] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

`score`(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like, shape = [n_samples, n_features]] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***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.

Returns

self

6.6.4 sklearn.covariance.GraphicalLassoCV

class sklearn.covariance.**GraphicalLassoCV**(*alphas=4*, *n_refinements=4*, *cv='warn'*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *mode='cd'*, *n_jobs=None*, *verbose=False*, *assume_centered=False*)
Sparse inverse covariance w/ cross-validated choice of the l1 penalty.

See glossary entry for *cross-validation estimator*.

Read more in the *User Guide*.

Parameters

alphas [integer, or list positive float, optional] If an integer is given, it fixes the number of points on the grids of alpha to be used. If a list is given, it gives the grid to be used. See the notes in the class docstring for more details.

n_refinements [strictly positive integer] The number of times the grid is refined. Not used if explicit values of alphas are passed.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

tol [positive float, optional] The tolerance to declare convergence: if the dual gap goes below this value, iterations are stopped.

enet_tol [positive float, optional] The tolerance for the elastic net solver used to calculate the descent direction. This parameter controls the accuracy of the search direction for a given column update, not of the overall parameter estimate. Only used for mode='cd'.

max_iter [integer, optional] Maximum number of iterations.

mode [{'cd', 'lars'}] The Lasso solver to use: coordinate descent or LARS. Use LARS for very sparse underlying graphs, where number of features is greater than number of samples. Elsewhere prefer cd which is more numerically stable.

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

verbose [boolean, optional] If verbose is True, the objective function and duality gap are printed at each iteration.

assume_centered [boolean] If True, data are not centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False, data are centered before computation.

Attributes

location_ [array-like, shape (n_features,)] Estimated location, i.e. the estimated mean.

covariance_ [numpy.ndarray, shape (n_features, n_features)] Estimated covariance matrix.

precision_ [numpy.ndarray, shape (n_features, n_features)] Estimated precision matrix (inverse covariance).

alpha_ [float] Penalization parameter selected.

cv_alphas_ [list of float] All penalization parameters explored.

grid_scores_ [2D numpy.ndarray (n_alphas, n_folds)] Log-likelihood score on left-out data across folds.

n_iter_ [int] Number of iterations run for the optimal alpha.

See also:

graphical_lasso*, *GraphicalLasso

Notes

The search for the optimal penalization parameter (alpha) is done on an iteratively refined grid: first the cross-validated scores on a grid are computed, then a new refined grid is centered around the maximum, and so on.

One of the challenges which is faced here is that the solvers can fail to converge to a well-conditioned estimate. The corresponding values of alpha then come out as missing values, but the optimum may be close to these missing values.

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import GraphicalLassoCV
>>> true_cov = np.array([[0.8, 0.0, 0.2, 0.0],
...                      [0.0, 0.4, 0.0, 0.0],
...                      [0.2, 0.0, 0.3, 0.1],
...                      [0.0, 0.0, 0.1, 0.7]])
>>> np.random.seed(0)
>>> X = np.random.multivariate_normal(mean=[0, 0, 0, 0],
...                                   cov=true_cov,
...                                   size=200)
>>> cov = GraphicalLassoCV(cv=5).fit(X)
```

```
>>> np.around(cov.covariance_, decimals=3)
array([[0.816, 0.051, 0.22 , 0.017],
       [0.051, 0.364, 0.018, 0.036],
       [0.22 , 0.018, 0.322, 0.094],
       [0.017, 0.036, 0.094, 0.69 ]])
>>> np.around(cov.location_, decimals=3)
array([0.073, 0.04 , 0.038, 0.143])
```

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the GraphicalLasso covariance model to X.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alphas=4*, *n_refinements=4*, *cv='warn'*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *mode='cd'*, *n_jobs=None*, *verbose=False*, *assume_centered=False*)

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like, shape = [n_features, n_features]] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

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

Fits the GraphicalLasso covariance model to X.

Parameters

X [ndarray, shape (n_samples, n_features)] Data from which to compute the covariance estimate

y [(ignored)]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like, shape = [n_samples, n_features]] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

score(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like, shape = [n_samples, n_features]] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.covariance.GraphicalLassoCV`

- *Visualizing the stock market structure*

- *Sparse inverse covariance estimation*

6.6.5 sklearn.covariance.LedoitWolf

class sklearn.covariance.**LedoitWolf**(*store_precision=True*, *assume_centered=False*, *block_size=1000*)

LedoitWolf Estimator

Ledoit-Wolf is a particular form of shrinkage, where the shrinkage coefficient is computed using O. Ledoit and M. Wolf's formula as described in "A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices", Ledoit and Wolf, Journal of Multivariate Analysis, Volume 88, Issue 2, February 2004, pages 365-411.

Read more in the *User Guide*.

Parameters

store_precision [bool, default=True] Specify if the estimated precision is stored.

assume_centered [bool, default=False] If True, data will not be centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False (default), data will be centered before computation.

block_size [int, default=1000] Size of the blocks into which the covariance matrix will be split during its Ledoit-Wolf estimation. This is purely a memory optimization and does not affect results.

Attributes

location_ [array-like, shape (n_features,)] Estimated location, i.e. the estimated mean.

covariance_ [array-like, shape (n_features, n_features)] Estimated covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix. (stored only if store_precision is True)

shrinkage_ [float, 0 <= shrinkage <= 1] Coefficient in the convex combination used for the computation of the shrunk estimate.

Notes

The regularised covariance is:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features and shrinkage is given by the Ledoit and Wolf formula (see References)

References

"A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices", Ledoit and Wolf, Journal of Multivariate Analysis, Volume 88, Issue 2, February 2004, pages 365-411.

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import LedoitWolf
>>> real_cov = np.array([[.4, .2],
...                      [.2, .8]])
```

```
>>> np.random.seed(0)
>>> X = np.random.multivariate_normal(mean=[0, 0],
...                                   cov=real_cov,
...                                   size=50)
>>> cov = LedoitWolf().fit(X)
>>> cov.covariance_
array([[0.4406..., 0.1616...],
       [0.1616..., 0.8022...]])
>>> cov.location_
array([ 0.0595... , -0.0075...])
```

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the Ledoit-Wolf shrunk covariance model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*, *block_size=1000*)

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like, shape = [n_features, n_features]] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

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

Fits the Ledoit-Wolf shrunk covariance model according to the given training data and parameters.

Parameters

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

y not used, present for API consistence purpose.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like, shape = [n_samples, n_features]] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

score(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like, shape = [n_samples, n_features]] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.covariance.LedoitWolf`

- *Ledoit-Wolf vs OAS estimation*
- *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*
- *Model selection with Probabilistic PCA and Factor Analysis (FA)*

6.6.6 `sklearn.covariance`.MinCovDet

class `sklearn.covariance.`**`MinCovDet`**(*store_precision=True*, *assume_centered=False*, *support_fraction=None*, *random_state=None*)

Minimum Covariance Determinant (MCD): robust estimator of covariance.

The Minimum Covariance Determinant covariance estimator is to be applied on Gaussian-distributed data, but could still be relevant on data drawn from a unimodal, symmetric distribution. It is not meant to be used with multi-modal data (the algorithm used to fit a MinCovDet object is likely to fail in such a case). One should consider projection pursuit methods to deal with multi-modal datasets.

Read more in the *User Guide*.

Parameters

store_precision [bool] Specify if the estimated precision is stored.

assume_centered [bool] If True, the support of the robust location and the covariance estimates is computed, and a covariance estimate is recomputed from it, without centering the data. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, the robust location and covariance are directly computed with the FastMCD algorithm without additional treatment.

support_fraction [float, 0 < support_fraction < 1] The proportion of points to be included in the support of the raw MCD estimate. Default is None, which implies that the minimum value of support_fraction will be used within the algorithm: [n_sample + n_features + 1] / 2

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Attributes

raw_location_ [array-like, shape (n_features,)] The raw robust estimated location before correction and re-weighting.

raw_covariance_ [array-like, shape (n_features, n_features)] The raw robust estimated covariance before correction and re-weighting.

raw_support_ [array-like, shape (n_samples,)] A mask of the observations that have been used to compute the raw robust estimates of location and shape, before correction and re-weighting.

location_ [array-like, shape (n_features,)] Estimated robust location

covariance_ [array-like, shape (n_features, n_features)] Estimated robust covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix. (stored only if store_precision is True)

support_ [array-like, shape (n_samples,)] A mask of the observations that have been used to compute the robust estimates of location and shape.

dist_ [array-like, shape (n_samples,)] Mahalanobis distances of the training set (on which *fit* is called) observations.

References

[R9f63e655f7bd-Rouseeuw1984], *[R9f63e655f7bd-Rousseeuw]*, *[R9f63e655f7bd-ButlerDavies]*

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import MinCovDet
>>> from sklearn.datasets import make_gaussian_quantiles
>>> real_cov = np.array([[.8, .3],
...                      [.3, .4]])
>>> rng = np.random.RandomState(0)
>>> X = rng.multivariate_normal(mean=[0, 0],
...                                   cov=real_cov,
...                                   size=500)
>>> cov = MinCovDet(random_state=0).fit(X)
>>> cov.covariance_
array([[0.7411..., 0.2535...],
       [0.2535..., 0.3053...]])
>>> cov.location_
array([0.0813... , 0.0427...])
```

Methods

correct_covariance(self, data)	Apply a correction to raw Minimum Covariance Determinant estimates.
error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits a Minimum Covariance Determinant with the FastMCD algorithm.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
reweight_covariance(self, data)	Re-weight raw Minimum Covariance Determinant estimates.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*, *support_fraction=None*, *random_state=None*)

correct_covariance(*self*, *data*)

Apply a correction to raw Minimum Covariance Determinant estimates.

Correction using the empirical correction factor suggested by Rousseeuw and Van Driessen in *[RVD]*.

Parameters

data [array-like, shape (n_samples, n_features)] The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

Returns

covariance_corrected [array-like, shape (n_features, n_features)] Corrected robust covariance estimate.

References

[RVD]

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like, shape = [n_features, n_features]] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

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

Fits a Minimum Covariance Determinant with the FastMCD algorithm.

Parameters

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

y not used, present for API consistence purpose.

Returns

self [object]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like, shape = [n_samples, n_features]] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

reweight_covariance(*self*, *data*)
Re-weight raw Minimum Covariance Determinant estimates.

Re-weight observations using Rousseeuw's method (equivalent to deleting outlying observations from the data set before computing location and covariance estimates) described in *[RVDriessen]*.

Parameters

data [array-like, shape (n_samples, n_features)] The data matrix, with p features and n samples. The data set must be the one which was used to compute the raw estimates.

Returns

location_reweighted [array-like, shape (n_features,)] Re-weighted robust location estimate.

covariance_reweighted [array-like, shape (n_features, n_features)] Re-weighted robust covariance estimate.

support_reweighted [array-like, type boolean, shape (n_samples,)] A mask of the observations that have been used to compute the re-weighted robust location and covariance estimates.

References

[RVDriessen]

score(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like, shape = [n_samples, n_features]] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of

features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.covariance.MinCovDet`

- *Robust covariance estimation and Mahalanobis distances relevance*
- *Robust vs Empirical covariance estimate*

6.6.7 `sklearn.covariance.OAS`

class `sklearn.covariance.`**OAS**(*store_precision=True*, *assume_centered=False*)
Oracle Approximating Shrinkage Estimator

Read more in the *User Guide*.

OAS is a particular form of shrinkage described in "Shrinkage Algorithms for MMSE Covariance Estimation" Chen et al., IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

The formula used here does not correspond to the one given in the article. In the original article, formula (23) states that 2/p is multiplied by Trace(cov*cov) in both the numerator and denominator, but this operation is omitted because for a large p, the value of 2/p is so small that it doesn't affect the value of the estimator.

Parameters

store_precision [bool, default=True] Specify if the estimated precision is stored.

assume_centered [bool, default=False] If True, data will not be centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False (default), data will be centered before computation.

Attributes

covariance_ [array-like, shape (n_features, n_features)] Estimated covariance matrix.

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix. (stored only if store_precision is True)

shrinkage_ [float, 0 <= shrinkage <= 1] coefficient in the convex combination used for the computation of the shrunk estimate.

Notes

The regularised covariance is:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features and shrinkage is given by the OAS formula (see References)

References

"Shrinkage Algorithms for MMSE Covariance Estimation" Chen et al., IEEE Trans. on Sign. Proc., Volume 58, Issue 10, October 2010.

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the Oracle Approximating Shrinkage covariance model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*)

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like, shape = [n_features, n_features]] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

fit (*self*, *X*, *y=None*)
Fits the Oracle Approximating Shrinkage covariance model according to the given training data and parameters.

Parameters

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

y not used, present for API consistence purpose.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_precision (*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis (*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like, shape = [n_samples, n_features]] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

score (*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like, shape = [n_samples, n_features]] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.covariance.OAS`

- *Ledoit-Wolf vs OAS estimation*
- *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*

6.6.8 `sklearn.covariance`.ShrunkCovariance

class `sklearn.covariance.`**ShrunkCovariance**(*store_precision=True*, *assume_centered=False*, *shrinkage=0.1*)
Covariance estimator with shrinkage

Read more in the *User Guide*.

Parameters

store_precision [boolean, default True] Specify if the estimated precision is stored

assume_centered [boolean, default False] If True, data will not be centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False, data will be centered before computation.

shrinkage [float, 0 <= shrinkage <= 1, default 0.1] Coefficient in the convex combination used for the computation of the shrunk estimate.

Attributes

location_ [array-like, shape (n_features,)] Estimated location, i.e. the estimated mean.

covariance_ [array-like, shape (n_features, n_features)] Estimated covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix. (stored only if store_precision is True)

shrinkage [float, 0 <= shrinkage <= 1] Coefficient in the convex combination used for the computation of the shrunk estimate.

Notes

The regularized covariance is given by:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features

Examples

```
>>> import numpy as np
>>> from sklearn.covariance import ShrunkCovariance
>>> from sklearn.datasets import make_gaussian_quantiles
>>> real_cov = np.array([[.8, .3],
...                      [.3, .4]])
>>> rng = np.random.RandomState(0)
>>> X = rng.multivariate_normal(mean=[0, 0],
...                                   cov=real_cov,
...                                   size=500)
>>> cov = ShrunkCovariance().fit(X)
>>> cov.covariance_
array([[0.7387..., 0.2536...],
       [0.2536..., 0.4110...]])
>>> cov.location_
array([0.0622..., 0.0193...])
```

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the shrunk covariance model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *store_precision=True*, *assume_centered=False*, *shrinkage=0.1*)

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)

Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like, shape = [n_features, n_features]] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

self and comp_cov covariance estimators.

fit(*self*, *X*, *y=None*)
Fits the shrunk covariance model according to the given training data and parameters.

Parameters

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

y not used, present for API consistence purpose.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like, shape = [n_samples, n_features]] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

score(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like, shape = [n_samples, n_features]] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.covariance.ShrunkCovariance`

- *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*
- *Model selection with Probabilistic PCA and Factor Analysis (FA)*

covariance.empirical_covariance(X[, ...])	Computes the Maximum likelihood covariance estimator
covariance.graphical_lasso(emp_cov, alpha[, ...])	l1-penalized covariance estimator
covariance.ledoit_wolf(X[, assume_centered, ...])	Estimates the shrunk Ledoit-Wolf covariance matrix.
covariance.oas(X[, assume_centered])	Estimate covariance with the Oracle Approximating Shrinkage algorithm.
covariance.shrunk_covariance(emp_cov[, ...])	Calculates a covariance matrix shrunk on the diagonal

6.6.9 sklearn.covariance.empirical_covariance

sklearn.covariance.**empirical_covariance**(*X*, *assume_centered=False*)
Computes the Maximum likelihood covariance estimator

Parameters

X [ndarray, shape (n_samples, n_features)] Data from which to compute the covariance estimate

assume_centered [boolean] If True, data will not be centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False, data will be centered before computation.

Returns

covariance [2D ndarray, shape (n_features, n_features)] Empirical covariance (Maximum Likelihood Estimator).

Examples using `sklearn.covariance.empirical_covariance`

- *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*

6.6.10 sklearn.covariance.graphical_lasso

sklearn.covariance.**graphical_lasso**(*emp_cov*, *alpha*, *cov_init=None*, *mode='cd'*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *verbose=False*, *return_costs=False*, *eps=2.220446049250313e-16*, *return_n_iter=False*)
l1-penalized covariance estimator

Read more in the *User Guide*.

Parameters

emp_cov [2D ndarray, shape (n_features, n_features)] Empirical covariance from which to compute the covariance estimate.

alpha [positive float] The regularization parameter: the higher alpha, the more regularization, the sparser the inverse covariance.

cov_init [2D array (n_features, n_features), optional] The initial guess for the covariance.

mode [{'cd', 'lars'}] The Lasso solver to use: coordinate descent or LARS. Use LARS for very sparse underlying graphs, where p > n. Elsewhere prefer cd which is more numerically stable.

tol [positive float, optional] The tolerance to declare convergence: if the dual gap goes below this value, iterations are stopped.

enet_tol [positive float, optional] The tolerance for the elastic net solver used to calculate the descent direction. This parameter controls the accuracy of the search direction for a given column update, not of the overall parameter estimate. Only used for mode='cd'.

max_iter [integer, optional] The maximum number of iterations.

verbose [boolean, optional] If verbose is True, the objective function and dual gap are printed at each iteration.

return_costs [boolean, optional] If return_costs is True, the objective function and dual gap at each iteration are returned.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

return_n_iter [bool, optional] Whether or not to return the number of iterations.

Returns

covariance [2D ndarray, shape (n_features, n_features)] The estimated covariance matrix.

precision [2D ndarray, shape (n_features, n_features)] The estimated (sparse) precision matrix.

costs [list of (objective, dual_gap) pairs] The list of values of the objective function and the dual gap at each iteration. Returned only if return_costs is True.

n_iter [int] Number of iterations. Returned only if `return_n_iter` is set to True.

See also:

GraphicalLasso*, *GraphicalLassoCV

Notes

The algorithm employed to solve this problem is the GLasso algorithm, from the Friedman 2008 Biostatistics paper. It is the same algorithm as in the R `glasso` package.

One possible difference with the `glasso` R package is that the diagonal coefficients are not penalized.

6.6.11 sklearn.covariance.ledoit_wolf

sklearn.covariance.**ledoit_wolf**(*X*, *assume_centered=False*, *block_size=1000*)
Estimates the shrunk Ledoit-Wolf covariance matrix.

Read more in the *User Guide*.

Parameters

X [array-like, shape (n_samples, n_features)] Data from which to compute the covariance estimate

assume_centered [boolean, default=False] If True, data will not be centered before computation. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, data will be centered before computation.

block_size [int, default=1000] Size of the blocks into which the covariance matrix will be split. This is purely a memory optimization and does not affect results.

Returns

shrunk_cov [array-like, shape (n_features, n_features)] Shrunk covariance.

shrinkage [float] Coefficient in the convex combination used for the computation of the shrunk estimate.

Notes

The regularized (shrunk) covariance is:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features

Examples using sklearn.covariance.ledoit_wolf

- *Sparse inverse covariance estimation*

6.6.12 sklearn.covariance.oas

sklearn.covariance.**oas**(*X*, *assume_centered=False*)
Estimate covariance with the Oracle Approximating Shrinkage algorithm.

Parameters

X [array-like, shape (n_samples, n_features)] Data from which to compute the covariance estimate.

assume_centered [boolean] If True, data will not be centered before computation. Useful to work with data whose mean is significantly equal to zero but is not exactly zero. If False, data will be centered before computation.

Returns

shrunk_cov [array-like, shape (n_features, n_features)] Shrunk covariance.

shrinkage [float] Coefficient in the convex combination used for the computation of the shrunk estimate.

Notes

The regularised (shrunk) covariance is:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features

The formula we used to implement the OAS is slightly modified compared to the one given in the article. See *OAS* for more details.

6.6.13 sklearn.covariance.shrunk_covariance

sklearn.covariance.**shrunk_covariance**(*emp_cov*, *shrinkage=0.1*)

Calculates a covariance matrix shrunk on the diagonal

Read more in the *User Guide*.

Parameters

emp_cov [array-like, shape (n_features, n_features)] Covariance matrix to be shrunk

shrinkage [float, 0 <= shrinkage <= 1] Coefficient in the convex combination used for the computation of the shrunk estimate.

Returns

shrunk_cov [array-like] Shrunk covariance.

Notes

The regularized (shrunk) covariance is given by:

(1 - shrinkage) * cov + shrinkage * mu * np.identity(n_features)

where mu = trace(cov) / n_features

6.7 sklearn.cross_decomposition: Cross decomposition

User guide: See the *Cross decomposition* section for further details.

cross_decomposition.CCA([n_components, ...])	CCA Canonical Correlation Analysis.
cross_decomposition.PLSCanonical([...])	PLSCanonical implements the 2 blocks canonical PLS of the original Wold algorithm [Tenenhaus 1998] p.204, referred as PLS-C2A in [Wegelin 2000].
cross_decomposition.PLSRegression([...])	PLS regression
cross_decomposition.PLSSVD([n_components, ...])	Partial Least Square SVD

6.7.1 sklearn.cross_decomposition.CCA

class sklearn.cross_decomposition.**CCA**(*n_components=2*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)

CCA Canonical Correlation Analysis.

CCA inherits from PLS with mode="B" and deflation_mode="canonical".

Read more in the *User Guide*.

Parameters

n_components [int, (default 2).] number of components to keep.

scale [boolean, (default True)] whether to scale the data?

max_iter [an integer, (default 500)] the maximum number of iterations of the NIPALS inner loop

tol [non-negative real, default 1e-06.] the tolerance used in the iterative algorithm

copy [boolean] Whether the deflation be done on a copy. Let the default value to True unless you don't care about side effects

Attributes

x_weights_ [array, [p, n_components]] X block weights vectors.

y_weights_ [array, [q, n_components]] Y block weights vectors.

x_loadings_ [array, [p, n_components]] X block loadings vectors.

y_loadings_ [array, [q, n_components]] Y block loadings vectors.

x_scores_ [array, [n_samples, n_components]] X scores.

y_scores_ [array, [n_samples, n_components]] Y scores.

x_rotations_ [array, [p, n_components]] X block to latents rotations.

y_rotations_ [array, [q, n_components]] Y block to latents rotations.

n_iter_ [array-like] Number of iterations of the NIPALS inner loop for each component.

See also:

PLSCanonical

PLSSVD

Notes

For each component k, find the weights u, v that maximizes max corr(Xk u, Yk v), such that `|u| = |v| = 1`

Note that it maximizes only the correlations between the scores.

The residual matrix of X (Xk+1) block is obtained by the deflation on the current X score: x_score.

The residual matrix of Y (Yk+1) block is obtained by deflation on the current Y score.

References

Jacob A. Wegelin. A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case. Technical Report 371, Department of Statistics, University of Washington, Seattle, 2000.

In french but still a reference: Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris: Editions Technic.

Examples

```
>>> from sklearn.cross_decomposition import CCA
>>> X = [[0., 0., 1.], [1.,0.,0.], [2.,2.,2.], [3.,5.,4.]]
>>> Y = [[0.1, -0.2], [0.9, 1.1], [6.2, 5.9], [11.9, 12.3]]
>>> cca = CCA(n_components=1)
>>> cca.fit(X, Y)
...
CCA(copy=True, max_iter=500, n_components=1, scale=True, tol=1e-06)
>>> X_c, Y_c = cca.transform(X, Y)
```

Methods

fit(self, X, Y)	Fit model to data.
fit_transform(self, X[, y])	Learn and apply the dimension reduction on the train data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, copy])	Apply the dimension reduction learned on the train data.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, Y, copy])	Apply the dimension reduction learned on the train data.

__init__ (*self*, *n_components=2*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)

fit (*self*, *X*, *Y*)
Fit model to data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

fit_transform (*self*, *X*, *y=None*)
Learn and apply the dimension reduction on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*, *copy=True*)

Apply the dimension reduction learned on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Notes

This call requires the estimation of a p x q matrix, which may be an issue in high dimensional space.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

transform(*self*, *X*, *Y=None*, *copy=True*)
Apply the dimension reduction learned on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

Examples using `sklearn.cross_decomposition.CCA`

- *Multilabel classification*
- *Compare cross decomposition methods*

6.7.2 `sklearn.cross_decomposition`.PLSCanonical

class `sklearn.cross_decomposition.`**PLSCanonical**(*n_components=2*, *scale=True*, *algorithm='nipals'*, *max_iter=500*, *tol=1e-06*, *copy=True*)
PLSCanonical implements the 2 blocks canonical PLS of the original Wold algorithm [Tenenhaus 1998] p.204, referred as PLS-C2A in [Wegelin 2000].

This class inherits from PLS with mode="A" and deflation_mode="canonical", norm_y_weights=True and algorithm="nipals", but svd should provide similar results up to numerical errors.

Read more in the *User Guide*.

Parameters

n_components [int, (default 2).] Number of components to keep

scale [boolean, (default True)] Option to scale data

algorithm [string, "nipals" or "svd"] The algorithm used to estimate the weights. It will be called n_components times, i.e. once for each iteration of the outer loop.

max_iter [an integer, (default 500)] the maximum number of iterations of the NIPALS inner loop (used only if algorithm="nipals")

tol [non-negative real, default 1e-06] the tolerance used in the iterative algorithm

copy [boolean, default True] Whether the deflation should be done on a copy. Let the default value to True unless you don't care about side effect

Attributes

x_weights_ [array, shape = [p, n_components]] X block weights vectors.

y_weights_ [array, shape = [q, n_components]] Y block weights vectors.

x_loadings_ [array, shape = [p, n_components]] X block loadings vectors.

y_loadings_ [array, shape = [q, n_components]] Y block loadings vectors.

x_scores_ [array, shape = [n_samples, n_components]] X scores.

y_scores_ [array, shape = [n_samples, n_components]] Y scores.

x_rotations_ [array, shape = [p, n_components]] X block to latents rotations.

y_rotations_ [array, shape = [q, n_components]] Y block to latents rotations.

n_iter_ [array-like] Number of iterations of the NIPALS inner loop for each component. Not useful if the algorithm provided is "svd".

See also:

CCA

PLSSVD

Notes

Matrices:

```
T: x_scores_
U: y_scores_
W: x_weights_
C: y_weights_
P: x_loadings_
Q: y_loadings__
```

Are computed such that:

```
X = T P.T + Err and Y = U Q.T + Err
T[:, k] = Xk W[:, k] for k in range(n_components)
U[:, k] = Yk C[:, k] for k in range(n_components)
x_rotations_ = W (P.T W)^(-1)
y_rotations_ = C (Q.T C)^(-1)
```

where Xk and Yk are residual matrices at iteration k.

Slides explaining PLS

For each component k, find weights u, v that optimize:

```
max corr(Xk u, Yk v) * std(Xk u) std(Yk u), such that ``|u| = |v| = 1``
```

Note that it maximizes both the correlations between the scores and the intra-block variances.

The residual matrix of X (Xk+1) block is obtained by the deflation on the current X score: x_score.

The residual matrix of Y (Yk+1) block is obtained by deflation on the current Y score. This performs a canonical symmetric version of the PLS regression. But slightly different than the CCA. This is mostly used for modeling.

This implementation provides the same results that the "plspm" package provided in the R language (R-project), using the function plsca(X, Y). Results are equal or collinear with the function `pls(..., mode = "canonical")` of the "mixOmics" package. The difference relies in the fact that mixOmics implementation does not exactly implement the Wold algorithm since it does not normalize y_weights to one.

References

Jacob A. Wegelin. A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case. Technical Report 371, Department of Statistics, University of Washington, Seattle, 2000.

Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris: Editions Technic.

Examples

```
>>> from sklearn.cross_decomposition import PLSCanonical
>>> X = [[0., 0., 1.], [1.,0.,0.], [2.,2.,2.], [2.,5.,4.]]
>>> Y = [[0.1, -0.2], [0.9, 1.1], [6.2, 5.9], [11.9, 12.3]]
>>> plsca = PLSCanonical(n_components=2)
>>> plsca.fit(X, Y)
...
PLSCanonical(algorithm='nipals', copy=True, max_iter=500, n_components=2,
             scale=True, tol=1e-06)
>>> X_c, Y_c = plsca.transform(X, Y)
```

Methods

fit(self, X, Y)	Fit model to data.
fit_transform(self, X[, y])	Learn and apply the dimension reduction on the train data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, copy])	Apply the dimension reduction learned on the train data.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, Y, copy])	Apply the dimension reduction learned on the train data.

__init__(*self*, *n_components=2*, *scale=True*, *algorithm='nipals'*, *max_iter=500*, *tol=1e-06*, *copy=True*)

fit(*self*, *X*, *Y*)

Fit model to data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

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

Learn and apply the dimension reduction on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*, *copy=True*)
Apply the dimension reduction learned on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Notes

This call requires the estimation of a p x q matrix, which may be an issue in high dimensional space.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*, *Y=None*, *copy=True*)
Apply the dimension reduction learned on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

Examples using `sklearn.cross_decomposition.PLSCanonical`

- *Compare cross decomposition methods*

6.7.3 `sklearn.cross_decomposition`.PLSRegression

class `sklearn.cross_decomposition.`**PLSRegression**(*n_components=2*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)
PLS regression

PLSRegression implements the PLS 2 blocks regression known as PLS2 or PLS1 in case of one dimensional response. This class inherits from _PLS with mode="A", deflation_mode="regression", norm_y_weights=False and algorithm="nipals".

Read more in the *User Guide*.

Parameters

n_components [int, (default 2)] Number of components to keep.

scale [boolean, (default True)] whether to scale the data

max_iter [an integer, (default 500)] the maximum number of iterations of the NIPALS inner loop (used only if algorithm="nipals")

tol [non-negative real] Tolerance used in the iterative algorithm default 1e-06.

copy [boolean, default True] Whether the deflation should be done on a copy. Let the default value to True unless you don't care about side effect

Attributes

x_weights_ [array, [p, n_components]] X block weights vectors.

y_weights_ [array, [q, n_components]] Y block weights vectors.

x_loadings_ [array, [p, n_components]] X block loadings vectors.

y_loadings_ [array, [q, n_components]] Y block loadings vectors.

x_scores_ [array, [n_samples, n_components]] X scores.

y_scores_ [array, [n_samples, n_components]] Y scores.

x_rotations_ [array, [p, n_components]] X block to latents rotations.

y_rotations_ [array, [q, n_components]] Y block to latents rotations.

coef_ [array, [p, q]] The coefficients of the linear model: `Y = X coef_ + Err`

n_iter_ [array-like] Number of iterations of the NIPALS inner loop for each component.

Notes

Matrices:

```
T: x_scores_
U: y_scores_
W: x_weights_
C: y_weights_
P: x_loadings_
Q: y_loadings_
```

Are computed such that:

```
X = T P.T + Err and Y = U Q.T + Err
T[:, k] = Xk W[:, k] for k in range(n_components)
U[:, k] = Yk C[:, k] for k in range(n_components)
x_rotations_ = W (P.T W)^(-1)
y_rotations_ = C (Q.T C)^(-1)
```

where Xk and Yk are residual matrices at iteration k.

Slides explaining PLS

For each component k, find weights u, v that optimizes: `max corr(Xk u, Yk v) * std(Xk u) std(Yk u)`, such that `|u| = 1`

Note that it maximizes both the correlations between the scores and the intra-block variances.

The residual matrix of X (Xk+1) block is obtained by the deflation on the current X score: x_score.

The residual matrix of Y (Yk+1) block is obtained by deflation on the current X score. This performs the PLS regression known as PLS2. This mode is prediction oriented.

This implementation provides the same results that 3 PLS packages provided in the R language (R-project):

- "mixOmics" with function pls(X, Y, mode = "regression")
- "plspm " with function plsreg2(X, Y)
- "pls" with function oscorespls.fit(X, Y)

References

Jacob A. Wegelin. A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case. Technical Report 371, Department of Statistics, University of Washington, Seattle, 2000.

In french but still a reference: Tenenhaus, M. (1998). La regression PLS: theorie et pratique. Paris: Editions Technic.

Examples

```
>>> from sklearn.cross_decomposition import PLSRegression
>>> X = [[0., 0., 1.], [1.,0.,0.], [2.,2.,2.], [2.,5.,4.]]
>>> Y = [[0.1, -0.2], [0.9, 1.1], [6.2, 5.9], [11.9, 12.3]]
>>> pls2 = PLSRegression(n_components=2)
>>> pls2.fit(X, Y)
...
PLSRegression(copy=True, max_iter=500, n_components=2, scale=True,
        tol=1e-06)
>>> Y_pred = pls2.predict(X)
```

Methods

fit(self, X, Y)	Fit model to data.
fit_transform(self, X[, y])	Learn and apply the dimension reduction on the train data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, copy])	Apply the dimension reduction learned on the train data.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, Y, copy])	Apply the dimension reduction learned on the train data.

__init__ (*self*, *n_components=2*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)

fit (*self*, *X*, *Y*)
Fit model to data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

fit_transform (*self*, *X*, *y=None*)
Learn and apply the dimension reduction on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*, *copy=True*)

Apply the dimension reduction learned on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Notes

This call requires the estimation of a p x q matrix, which may be an issue in high dimensional space.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

transform(*self*, *X*, *Y=None*, *copy=True*)
Apply the dimension reduction learned on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

copy [boolean, default True] Whether to copy X and Y, or perform in-place normalization.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

Examples using `sklearn.cross_decomposition.PLSRegression`

- *Compare cross decomposition methods*

6.7.4 `sklearn.cross_decomposition`.PLSSVD

class `sklearn.cross_decomposition.`**PLSSVD**(*n_components=2*, *scale=True*, *copy=True*)
Partial Least Square SVD

Simply perform a svd on the crosscovariance matrix: X'Y There are no iterative deflation here.

Read more in the *User Guide*.

Parameters

n_components [int, default 2] Number of components to keep.

scale [boolean, default True] Whether to scale X and Y.

copy [boolean, default True] Whether to copy X and Y, or perform in-place computations.

Attributes

x_weights_ [array, [p, n_components]] X block weights vectors.

y_weights_ [array, [q, n_components]] Y block weights vectors.

x_scores_ [array, [n_samples, n_components]] X scores.

y_scores_ [array, [n_samples, n_components]] Y scores.

See also:

PLSCanonical

CCA

Examples

```
>>> import numpy as np
>>> from sklearn.cross_decomposition import PLSSVD
>>> X = np.array([[0., 0., 1.],
...     [1.,0.,0.],
...     [2.,2.,2.],
...     [2.,5.,4.]])
>>> Y = np.array([[0.1, -0.2],
...     [0.9, 1.1],
...     [6.2, 5.9],
...     [11.9, 12.3]])
>>> plsca = PLSSVD(n_components=2)
>>> plsca.fit(X, Y)
PLSSVD(copy=True, n_components=2, scale=True)
>>> X_c, Y_c = plsca.transform(X, Y)
>>> X_c.shape, Y_c.shape
((4, 2), (4, 2))
```

Methods

fit(self, X, Y)	Fit model to data.
fit_transform(self, X[, y])	Learn and apply the dimension reduction on the train data.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, Y])	Apply the dimension reduction learned on the train data.

__init__ (*self*, *n_components=2*, *scale=True*, *copy=True*)

fit (*self*, *X*, *Y*)
Fit model to data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

fit_transform (*self*, *X*, *y=None*)
Learn and apply the dimension reduction on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

Returns

x_scores if Y is not given, (x_scores, y_scores) otherwise.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*, *Y=None*)
Apply the dimension reduction learned on the train data.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of predictors.

Y [array-like, shape = [n_samples, n_targets]] Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

6.8 `sklearn.datasets`: Datasets

The *sklearn.datasets* module includes utilities to load datasets, including methods to load and fetch popular reference datasets. It also features some artificial data generators.

User guide: See the *Dataset loading utilities* section for further details.

6.8.1 Loaders

datasets.clear_data_home([data_home])	Delete all the content of the data home cache.
datasets.dump_svmlight_file(X, y, f[, ...])	Dump the dataset in svmlight / libsvm file format.
datasets.fetch_20newsgroups([data_home, ...])	Load the filenames and data from the 20 newsgroups dataset (classification).
datasets.fetch_20newsgroups_vectorized([...])	Load the 20 newsgroups dataset and vectorize it into token counts (classification).
datasets.fetch_california_housing([...])	Load the California housing dataset (regression).
datasets.fetch_covtype([data_home, ...])	Load the covertype dataset (classification).
datasets.fetch_kddcup99([subset, data_home, ...])	Load the kddcup99 dataset (classification).
datasets.fetch_lfw_pairs([subset, ...])	Load the Labeled Faces in the Wild (LFW) pairs dataset (classification).
datasets.fetch_lfw_people([data_home, ...])	Load the Labeled Faces in the Wild (LFW) people dataset (classification).
datasets.fetch_olivetti_faces([data_home, ...])	Load the Olivetti faces data-set from AT&T (classification).
datasets.fetch_openml([name, version, ...])	Fetch dataset from openml by name or dataset id.

Continued on next page

Table 6.47 – continued from previous page

datasets.fetch_rcv1([data_home, subset, ...])	Load the RCV1 multilabel dataset (classification).
datasets.fetch_species_distributions([...])	Loader for species distribution dataset from Phillips et.
datasets.get_data_home([data_home])	Return the path of the scikit-learn data dir.
datasets.load_boston([return_X_y])	Load and return the boston house-prices dataset (regression).
datasets.load_breast_cancer([return_X_y])	Load and return the breast cancer wisconsin dataset (classification).
datasets.load_diabetes([return_X_y])	Load and return the diabetes dataset (regression).
datasets.load_digits([n_class, return_X_y])	Load and return the digits dataset (classification).
datasets.load_files(container_path[, ...])	Load text files with categories as subfolder names.
datasets.load_iris([return_X_y])	Load and return the iris dataset (classification).
datasets.load_linnerud([return_X_y])	Load and return the linnerud dataset (multivariate regression).
datasets.load_sample_image(image_name)	Load the numpy array of a single sample image
datasets.load_sample_images()	Load sample images for image manipulation.
datasets.load_svmlight_file(f[, n_features, ...])	Load datasets in the svmlight / libsvm format into sparse CSR matrix
datasets.load_svmlight_files(files[, ...])	Load dataset from multiple files in SVMlight format
datasets.load_wine([return_X_y])	Load and return the wine dataset (classification).

sklearn.datasets.clear_data_home

sklearn.datasets.**clear_data_home**(*data_home=None*)

Delete all the content of the data home cache.

Parameters

data_home [str | None] The path to scikit-learn data dir.

sklearn.datasets.dump_svmlight_file

sklearn.datasets.**dump_svmlight_file**(*X*, *y*, *f*, *zero_based=True*, *comment=None*, *query_id=None*, *multilabel=False*)

Dump the dataset in svmlight / libsvm file format.

This format is a text-based format, with one sample per line. It does not store zero valued features hence is suitable for sparse dataset.

The first element of each line can be used to store a target variable to predict.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [{array-like, sparse matrix}, shape = [n_samples (, n_labels)]] Target values. Class labels must be an integer or float, or array-like objects of integer or float for multilabel classifications.

f [string or file-like in binary mode] If string, specifies the path that will contain the data. If file-like, data will be written to f. f should be opened in binary mode.

zero_based [boolean, optional] Whether column indices should be written zero-based (True) or one-based (False).

comment [string, optional] Comment to insert at the top of the file. This should be either a Unicode string, which will be encoded as UTF-8, or an ASCII byte string. If a comment

is given, then it will be preceded by one that identifies the file as having been dumped by scikit-learn. Note that not all tools grok comments in SVMlight files.

query_id [array-like, shape = [n_samples]] Array containing pairwise preference constraints (qid in svmlight format).

multilabel [boolean, optional] Samples may have several labels each (see https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel.html)

New in version 0.17: parameter *multilabel* to support multilabel datasets.

Examples using `sklearn.datasets.dump_svmlight_file`

- *Libsvm GUI*

`sklearn.datasets`.fetch_20newsgroups

`sklearn.datasets.`**`fetch_20newsgroups`**(*data_home=None*, *subset='train'*, *categories=None*, *shuffle=True*, *random_state=42*, *remove=()*, *download_if_missing=True*)

Load the filenames and data from the 20 newsgroups dataset (classification).

Download it if necessary.

Classes	20
Samples total	18846
Dimensionality	1
Features	text

Read more in the *User Guide*.

Parameters

data_home [optional, default: None] Specify a download and cache folder for the datasets. If None, all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

subset ['train' or 'test', 'all', optional] Select the dataset to load: 'train' for the training set, 'test' for the test set, 'all' for both, with shuffled ordering.

categories [None or collection of string or unicode] If None (default), load all the categories. If not None, list of category names to load (other categories ignored).

shuffle [bool, optional] Whether or not to shuffle the data: might be important for models that make the assumption that the samples are independent and identically distributed (i.i.d.), such as stochastic gradient descent.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple function calls. See *Glossary*.

remove [tuple] May contain any subset of ('headers', 'footers', 'quotes'). Each of these are kinds of text that will be detected and removed from the newsgroup posts, preventing classifiers from overfitting on metadata.

'headers' removes newsgroup headers, 'footers' removes blocks at the ends of posts that look like signatures, and 'quotes' removes lines that appear to be quoting another post.

'headers' follows an exact standard; the other filters are not always correct.

download_if_missing [optional, True by default] If False, raise an IOError if the data is not locally available instead of trying to download the data from the source site.

Returns

bunch [Bunch object with the following attribute:]

- bunch.data: list, length [n_samples]
- bunch.target: array, shape [n_samples]
- bunch.filenames: list, length [n_samples]
- bunch.DESCR: a description of the dataset.
- bunch.target_names: a list of categories of the returned data, length [n_classes]. This depends on the `categories` parameter.

Examples using `sklearn.datasets.fetch_20newsgroups`

- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*
- *Biclustering documents with the Spectral Co-clustering algorithm*
- *Column Transformer with Heterogeneous Data Sources*
- *Sample pipeline for text feature extraction and evaluation*
- *FeatureHasher and DictVectorizer Comparison*
- *Clustering text documents using k-means*
- *Classification of text documents using sparse features*

`sklearn.datasets`.fetch_20newsgroups_vectorized

`sklearn.datasets.`**`fetch_20newsgroups_vectorized`**(*subset='train'*, *remove=()*, *data_home=None*, *download_if_missing=True*, *return_X_y=False*)

Load the 20 newsgroups dataset and vectorize it into token counts (classification).

Download it if necessary.

This is a convenience function; the transformation is done using the default settings for *sklearn.feature_extraction.text.CountVectorizer*. For more advanced usage (stopword filtering, n-gram extraction, etc.), combine fetch_20newsgroups with a custom *sklearn.feature_extraction.text.CountVectorizer*, *sklearn.feature_extraction.text.HashingVectorizer*, *sklearn.feature_extraction.text.TfidfTransformer* or *sklearn.feature_extraction.text.TfidfVectorizer*.

Classes	20
Samples total	18846
Dimensionality	130107
Features	real

Read more in the *User Guide*.

Parameters

subset ['train' or 'test', 'all', optional] Select the dataset to load: 'train' for the training set, 'test' for the test set, 'all' for both, with shuffled ordering.

remove [tuple] May contain any subset of ('headers', 'footers', 'quotes'). Each of these are kinds of text that will be detected and removed from the newsgroup posts, preventing classifiers from overfitting on metadata.

'headers' removes newsgroup headers, 'footers' removes blocks at the ends of posts that look like signatures, and 'quotes' removes lines that appear to be quoting another post.

data_home [optional, default: None] Specify an download and cache folder for the datasets. If None, all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

download_if_missing [optional, True by default] If False, raise an IOError if the data is not locally available instead of trying to download the data from the source site.

return_X_y [boolean, default=False.] If True, returns `(data.data, data.target)` instead of a Bunch object.

New in version 0.20.

Returns

bunch [Bunch object with the following attribute:]

- bunch.data: sparse matrix, shape [n_samples, n_features]
- bunch.target: array, shape [n_samples]
- bunch.target_names: a list of categories of the returned data, length [n_classes].
- bunch.DESCR: a description of the dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

Examples using `sklearn.datasets.fetch_20newsgroups_vectorized`

- *The Johnson-Lindenstrauss bound for embedding with random projections*
- *Model Complexity Influence*
- *Multiclass sparse logisitic regression on newgroups20*

`sklearn.datasets`.fetch_california_housing

`sklearn.datasets.`**`fetch_california_housing`**(*data_home=None*, *download_if_missing=True*, *return_X_y=False*)

Load the California housing dataset (regression).

Samples total	20640
Dimensionality	8
Features	real
Target	real 0.15 - 5.

Read more in the *User Guide*.

Parameters

data_home [optional, default: None] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

download_if_missing [optional, default=True] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

return_X_y [boolean, default=False.] If True, returns `(data.data, data.target)` instead of a Bunch object.

New in version 0.20.

Returns

dataset [dict-like object with the following attributes:]

dataset.data [ndarray, shape [20640, 8]] Each row corresponding to the 8 feature values in order.

dataset.target [numpy array of shape (20640,)] Each value corresponds to the average house value in units of 100,000.

dataset.feature_names [array of length 8] Array of ordered feature names used in the dataset.

dataset.DESCR [string] Description of the California housing dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

Notes

This dataset consists of 20,640 samples and 9 features.

Examples using `sklearn.datasets.fetch_california_housing`

- *Imputing missing values with variants of IterativeImputer*
- *Partial Dependence Plots*
- *Compare the effect of different scalers on data with outliers*

sklearn.datasets.fetch_covtype

`sklearn.datasets.`**`fetch_covtype`**(*data_home=None*, *download_if_missing=True*, *random_state=None*, *shuffle=False*, *return_X_y=False*)

Load the covertype dataset (classification).

Download it if necessary.

Classes	7
Samples total	581012
Dimensionality	54
Features	int

Read more in the *User Guide*.

Parameters

data_home [string, optional] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

download_if_missing [boolean, default=True] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple function calls. See *Glossary*.

shuffle [bool, default=False] Whether to shuffle dataset.

return_X_y [boolean, default=False.] If True, returns `(data.data, data.target)` instead of a Bunch object.

New in version 0.20.

Returns

dataset [dict-like object with the following attributes:]

dataset.data [numpy array of shape (581012, 54)] Each row corresponds to the 54 features in the dataset.

dataset.target [numpy array of shape (581012,)] Each value corresponds to one of the 7 forest covertypes with values ranging between 1 to 7.

dataset.DESCR [string] Description of the forest covertype dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

sklearn.datasets.fetch_kddcup99

`sklearn.datasets.`**`fetch_kddcup99`**(*subset=None*, *data_home=None*, *shuffle=False*, *random_state=None*, *percent10=True*, *download_if_missing=True*, *return_X_y=False*)

Load the kddcup99 dataset (classification).

Download it if necessary.

Classes	23
Samples total	4898431
Dimensionality	41
Features	discrete (int) or continuous (float)

Read more in the *User Guide*.

New in version 0.18.

Parameters

subset [None, 'SA', 'SF', 'http', 'smtp'] To return the corresponding classical subsets of kddcup 99. If None, return the entire kddcup 99 dataset.

data_home [string, optional] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders. .. versionadded:: 0.19

shuffle [bool, default=False] Whether to shuffle dataset.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling and for selection of abnormal samples if `subset='SA'`. Pass an int for reproducible output across multiple function calls. See *Glossary*.

percent10 [bool, default=True] Whether to load only 10 percent of the data.

download_if_missing [bool, default=True] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and *target* object.

New in version 0.20.

Returns

data [Bunch]

Dictionary-like object, the interesting attributes are:

- 'data', the data to learn.
- 'target', the regression target for each sample.
- 'DESCR', a description of the dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

sklearn.datasets.fetch_lfw_pairs

sklearn.datasets.**fetch_lfw_pairs**(*subset='train'*, *data_home=None*, *funneled=True*, *resize=0.5*, *color=False*, *slice_=(slice(70*, *195*, *None)*, *slice(78*, *172*, *None))*, *download_if_missing=True*)

Load the Labeled Faces in the Wild (LFW) pairs dataset (classification).

Download it if necessary.

Classes	5749
Samples total	13233
Dimensionality	5828
Features	real, between 0 and 255

In the official README.txt this task is described as the "Restricted" task. As I am not sure as to implement the "Unrestricted" variant correctly, I left it as unsupported for now.

The original images are 250 x 250 pixels, but the default slice and resize arguments reduce them to 62 x 47.

Read more in the *User Guide*.

Parameters

subset [optional, default: 'train'] Select the dataset to load: 'train' for the development training set, 'test' for the development test set, and '10_folds' for the official evaluation set that is meant to be used with a 10-folds cross validation.

data_home [optional, default: None] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

funneled [boolean, optional, default: True] Download and use the funneled variant of the dataset.

resize [float, optional, default 0.5] Ratio used to resize the each face picture.

color [boolean, optional, default False] Keep the 3 RGB channels instead of averaging them to a single gray level channel. If color is True the shape of the data has one more dimension than the shape with color = False.

slice_ [optional] Provide a custom 2D slice (height, width) to extract the 'interesting' part of the jpeg files and avoid use statistical correlation from the background

download_if_missing [optional, True by default] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

Returns

The data is returned as a Bunch object with the following attributes:

data [numpy array of shape (2200, 5828). Shape depends on `subset`.] Each row corresponds to 2 ravel'd face images of original size 62 x 47 pixels. Changing the `slice_`, `resize` or `subset` parameters will change the shape of the output.

pairs [numpy array of shape (2200, 2, 62, 47). Shape depends on `subset`] Each row has 2 face images corresponding to same or different person from the dataset containing 5749 people. Changing the `slice_`, `resize` or `subset` parameters will change the shape of the output.

target [numpy array of shape (2200,). Shape depends on `subset`.] Labels associated to each pair of images. The two label values being different persons or the same person.

DESCR [string] Description of the Labeled Faces in the Wild (LFW) dataset.

sklearn.datasets.fetch_lfw_people

sklearn.datasets.**fetch_lfw_people**(*data_home=None*, *funneled=True*, *resize=0.5*, *min_faces_per_person=0*, *color=False*, *slice_=(slice(70, 195, None), slice(78, 172, None))*, *download_if_missing=True*, *return_X_y=False*)

Load the Labeled Faces in the Wild (LFW) people dataset (classification).

Download it if necessary.

Classes	5749
Samples total	13233
Dimensionality	5828
Features	real, between 0 and 255

Read more in the *User Guide*.

Parameters

data_home [optional, default: None] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

funneled [boolean, optional, default: True] Download and use the funneled variant of the dataset.

resize [float, optional, default 0.5] Ratio used to resize the each face picture.

min_faces_per_person [int, optional, default None] The extracted dataset will only retain pictures of people that have at least `min_faces_per_person` different pictures.

color [boolean, optional, default False] Keep the 3 RGB channels instead of averaging them to a single gray level channel. If color is True the shape of the data has one more dimension than the shape with color = False.

slice_ [optional] Provide a custom 2D slice (height, width) to extract the 'interesting' part of the jpeg files and avoid use statistical correlation from the background

download_if_missing [optional, True by default] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

return_X_y [boolean, default=False.] If True, returns `(dataset.data, dataset.target)` instead of a Bunch object. See below for more information about the `dataset.data` and `dataset.target` object.

New in version 0.20.

Returns

dataset [dict-like object with the following attributes:]

dataset.data [numpy array of shape (13233, 2914)] Each row corresponds to a ravelled face image of original size 62 x 47 pixels. Changing the `slice_` or resize parameters will change the shape of the output.

dataset.images [numpy array of shape (13233, 62, 47)] Each row is a face image corresponding to one of the 5749 people in the dataset. Changing the `slice_` or resize parameters will change the shape of the output.

dataset.target [numpy array of shape (13233,)] Labels associated to each face image. Those labels range from 0-5748 and correspond to the person IDs.

dataset.DESCR [string] Description of the Labeled Faces in the Wild (LFW) dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

Examples using `sklearn.datasets.fetch_lfw_people`

- *Faces recognition example using eigenfaces and SVMs*

`sklearn.datasets`.fetch_olivetti_faces

`sklearn.datasets.`**`fetch_olivetti_faces`**(*data_home=None*, *shuffle=False*, *random_state=0*, *download_if_missing=True*)

Load the Olivetti faces data-set from AT&T (classification).

Download it if necessary.

Classes	40
Samples total	400
Dimensionality	4096
Features	real, between 0 and 1

Read more in the *User Guide*.

Parameters

data_home [optional, default: None] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

shuffle [boolean, optional] If True the order of the dataset is shuffled to avoid having images of the same person grouped.

random_state [int, RandomState instance or None (default=0)] Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple function calls. See *Glossary*.

download_if_missing [optional, True by default] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

Returns

An object with the following attributes:

data [numpy array of shape (400, 4096)] Each row corresponds to a ravelled face image of original size 64 x 64 pixels.

images [numpy array of shape (400, 64, 64)] Each row is a face image corresponding to one of the 40 subjects of the dataset.

target [numpy array of shape (400,)] Labels associated to each face image. Those labels are ranging from 0-39 and correspond to the Subject IDs.

DESCR [string] Description of the modified Olivetti Faces Dataset.

Examples using `sklearn.datasets.fetch_olivetti_faces`

- *Face completion with a multi-output estimators*
- *Online learning of a dictionary of parts of faces*
- *Faces dataset decompositions*
- *Pixel importances with a parallel forest of trees*

`sklearn.datasets`.fetch_openml

`sklearn.datasets.`**`fetch_openml`**(*name=None*, *version='active'*, *data_id=None*, *data_home=None*, *target_column='default-target'*, *cache=True*, *return_X_y=False*)

Fetch dataset from openml by name or dataset id.

Datasets are uniquely identified by either an integer ID or by a combination of name and version (i.e. there might be multiple versions of the 'iris' dataset). Please give either name or data_id (not both). In case a name is given, a version can also be provided.

Read more in the *User Guide*.

Note: EXPERIMENTAL

The API is experimental (particularly the return value structure), and might have small backward-incompatible changes in future releases.

Parameters

name [str or None] String identifier of the dataset. Note that OpenML can have multiple datasets with the same name.

version [integer or 'active', default='active'] Version of the dataset. Can only be provided if also `name` is given. If 'active' the oldest version that's still active is used. Since there may be more than one active version of a dataset, and those versions may fundamentally be different from one another, setting an exact version is highly recommended.

data_id [int or None] OpenML ID of the dataset. The most specific way of retrieving a dataset. If data_id is not given, name (and potential version) are used to obtain a dataset.

data_home [string or None, default None] Specify another download and cache folder for the data sets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

target_column [string, list or None, default 'default-target'] Specify the column name in the data to use as target. If 'default-target', the standard target column a stored on the server is used. If `None`, all columns are returned as data and the target is `None`. If list (of strings), all columns with these names are returned as multi-target (Note: not all scikit-learn classifiers can handle all types of multi-output combinations)

cache [boolean, default=True] Whether to cache downloaded datasets using joblib.

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and *target* objects.

Returns

data [Bunch] Dictionary-like object, with attributes:

data [np.array or scipy.sparse.csr_matrix of floats] The feature matrix. Categorical features are encoded as ordinals.

target [np.array] The regression target or classification labels, if applicable. Dtype is float if numeric, and object if categorical.

DESCR [str] The full description of the dataset

feature_names [list] The names of the dataset columns

categories [dict] Maps each categorical feature name to a list of values, such that the value encoded as i is ith in the list.

details [dict] More metadata from OpenML

(data, target) [tuple if `return_X_y` is True]

Note: EXPERIMENTAL

This interface is **experimental** and subsequent releases may change attributes without notice (although there should only be minor changes to `data` and `target`).

Missing values in the 'data' are represented as NaN's. Missing values in 'target' are represented as NaN's (numerical target) or None (categorical target)

Examples using `sklearn.datasets.fetch_openml`

- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*
- *MNIST classfification using multinomial logistic + L1*
- *Early stopping of Stochastic Gradient Descent*
- *Classifier Chain*
- *Visualization of MLP weights on MNIST*

`sklearn.datasets.fetch_rcv1`

`sklearn.datasets.`**`fetch_rcv1`**(*data_home=None*, *subset='all'*, *download_if_missing=True*, *random_state=None*, *shuffle=False*, *return_X_y=False*)

Load the RCV1 multilabel dataset (classification).

Download it if necessary.

Version: RCV1-v2, vectors, full sets, topics multilabels.

Classes	103
Samples total	804414
Dimensionality	47236
Features	real, between 0 and 1

Read more in the *User Guide*.

New in version 0.17.

Parameters

data_home [string, optional] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

subset [string, 'train', 'test', or 'all', default='all'] Select the dataset to load: 'train' for the training set (23149 samples), 'test' for the test set (781265 samples), 'all' for both, with the training samples first if shuffle is False. This follows the official LYRL2004 chronological split.

download_if_missing [boolean, default=True] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple function calls. See *Glossary*.

shuffle [bool, default=False] Whether to shuffle dataset.

return_X_y [boolean, default=False.] If True, returns `(dataset.data, dataset.target)` instead of a Bunch object. See below for more information about the `dataset.data` and `dataset.target` object.

New in version 0.20.

Returns

dataset [dict-like object with the following attributes:]

dataset.data [scipy csr array, dtype np.float64, shape (804414, 47236)] The array has 0.16% of non zero values.

dataset.target [scipy csr array, dtype np.uint8, shape (804414, 103)] Each sample has a value of 1 in its categories, and 0 in others. The array has 3.15% of non zero values.

dataset.sample_id [numpy array, dtype np.uint32, shape (804414,)] Identification number of each sample, as ordered in dataset.data.

dataset.target_names [numpy array, dtype object, length (103)] Names of each target (RCV1 topics), as ordered in dataset.target.

dataset.DESCR [string] Description of the RCV1 dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.20.

sklearn.datasets.fetch_species_distributions

sklearn.datasets.**fetch_species_distributions**(*data_home=None*, *download_if_missing=True*)

Loader for species distribution dataset from Phillips et. al. (2006)

Read more in the *User Guide*.

Parameters

data_home [optional, default: None] Specify another download and cache folder for the datasets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

download_if_missing [optional, True by default] If False, raise a IOError if the data is not locally available instead of trying to download the data from the source site.

Returns

The data is returned as a Bunch object with the following attributes:

coverages [array, shape = [14, 1592, 1212]] These represent the 14 features measured at each point of the map grid. The latitude/longitude values for the grid are discussed below. Missing data is represented by the value -9999.

train [record array, shape = (1624,)] The training points for the data. Each point has three fields:

- train['species'] is the species name
- train['dd long'] is the longitude, in degrees
- train['dd lat'] is the latitude, in degrees

test [record array, shape = (620,)] The test points for the data. Same format as the training data.

Nx, Ny [integers] The number of longitudes (x) and latitudes (y) in the grid

x_left_lower_corner, y_left_lower_corner [floats] The (x,y) position of the lower-left corner, in degrees

grid_size [float] The spacing between points of the grid, in degrees

Notes

This dataset represents the geographic distribution of species. The dataset is provided by Phillips et. al. (2006).

The two species are:

- "Bradypus variegatus" , the Brown-throated Sloth.
- "Microryzomys minutus" , also known as the Forest Small Rice Rat, a rodent that lives in Peru, Colombia, Ecuador, Peru, and Venezuela.
- For an example of using this dataset with scikit-learn, see *examples/applications/plot_species_distribution_modeling.py*.

References

- "Maximum entropy modeling of species geographic distributions" S. J. Phillips, R. P. Anderson, R. E. Schapire - Ecological Modelling, 190:231-259, 2006.

Examples using `sklearn.datasets.fetch_species_distributions`

- *Species distribution modeling*
- *Kernel Density Estimate of Species Distributions*

sklearn.datasets.get_data_home

sklearn.datasets.**get_data_home**(*data_home=None*)
Return the path of the scikit-learn data dir.

This folder is used by some large dataset loaders to avoid downloading the data several times.

By default the data dir is set to a folder named 'scikit_learn_data' in the user home folder.

Alternatively, it can be set by the 'SCIKIT_LEARN_DATA' environment variable or programmatically by giving an explicit folder path. The '~' symbol is expanded to the user home folder.

If the folder does not already exist, it is automatically created.

Parameters

data_home [str | None] The path to scikit-learn data dir.

Examples using sklearn.datasets.get_data_home

- *Out-of-core classification of text documents*

sklearn.datasets.load_boston

sklearn.datasets.**load_boston**(*return_X_y=False*)
Load and return the boston house-prices dataset (regression).

Samples total	506
Dimensionality	13
Features	real, positive
Targets	real 5. - 50.

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and *target* object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the regression targets, 'DESCR', the full description of the dataset, and 'filename', the physical location of boston csv dataset (added in version `0.20`).

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

Notes

Changed in version 0.20: Fixed a wrong data point at [445, 0].

Examples

```
>>> from sklearn.datasets import load_boston
>>> boston = load_boston()
>>> print(boston.data.shape)
(506, 13)
```

Examples using `sklearn.datasets.load_boston`

- *Outlier detection on a real data set*
- *Model Complexity Influence*
- *Effect of transforming the targets in regression model*
- *Plot individual and voting regression predictions*
- *Gradient Boosting regression*
- *Feature selection using SelectFromModel and LassoCV*
- *Imputing missing values before building an estimator*
- *Plotting Cross-Validated Predictions*

`sklearn.datasets`.load_breast_cancer

`sklearn.datasets.`**`load_breast_cancer`**(*return_X_y=False*)

Load and return the breast cancer wisconsin dataset (classification).

The breast cancer dataset is a classic and very easy binary classification dataset.

Classes	2
Samples per class	212(M),357(B)
Samples total	569
Dimensionality	30
Features	real, positive

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and *target* object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset, 'filename', the physical location of breast cancer csv dataset (added in version `0.20`).

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

The copy of UCI ML Breast Cancer Wisconsin (Diagnostic) dataset is

downloaded from:

https://goo.gl/U2Uwz2

Examples

Let's say you are interested in the samples 10, 50, and 85, and want to know their class name.

```
>>> from sklearn.datasets import load_breast_cancer
>>> data = load_breast_cancer()
>>> data.target[[10, 50, 85]]
array([0, 1, 0])
>>> list(data.target_names)
['malignant', 'benign']
```

sklearn.datasets.load_diabetes

sklearn.datasets.**load_diabetes**(*return_X_y=False*)

Load and return the diabetes dataset (regression).

Samples total	442
Dimensionality	10
Features	real, -.2 < x < .2
Targets	integer 25 - 346

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and *target* object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the regression target for each sample, 'data_filename', the physical location of diabetes data csv dataset, and 'target_filename', the physical location of diabetes targets csv datataset (added in version `0.20`).

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

Examples using sklearn.datasets.load_diabetes

- *Cross-validation on diabetes Dataset Exercise*
- *Imputing missing values before building an estimator*
- *Lasso path using LARS*
- *Linear Regression Example*
- *Sparsity Example: Fitting only features 1 and 2*
- *Lasso and Elastic Net*
- *Lasso model selection: Cross-Validation / AIC / BIC*

sklearn.datasets.load_digits

sklearn.datasets.**load_digits**(*n_class=10*, *return_X_y=False*)

Load and return the digits dataset (classification).

Each datapoint is a 8x8 image of a digit.

Classes	10
Samples per class	~180
Samples total	1797
Dimensionality	64
Features	integers 0-16

Read more in the *User Guide*.

Parameters

n_class [integer, between 0 and 10, optional (default=10)] The number of classes to return.

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and *target* object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'images', the images corresponding to each sample, 'target', the classification labels for each sample, 'target_names', the meaning of the labels, and 'DESCR', the full description of the dataset.

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

This is a copy of the test set of the UCI ML hand-written digits datasets

https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

Examples

To load the data and visualize the images:

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
>>> print(digits.data.shape)
(1797, 64)
>>> import matplotlib.pyplot as plt
>>> plt.gray()
>>> plt.matshow(digits.images[0])
>>> plt.show()
```

Examples using sklearn.datasets.load_digits

- *The Johnson-Lindenstrauss bound for embedding with random projections*
- *Explicit feature map approximation for RBF kernels*
- *Recognizing hand-written digits*

- *Feature agglomeration*
- *Various Agglomerative Clustering on a 2D embedding of digits*
- *A demo of K-Means clustering on the handwritten digits data*
- *Pipelining: chaining a PCA and a logistic regression*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *The Digit Dataset*
- *Early stopping of Gradient Boosting*
- *Digits Classification Exercise*
- *Cross-validation on Digits Dataset Exercise*
- *Recursive feature elimination*
- *Comparing various online solvers*
- *L1 Penalty and Sparsity in Logistic Regression*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*
- *Plotting Validation Curves*
- *Parameter estimation using grid search with cross-validation*
- *Comparing randomized search and grid search for hyperparameter estimation*
- *Balance model complexity and cross-validated score*
- *Plotting Learning Curves*
- *Kernel Density Estimation*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Restricted Boltzmann Machine features for digit classification*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Label Propagation digits: Demonstrating performance*
- *Label Propagation digits active learning*

sklearn.datasets.load_files

sklearn.datasets.**load_files**(*container_path*, *description=None*, *categories=None*, *load_content=True*, *shuffle=True*, *encoding=None*, *decode_error='strict'*, *random_state=0*)

Load text files with categories as subfolder names.

Individual samples are assumed to be files stored a two levels folder structure such as the following:

container_folder/

category_1_folder/ file_1.txt file_2.txt ... file_42.txt

category_2_folder/ file_43.txt file_44.txt ...

The folder names are used as supervised signal label names. The individual file names are not important.

This function does not try to extract features into a numpy array or scipy sparse matrix. In addition, if load_content is false it does not try to load the files in memory.

To use text files in a scikit-learn classification or clustering algorithm, you will need to use the *sklearn.feature_extraction.text* module to build a feature extraction transformer that suits your problem.

If you set load_content=True, you should also specify the encoding of the text using the 'encoding' parameter. For many modern text files, 'utf-8' will be the correct encoding. If you leave encoding equal to None, then the content will be made of bytes instead of Unicode, and you will not be able to use most functions in *sklearn.feature_extraction.text*.

Similar feature extractors should be built for other kind of unstructured data input such as images, audio, video, ...

Read more in the *User Guide*.

Parameters

container_path [string or unicode] Path to the main folder holding one subfolder per category

description [string or unicode, optional (default=None)] A paragraph describing the characteristic of the dataset: its source, reference, etc.

categories [A collection of strings or None, optional (default=None)] If None (default), load all the categories. If not None, list of category names to load (other categories ignored).

load_content [boolean, optional (default=True)] Whether to load or not the content of the different files. If true a 'data' attribute containing the text information is present in the data structure returned. If not, a filenames attribute gives the path to the files.

shuffle [bool, optional (default=True)] Whether or not to shuffle the data: might be important for models that make the assumption that the samples are independent and identically distributed (i.i.d.), such as stochastic gradient descent.

encoding [string or None (default is None)] If None, do not try to decode the content of the files (e.g. for images or other non-text content). If not None, encoding to use to decode text files to Unicode if load_content is True.

decode_error [{'strict', 'ignore', 'replace'}, optional] Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given `encoding`. Passed as keyword argument 'errors' to bytes.decode.

random_state [int, RandomState instance or None (default=0)] Determines random number generation for dataset shuffling. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: either data, the raw text data to learn, or 'filenames', the files holding it, 'target', the classification labels (integer index), 'target_names', the meaning of the labels, and 'DESCR', the full description of the dataset.

sklearn.datasets.load_iris

sklearn.datasets.**load_iris**(*return_X_y=False*)

Load and return the iris dataset (classification).

The iris dataset is a classic and very easy multi-class classification dataset.

Classes	3
Samples per class	50
Samples total	150
Dimensionality	4
Features	real, positive

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and *target* object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, 'DESCR', the full description of the dataset, 'filename', the physical location of iris csv dataset (added in version `0.20`).

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

Notes

Changed in version 0.20: Fixed two wrong data points according to Fisher's paper. The new version is the same as in R, but not as in the UCI Machine Learning Repository.

Examples

Let's say you are interested in the samples 10, 25, and 50, and want to know their class name.

```
>>> from sklearn.datasets import load_iris
>>> data = load_iris()
>>> data.target[[10, 25, 50]]
array([0, 0, 1])
>>> list(data.target_names)
['setosa', 'versicolor', 'virginica']
```

Examples using `sklearn.datasets.load_iris`

- *Plot classification probability*
- *K-means Clustering*
- *Concatenating multiple feature extraction methods*
- *The Iris Dataset*
- *PCA example with Iris Data-set*
- *Incremental PCA*
- *Comparison of LDA and PCA 2D projection of Iris dataset*
- *Plot the decision boundaries of a VotingClassifier*

- *Early stopping of Gradient Boosting*
- *Plot the decision surfaces of ensembles of trees on the iris dataset*
- *SVM Exercise*
- *Test with permutations the significance of a classification score*
- *Univariate Feature Selection*
- *Gaussian process classification (GPC) on iris dataset*
- *Regularization path of L1- Logistic Regression*
- *Logistic Regression 3-class Classifier*
- *Plot multi-class SGD on the iris dataset*
- *GMM covariances*
- *Receiver Operating Characteristic (ROC) with cross validation*
- *Nested versus non-nested cross-validation*
- *Confusion matrix*
- *Receiver Operating Characteristic (ROC)*
- *Precision-Recall*
- *Nearest Neighbors Classification*
- *Nearest Centroid Classification*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Decision boundary of label propagation versus SVM on the Iris dataset*
- *SVM with custom kernel*
- *SVM-Anova: SVM with univariate feature selection*
- *Plot different SVM classifiers in the iris dataset*
- *RBF SVM parameters*
- *Plot the decision surface of a decision tree on the iris dataset*
- *Understanding the decision tree structure*

sklearn.datasets.load_linnerud

sklearn.datasets.**load_linnerud**(*return_X_y=False*)

Load and return the linnerud dataset (multivariate regression).

Samples total	20
Dimensionality	3 (for both data and target)
Features	integer
Targets	integer

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and *target* object.

New in version 0.18.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data' and 'target', the two multivariate datasets, with 'data' corresponding to the exercise and 'target' corresponding to the physiological measurements, as well as 'feature_names' and 'target_names'. In addition, you will also have access to 'data_filename', the physical location of linnerud data csv dataset, and 'target_filename', the physical location of linnerud targets csv datataset (added in version `0.20`).

(data, target) [tuple if `return_X_y` is True] New in version 0.18.

sklearn.datasets.load_sample_image

sklearn.datasets.**load_sample_image**(*image_name*)

Load the numpy array of a single sample image

Read more in the *User Guide*.

Parameters

image_name [{`china.jpg`, `flower.jpg`}] The name of the sample image loaded

Returns

img [3D array] The image as a numpy array: height x width x color

Examples

```
>>> from sklearn.datasets import load_sample_image
>>> china = load_sample_image('china.jpg')
>>> china.dtype
dtype('uint8')
>>> china.shape
(427, 640, 3)
>>> flower = load_sample_image('flower.jpg')
>>> flower.dtype
dtype('uint8')
>>> flower.shape
(427, 640, 3)
```

Examples using sklearn.datasets.load_sample_image

- *Color Quantization using K-Means*

sklearn.datasets.load_sample_images

sklearn.datasets.**load_sample_images**()

Load sample images for image manipulation.

Loads both, `china` and `flower`.

Read more in the *User Guide*.

Returns

data [Bunch] Dictionary-like object with the following attributes : 'images', the two sample images, 'filenames', the file names for the images, and 'DESCR' the full description of the dataset.

Examples

To load the data and visualize the images:

```
>>> from sklearn.datasets import load_sample_images
>>> dataset = load_sample_images()
>>> len(dataset.images)
2
>>> first_img_data = dataset.images[0]
>>> first_img_data.shape
(427, 640, 3)
>>> first_img_data.dtype
dtype('uint8')
```

sklearn.datasets.load_svmlight_file

sklearn.datasets.**load_svmlight_file**(*f*, *n_features=None*, *dtype=<class 'numpy.float64'>*, *multilabel=False*, *zero_based='auto'*, *query_id=False*, *offset=0*, *length=-1*)

Load datasets in the svmlight / libsvm format into sparse CSR matrix

This format is a text-based format, with one sample per line. It does not store zero valued features hence is suitable for sparse dataset.

The first element of each line can be used to store a target variable to predict.

This format is used as the default format for both svmlight and the libsvm command line programs.

Parsing a text based source can be expensive. When working on repeatedly on the same dataset, it is recommended to wrap this loader with joblib.Memory.cache to store a memmapped backup of the CSR results of the first call and benefit from the near instantaneous loading of memmapped structures for the subsequent calls.

In case the file contains a pairwise preference constraint (known as "qid" in the svmlight format) these are ignored unless the query_id parameter is set to True. These pairwise preference constraints can be used to constraint the combination of samples when using pairwise loss functions (as is the case in some learning to rank problems) so that only pairs with the same query_id value are considered.

This implementation is written in Cython and is reasonably fast. However, a faster API-compatible loader is also available at:

https://github.com/mblondel/svmlight-loader

Parameters

f [{str, file-like, int}] (Path to) a file to load. If a path ends in ".gz" or ".bz2", it will be uncompressed on the fly. If an integer is passed, it is assumed to be a file descriptor. A file-like or file descriptor will not be closed by this function. A file-like object must be opened in binary mode.

n_features [int or None] The number of features to use. If None, it will be inferred. This argument is useful to load several files that are subsets of a bigger sliced dataset: each subset might not have examples of every feature, hence the inferred shape might vary from one slice to another. n_features is only required if `offset` or `length` are passed a non-default value.

dtype [numpy data type, default np.float64] Data type of dataset to be loaded. This will be the data type of the output numpy arrays `X` and `y`.

multilabel [boolean, optional, default False] Samples may have several labels each (see https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel.html)

zero_based [boolean or "auto", optional, default "auto"] Whether column indices in f are zero-based (True) or one-based (False). If column indices are one-based, they are transformed to zero-based to match Python/NumPy conventions. If set to "auto", a heuristic check is applied to determine this from the file contents. Both kinds of files occur "in the wild", but they are unfortunately not self-identifying. Using "auto" or True should always be safe when no `offset` or `length` is passed. If `offset` or `length` are passed, the "auto" mode falls back to `zero_based=True` to avoid having the heuristic check yield inconsistent results on different segments of the file.

query_id [boolean, default False] If True, will return the query_id array for each file.

offset [integer, optional, default 0] Ignore the offset first bytes by seeking forward, then discarding the following bytes up until the next new line character.

length [integer, optional, default -1] If strictly positive, stop reading any new line of data once the position in the file has reached the (offset + length) bytes threshold.

Returns

X [scipy.sparse matrix of shape (n_samples, n_features)]

y [ndarray of shape (n_samples,), or, in the multilabel a list of] tuples of length n_samples.

query_id [array of shape (n_samples,)] query_id for each sample. Only returned when query_id is set to True.

See also:

load_svmlight_files similar function for loading multiple files in this format, enforcing the same number of features/columns on all of them.

Examples

To use joblib.Memory to cache the svmlight file:

```
from joblib import Memory
from .datasets import load_svmlight_file
mem = Memory("./mycache")

@mem.cache
def get_data():
    data = load_svmlight_file("mysvmlightfile")
    return data[0], data[1]

X, y = get_data()
```

sklearn.datasets.load_svmlight_files

sklearn.datasets.**load_svmlight_files**(*files*, *n_features=None*, *dtype=<class 'numpy.float64'>*, *multilabel=False*, *zero_based='auto'*, *query_id=False*, *offset=0*, *length=-1*)

Load dataset from multiple files in SVMlight format

This function is equivalent to mapping load_svmlight_file over a list of files, except that the results are concatenated into a single, flat list and the samples vectors are constrained to all have the same number of features.

In case the file contains a pairwise preference constraint (known as "qid" in the svmlight format) these are ignored unless the query_id parameter is set to True. These pairwise preference constraints can be used to constraint the combination of samples when using pairwise loss functions (as is the case in some learning to rank problems) so that only pairs with the same query_id value are considered.

Parameters

files [iterable over {str, file-like, int}] (Paths of) files to load. If a path ends in ".gz" or ".bz2", it will be uncompressed on the fly. If an integer is passed, it is assumed to be a file descriptor. File-likes and file descriptors will not be closed by this function. File-like objects must be opened in binary mode.

n_features [int or None] The number of features to use. If None, it will be inferred from the maximum column index occurring in any of the files.

This can be set to a higher value than the actual number of features in any of the input files, but setting it to a lower value will cause an exception to be raised.

dtype [numpy data type, default np.float64] Data type of dataset to be loaded. This will be the data type of the output numpy arrays `X` and `y`.

multilabel [boolean, optional] Samples may have several labels each (see https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel.html)

zero_based [boolean or "auto", optional] Whether column indices in f are zero-based (True) or one-based (False). If column indices are one-based, they are transformed to zero-based to match Python/NumPy conventions. If set to "auto", a heuristic check is applied to determine this from the file contents. Both kinds of files occur "in the wild", but they are unfortunately not self-identifying. Using "auto" or True should always be safe when no offset or length is passed. If offset or length are passed, the "auto" mode falls back to zero_based=True to avoid having the heuristic check yield inconsistent results on different segments of the file.

query_id [boolean, defaults to False] If True, will return the query_id array for each file.

offset [integer, optional, default 0] Ignore the offset first bytes by seeking forward, then discarding the following bytes up until the next new line character.

length [integer, optional, default -1] If strictly positive, stop reading any new line of data once the position in the file has reached the (offset + length) bytes threshold.

Returns

[X1, y1, . . . , Xn, yn]

where each (Xi, yi) pair is the result from load_svmlight_file(files[i]).

If query_id is set to True, this will return instead [X1, y1, q1,

. . . , Xn, yn, qn] where (Xi, yi, qi) is the result from

load_svmlight_file(files[i])

See also:

load_svmlight_file

Notes

When fitting a model to a matrix X_train and evaluating it against a matrix X_test, it is essential that X_train and X_test have the same number of features (X_train.shape[1] == X_test.shape[1]). This may not be the case if you load the files individually with load_svmlight_file.

sklearn.datasets.load_wine

sklearn.datasets.**load_wine**(*return_X_y=False*)

Load and return the wine dataset (classification).

New in version 0.18.

The wine dataset is a classic and very easy multi-class classification dataset.

Classes	3
Samples per class	[59,71,48]
Samples total	178
Dimensionality	13
Features	real, positive

Read more in the *User Guide*.

Parameters

return_X_y [boolean, default=False.] If True, returns `(data, target)` instead of a Bunch object. See below for more information about the `data` and *target* object.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.

(data, target) [tuple if `return_X_y` is True]

The copy of UCI ML Wine Data Set dataset is downloaded and modified to fit

standard format from:

https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data

Examples

Let's say you are interested in the samples 10, 80, and 140, and want to know their class name.

```
>>> from sklearn.datasets import load_wine
>>> data = load_wine()
>>> data.target[[10, 80, 140]]
array([0, 1, 2])
>>> list(data.target_names)
['class_0', 'class_1', 'class_2']
```

Examples using `sklearn.datasets.load_wine`

- *Importance of Feature Scaling*

6.8.2 Samples generator

datasets.make_biclusters(shape, n_clusters)	Generate an array with constant block diagonal structure for biclustering.
datasets.make_blobs([n_samples, n_features, ...])	Generate isotropic Gaussian blobs for clustering.
datasets.make_checkerboard(shape, n_clusters)	Generate an array with block checkerboard structure for biclustering.
datasets.make_circles([n_samples, shuffle, ...])	Make a large circle containing a smaller circle in 2d.
datasets.make_classification([n_samples, ...])	Generate a random n-class classification problem.
datasets.make_friedman1([n_samples, ...])	Generate the "Friedman #1" regression problem
datasets.make_friedman2([n_samples, noise, ...])	Generate the "Friedman #2" regression problem
datasets.make_friedman3([n_samples, noise, ...])	Generate the "Friedman #3" regression problem
datasets.make_gaussian_quantiles([mean, ...])	Generate isotropic Gaussian and label samples by quantile
datasets.make_hastie_10_2([n_samples, ...])	Generates data for binary classification used in Hastie et al.
datasets.make_low_rank_matrix([n_samples, ...])	Generate a mostly low rank matrix with bell-shaped singular values
datasets.make_moons([n_samples, shuffle, ...])	Make two interleaving half circles
datasets.make_multilabel_classification([...])	Generate a random multilabel classification problem.
datasets.make_regression([n_samples, ...])	Generate a random regression problem.
datasets.make_s_curve([n_samples, noise, ...])	Generate an S curve dataset.
datasets.make_sparse_coded_signal(n_samples, ...)	Generate a signal as a sparse combination of dictionary elements.
datasets.make_sparse_spd_matrix([dim, ...])	Generate a sparse symmetric definite positive matrix.
datasets.make_sparse_uncorrelated([...])	Generate a random regression problem with sparse uncorrelated design
datasets.make_spd_matrix(n_dim[, random_state])	Generate a random symmetric, positive-definite matrix.
datasets.make_swiss_roll([n_samples, noise, ...])	Generate a swiss roll dataset.

`sklearn.datasets.make_biclusters`

`sklearn.datasets.`**`make_biclusters`**(*shape*, *n_clusters*, *noise=0.0*, *minval=10*, *maxval=100*, *shuffle=True*, *random_state=None*)

Generate an array with constant block diagonal structure for biclustering.

Read more in the *User Guide*.

Parameters

shape [iterable (n_rows, n_cols)] The shape of the result.

n_clusters [integer] The number of biclusters.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise.

minval [int, optional (default=10)] Minimum value of a bicluster.

maxval [int, optional (default=100)] Maximum value of a bicluster.

shuffle [boolean, optional (default=True)] Shuffle the samples.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape `shape`] The generated array.

rows [array of shape (n_clusters, X.shape[0],)] The indicators for cluster membership of each row.

cols [array of shape (n_clusters, X.shape[1],)] The indicators for cluster membership of each column.

See also:

make_checkerboard

References

[1]

Examples using `sklearn.datasets.make_biclusters`

- *A demo of the Spectral Co-Clustering algorithm*

`sklearn.datasets.make_blobs`

`sklearn.datasets.`**`make_blobs`**(*n_samples=100*, *n_features=2*, *centers=None*, *cluster_std=1.0*, *center_box=(-10.0*, *10.0)*, *shuffle=True*, *random_state=None*)

Generate isotropic Gaussian blobs for clustering.

Read more in the *User Guide*.

Parameters

n_samples [int or array-like, optional (default=100)] If int, it is the total number of points equally divided among clusters. If array-like, each element of the sequence indicates the number of samples per cluster.

n_features [int, optional (default=2)] The number of features for each sample.

centers [int or array of shape [n_centers, n_features], optional] (default=None) The number of centers to generate, or the fixed center locations. If n_samples is an int and centers is None, 3 centers are generated. If n_samples is array-like, centers must be either None or an array of length equal to the length of n_samples.

cluster_std [float or sequence of floats, optional (default=1.0)] The standard deviation of the clusters.

center_box [pair of floats (min, max), optional (default=(-10.0, 10.0))] The bounding box for each cluster center when centers are generated at random.

shuffle [boolean, optional (default=True)] Shuffle the samples.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

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

y [array of shape [n_samples]] The integer labels for cluster membership of each sample.

See also:

make_classification a more intricate variant

Examples

```
>>> from sklearn.datasets.samples_generator import make_blobs
>>> X, y = make_blobs(n_samples=10, centers=3, n_features=2,
...                   random_state=0)
>>> print(X.shape)
(10, 2)
>>> y
array([0, 0, 1, 0, 2, 2, 2, 1, 1, 0])
>>> X, y = make_blobs(n_samples=[3, 3, 4], centers=None, n_features=2,
...                   random_state=0)
>>> print(X.shape)
(10, 2)
>>> y
array([0, 1, 2, 0, 2, 2, 2, 1, 1, 0])
```

Examples using `sklearn.datasets.make_blobs`

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *Probability calibration of classifiers*
- *Probability Calibration for 3-class classification*
- *Normal and Shrinkage Linear Discriminant Analysis for classification*
- *A demo of the mean-shift clustering algorithm*
- *Demonstration of k-means assumptions*
- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *Inductive Clustering*
- *Compare BIRCH and MiniBatchKMeans*
- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*
- *Comparing different hierarchical linkage methods on toy datasets*
- *Selecting the number of clusters with silhouette analysis on KMeans clustering*
- *Comparing different clustering algorithms on toy datasets*

- *Plot randomly generated classification dataset*
- *SGD: Maximum margin separating hyperplane*
- *Plot multinomial and One-vs-Rest Logistic Regression*
- *Demonstrating the different strategies of KBinsDiscretizer*
- *SVM: Maximum margin separating hyperplane*
- *SVM: Separating hyperplane for unbalanced classes*

sklearn.datasets.make_checkerboard

sklearn.datasets.**make_checkerboard**(*shape*, *n_clusters*, *noise=0.0*, *minval=10*, *maxval=100*, *shuffle=True*, *random_state=None*)

Generate an array with block checkerboard structure for biclustering.

Read more in the *User Guide*.

Parameters

shape [iterable (n_rows, n_cols)] The shape of the result.

n_clusters [integer or iterable (n_row_clusters, n_column_clusters)] The number of row and column clusters.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise.

minval [int, optional (default=10)] Minimum value of a bicluster.

maxval [int, optional (default=100)] Maximum value of a bicluster.

shuffle [boolean, optional (default=True)] Shuffle the samples.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape `shape`] The generated array.

rows [array of shape (n_clusters, X.shape[0],)] The indicators for cluster membership of each row.

cols [array of shape (n_clusters, X.shape[1],)] The indicators for cluster membership of each column.

See also:

make_biclusters

References

[1]

Examples using sklearn.datasets.make_checkerboard

- *A demo of the Spectral Biclustering algorithm*

sklearn.datasets.make_circles

sklearn.datasets.**make_circles**(*n_samples=100*, *shuffle=True*, *noise=None*, *random_state=None*, *factor=0.8*)

Make a large circle containing a smaller circle in 2d.

A simple toy dataset to visualize clustering and classification algorithms.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The total number of points generated. If odd, the inner circle will have one point more than the outer circle.

shuffle [bool, optional (default=True)] Whether to shuffle the samples.

noise [double or None (default=None)] Standard deviation of Gaussian noise added to the data.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling and noise. Pass an int for reproducible output across multiple function calls. See *Glossary*.

factor [0 < double < 1 (default=.8)] Scale factor between inner and outer circle.

Returns

X [array of shape [n_samples, 2]] The generated samples.

y [array of shape [n_samples]] The integer labels (0 or 1) for class membership of each sample.

Examples using sklearn.datasets.make_circles

- *Classifier comparison*
- *Comparing different hierarchical linkage methods on toy datasets*
- *Comparing different clustering algorithms on toy datasets*
- *Kernel PCA*
- *Hashing feature transformation using Totally Random Trees*
- *t-SNE: The effect of various perplexity values on the shape*
- *Varying regularization in Multi-layer Perceptron*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Feature discretization*
- *Label Propagation learning a complex structure*

sklearn.datasets.make_classification

sklearn.datasets.**make_classification**(*n_samples=100*, *n_features=20*, *n_informative=2*, *n_redundant=2*, *n_repeated=0*, *n_classes=2*, *n_clusters_per_class=2*, *weights=None*, *flip_y=0.01*, *class_sep=1.0*, *hypercube=True*, *shift=0.0*, *scale=1.0*, *shuffle=True*, *random_state=None*)

Generate a random n-class classification problem.

This initially creates clusters of points normally distributed (std=1) about vertices of an `n_informative`-dimensional hypercube with sides of length `2*class_sep` and assigns an equal number of clusters to each class. It introduces interdependence between these features and adds various types of further noise to the data.

Without shuffling, `X` horizontally stacks features in the following order: the primary `n_informative` features, followed by `n_redundant` linear combinations of the informative features, followed by `n_repeated` duplicates, drawn randomly with replacement from the informative and redundant features. The remaining features are filled with random noise. Thus, without shuffling, all useful features are contained in the columns `X[:, :n_informative + n_redundant + n_repeated]`.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=20)] The total number of features. These comprise `n_informative` informative features, `n_redundant` redundant features, `n_repeated` duplicated features and `n_features-n_informative-n_redundant-n_repeated` useless features drawn at random.

n_informative [int, optional (default=2)] The number of informative features. Each class is composed of a number of gaussian clusters each located around the vertices of a hypercube in a subspace of dimension `n_informative`. For each cluster, informative features are drawn independently from N(0, 1) and then randomly linearly combined within each cluster in order to add covariance. The clusters are then placed on the vertices of the hypercube.

n_redundant [int, optional (default=2)] The number of redundant features. These features are generated as random linear combinations of the informative features.

n_repeated [int, optional (default=0)] The number of duplicated features, drawn randomly from the informative and the redundant features.

n_classes [int, optional (default=2)] The number of classes (or labels) of the classification problem.

n_clusters_per_class [int, optional (default=2)] The number of clusters per class.

weights [list of floats or None (default=None)] The proportions of samples assigned to each class. If None, then classes are balanced. Note that if `len(weights) == n_classes - 1`, then the last class weight is automatically inferred. More than `n_samples` samples may be returned if the sum of `weights` exceeds 1.

flip_y [float, optional (default=0.01)] The fraction of samples whose class are randomly exchanged. Larger values introduce noise in the labels and make the classification task harder.

class_sep [float, optional (default=1.0)] The factor multiplying the hypercube size. Larger values spread out the clusters/classes and make the classification task easier.

hypercube [boolean, optional (default=True)] If True, the clusters are put on the vertices of a hypercube. If False, the clusters are put on the vertices of a random polytope.

shift [float, array of shape [n_features] or None, optional (default=0.0)] Shift features by the specified value. If None, then features are shifted by a random value drawn in [-class_sep, class_sep].

scale [float, array of shape [n_features] or None, optional (default=1.0)] Multiply features by the specified value. If None, then features are scaled by a random value drawn in [1, 100]. Note that scaling happens after shifting.

shuffle [boolean, optional (default=True)] Shuffle the samples and the features.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

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

y [array of shape [n_samples]] The integer labels for class membership of each sample.

See also:

make_blobs simplified variant

make_multilabel_classification unrelated generator for multilabel tasks

Notes

The algorithm is adapted from Guyon [1] and was designed to generate the "Madelon" dataset.

References

[1]

Examples using `sklearn.datasets.make_classification`

- *Comparison of Calibration of Classifiers*
- *Probability Calibration curves*
- *Classifier comparison*
- *Plot randomly generated classification dataset*
- *Feature importances with forests of trees*
- *OOB Errors for Random Forests*
- *Feature transformations with ensembles of trees*
- *Pipeline Anova SVM*
- *Recursive feature elimination with cross-validation*
- *Neighborhood Components Analysis Illustration*
- *Varying regularization in Multi-layer Perceptron*
- *Feature discretization*
- *Scaling the regularization parameter for SVCs*

`sklearn.datasets.make_friedman1`

`sklearn.datasets.`**`make_friedman1`**(*n_samples=100*, *n_features=10*, *noise=0.0*, *random_state=None*)

Generate the "Friedman #1" regression problem

This dataset is described in Friedman [1] and Breiman [2].

Inputs *X* are independent features uniformly distributed on the interval [0, 1]. The output *y* is created according to the formula:

```
y(X) = 10 * sin(pi * X[:, 0] * X[:, 1]) + 20 * (X[:, 2] - 0.5) ** 2 + 10 * X[:,
↪3] + 5 * X[:, 4] + noise * N(0, 1).
```

Out of the *n_features* features, only 5 are actually used to compute *y*. The remaining features are independent of *y*.

The number of features has to be >= 5.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=10)] The number of features. Should be at least 5.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise applied to the output.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset noise. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

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

y [array of shape [n_samples]] The output values.

References

[1], *[2]*

sklearn.datasets.make_friedman2

sklearn.datasets.**make_friedman2**(*n_samples=100*, *noise=0.0*, *random_state=None*)

Generate the "Friedman #2" regression problem

This dataset is described in Friedman [1] and Breiman [2].

Inputs *X* are 4 independent features uniformly distributed on the intervals:

```
0 <= X[:, 0] <= 100,
40 * pi <= X[:, 1] <= 560 * pi,
0 <= X[:, 2] <= 1,
1 <= X[:, 3] <= 11.
```

The output *y* is created according to the formula:

```
y(X) = (X[:, 0] ** 2 + (X[:, 1] * X[:, 2]  - 1 / (X[:, 1] * X[:, 3])) ** 2) ** 0.
↪5 + noise * N(0, 1).
```

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise applied to the output.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset noise. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 4]] The input samples.

y [array of shape [n_samples]] The output values.

References

[1], *[2]*

sklearn.datasets.make_friedman3

sklearn.datasets.**make_friedman3**(*n_samples=100*, *noise=0.0*, *random_state=None*)

Generate the "Friedman #3" regression problem

This dataset is described in Friedman [1] and Breiman [2].

Inputs *X* are 4 independent features uniformly distributed on the intervals:

```
0 <= X[:, 0] <= 100,
40 * pi <= X[:, 1] <= 560 * pi,
0 <= X[:, 2] <= 1,
1 <= X[:, 3] <= 11.
```

The output *y* is created according to the formula:

```
y(X) = arctan((X[:, 1] * X[:, 2] - 1 / (X[:, 1] * X[:, 3])) / X[:, 0]) + noise *␣
↪N(0, 1).
```

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise applied to the output.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset noise. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 4]] The input samples.

y [array of shape [n_samples]] The output values.

References

[1], *[2]*

sklearn.datasets.make_gaussian_quantiles

sklearn.datasets.**make_gaussian_quantiles**(*mean=None*, *cov=1.0*, *n_samples=100*, *n_features=2*, *n_classes=3*, *shuffle=True*, *random_state=None*)

Generate isotropic Gaussian and label samples by quantile

This classification dataset is constructed by taking a multi-dimensional standard normal distribution and defining classes separated by nested concentric multi-dimensional spheres such that roughly equal numbers of samples are in each class (quantiles of the χ^2 distribution).

Read more in the *User Guide*.

Parameters

mean [array of shape [n_features], optional (default=None)] The mean of the multi-dimensional normal distribution. If None then use the origin (0, 0, ...).

cov [float, optional (default=1.)] The covariance matrix will be this value times the unit matrix. This dataset only produces symmetric normal distributions.

n_samples [int, optional (default=100)] The total number of points equally divided among classes.

n_features [int, optional (default=2)] The number of features for each sample.

n_classes [int, optional (default=3)] The number of classes

shuffle [boolean, optional (default=True)] Shuffle the samples.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

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

y [array of shape [n_samples]] The integer labels for quantile membership of each sample.

Notes

The dataset is from Zhu et al [1].

References

[1]

Examples using sklearn.datasets.make_gaussian_quantiles

- *Plot randomly generated classification dataset*
- *Two-class AdaBoost*
- *Multi-class AdaBoosted Decision Trees*

sklearn.datasets.make_hastie_10_2

sklearn.datasets.**make_hastie_10_2**(*n_samples=12000*, *random_state=None*)

Generates data for binary classification used in Hastie et al. 2009, Example 10.2.

The ten features are standard independent Gaussian and the target `y` is defined by:

```
y[i] = 1 if np.sum(X[i] ** 2) > 9.34 else -1
```

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=12000)] The number of samples.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 10]] The input samples.

y [array of shape [n_samples]] The output values.

See also:

make_gaussian_quantiles a generalization of this dataset approach

References

[1]

Examples using sklearn.datasets.make_hastie_10_2

- *Gradient Boosting regularization*
- *Discrete versus Real AdaBoost*
- *Early stopping of Gradient Boosting*
- *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV*

sklearn.datasets.make_low_rank_matrix

sklearn.datasets.**make_low_rank_matrix**(*n_samples=100*, *n_features=100*, *effective_rank=10*, *tail_strength=0.5*, *random_state=None*)

Generate a mostly low rank matrix with bell-shaped singular values

Most of the variance can be explained by a bell-shaped curve of width effective_rank: the low rank part of the singular values profile is:

```
(1 - tail_strength) * exp(-1.0 * (i / effective_rank) ** 2)
```

The remaining singular values' tail is fat, decreasing as:

```
tail_strength * exp(-0.1 * i / effective_rank).
```

The low rank part of the profile can be considered the structured signal part of the data while the tail can be considered the noisy part of the data that cannot be summarized by a low number of linear components (singular vectors).

This kind of singular profiles is often seen in practice, for instance:

- gray level pictures of faces
- TF-IDF vectors of text documents crawled from the web

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=100)] The number of features.

effective_rank [int, optional (default=10)] The approximate number of singular vectors required to explain most of the data by linear combinations.

tail_strength [float between 0.0 and 1.0, optional (default=0.5)] The relative importance of the fat noisy tail of the singular values profile.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, n_features]] The matrix.

sklearn.datasets.make_moons

sklearn.datasets.**make_moons**(*n_samples=100*, *shuffle=True*, *noise=None*, *random_state=None*)
Make two interleaving half circles

A simple toy dataset to visualize clustering and classification algorithms. Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The total number of points generated.

shuffle [bool, optional (default=True)] Whether to shuffle the samples.

noise [double or None (default=None)] Standard deviation of Gaussian noise added to the data.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset shuffling and noise. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 2]] The generated samples.

y [array of shape [n_samples]] The integer labels (0 or 1) for class membership of each sample.

Examples using sklearn.datasets.make_moons

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *Classifier comparison*
- *Comparing different hierarchical linkage methods on toy datasets*

- *Comparing different clustering algorithms on toy datasets*
- *Varying regularization in Multi-layer Perceptron*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Feature discretization*

sklearn.datasets.make_multilabel_classification

sklearn.datasets.**make_multilabel_classification**(*n_samples=100*, *n_features=20*, *n_classes=5*, *n_labels=2*, *length=50*, *allow_unlabeled=True*, *sparse=False*, *return_indicator='dense'*, *return_distributions=False*, *random_state=None*)

Generate a random multilabel classification problem.

For each sample, the generative process is:

- pick the number of labels: n ~ Poisson(n_labels)
- n times, choose a class c: c ~ Multinomial(theta)
- pick the document length: k ~ Poisson(length)
- k times, choose a word: w ~ Multinomial(theta_c)

In the above process, rejection sampling is used to make sure that n is never zero or more than `n_classes`, and that the document length is never zero. Likewise, we reject classes which have already been chosen.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=20)] The total number of features.

n_classes [int, optional (default=5)] The number of classes of the classification problem.

n_labels [int, optional (default=2)] The average number of labels per instance. More precisely, the number of labels per sample is drawn from a Poisson distribution with `n_labels` as its expected value, but samples are bounded (using rejection sampling) by `n_classes`, and must be nonzero if `allow_unlabeled` is False.

length [int, optional (default=50)] The sum of the features (number of words if documents) is drawn from a Poisson distribution with this expected value.

allow_unlabeled [bool, optional (default=True)] If `True`, some instances might not belong to any class.

sparse [bool, optional (default=False)] If `True`, return a sparse feature matrix

New in version 0.17: parameter to allow *sparse* output.

return_indicator [‘dense’ (default) | ‘sparse’ | False] If `dense` return `Y` in the dense binary indicator format. If `'sparse'` return `Y` in the sparse binary indicator format. `False` returns a list of lists of labels.

return_distributions [bool, optional (default=False)] If `True`, return the prior class probability and conditional probabilities of features given classes, from which the data was drawn.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

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

Y [array or sparse CSR matrix of shape [n_samples, n_classes]] The label sets.

p_c [array, shape [n_classes]] The probability of each class being drawn. Only returned if `return_distributions=True`.

p_w_c [array, shape [n_features, n_classes]] The probability of each feature being drawn given each class. Only returned if `return_distributions=True`.

Examples using `sklearn.datasets.make_multilabel_classification`

- *Multilabel classification*
- *Plot randomly generated multilabel dataset*

`sklearn.datasets.make_regression`

`sklearn.datasets.`**`make_regression`**(*n_samples=100*, *n_features=100*, *n_informative=10*, *n_targets=1*, *bias=0.0*, *effective_rank=None*, *tail_strength=0.5*, *noise=0.0*, *shuffle=True*, *coef=False*, *random_state=None*)

Generate a random regression problem.

The input set can either be well conditioned (by default) or have a low rank-fat tail singular profile. See `make_low_rank_matrix` for more details.

The output is generated by applying a (potentially biased) random linear regression model with `n_informative` nonzero regressors to the previously generated input and some gaussian centered noise with some adjustable scale.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=100)] The number of features.

n_informative [int, optional (default=10)] The number of informative features, i.e., the number of features used to build the linear model used to generate the output.

n_targets [int, optional (default=1)] The number of regression targets, i.e., the dimension of the y output vector associated with a sample. By default, the output is a scalar.

bias [float, optional (default=0.0)] The bias term in the underlying linear model.

effective_rank [int or None, optional (default=None)]

if not None: The approximate number of singular vectors required to explain most of the input data by linear combinations. Using this kind of singular spectrum in the input allows the generator to reproduce the correlations often observed in practice.

if None: The input set is well conditioned, centered and gaussian with unit variance.

tail_strength [float between 0.0 and 1.0, optional (default=0.5)] The relative importance of the fat noisy tail of the singular values profile if `effective_rank` is not None.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise applied to the output.

shuffle [boolean, optional (default=True)] Shuffle the samples and the features.

coef [boolean, optional (default=False)] If True, the coefficients of the underlying linear model are returned.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

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

y [array of shape [n_samples] or [n_samples, n_targets]] The output values.

coef [array of shape [n_features] or [n_features, n_targets], optional] The coefficient of the underlying linear model. It is returned only if coef is True.

Examples using `sklearn.datasets.make_regression`

- *Prediction Latency*
- *Effect of transforming the targets in regression model*
- *Plot Ridge coefficients as a function of the L2 regularization*
- *Robust linear model estimation using RANSAC*
- *Lasso on dense and sparse data*
- *HuberRegressor vs Ridge on dataset with strong outliers*

sklearn.datasets.make_s_curve

sklearn.datasets.**make_s_curve**(*n_samples=100*, *noise=0.0*, *random_state=None*)

Generate an S curve dataset.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of sample points on the S curve.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 3]] The points.

t [array of shape [n_samples]] The univariate position of the sample according to the main dimension of the points in the manifold.

Examples using `sklearn.datasets.make_s_curve`

- *t-SNE: The effect of various perplexity values on the shape*
- *Comparison of Manifold Learning methods*

`sklearn.datasets`.make_sparse_coded_signal

`sklearn.datasets.`**`make_sparse_coded_signal`**(*n_samples*, *n_components*, *n_features*, *n_nonzero_coefs*, *random_state=None*)

Generate a signal as a sparse combination of dictionary elements.

Returns a matrix Y = DX, such as D is (n_features, n_components), X is (n_components, n_samples) and each column of X has exactly n_nonzero_coefs non-zero elements.

Read more in the *User Guide*.

Parameters

n_samples [int] number of samples to generate

n_components [int,] number of components in the dictionary

n_features [int] number of features of the dataset to generate

n_nonzero_coefs [int] number of active (non-zero) coefficients in each sample

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

data [array of shape [n_features, n_samples]] The encoded signal (Y).

dictionary [array of shape [n_features, n_components]] The dictionary with normalized components (D).

code [array of shape [n_components, n_samples]] The sparse code such that each column of this matrix has exactly n_nonzero_coefs non-zero items (X).

Examples using `sklearn.datasets.make_sparse_coded_signal`

- *Orthogonal Matching Pursuit*

`sklearn.datasets`.make_sparse_spd_matrix

`sklearn.datasets.`**`make_sparse_spd_matrix`**(*dim=1*, *alpha=0.95*, *norm_diag=False*, *smallest_coef=0.1*, *largest_coef=0.9*, *random_state=None*)

Generate a sparse symmetric definite positive matrix.

Read more in the *User Guide*.

Parameters

dim [integer, optional (default=1)] The size of the random matrix to generate.

alpha [float between 0 and 1, optional (default=0.95)] The probability that a coefficient is zero (see notes). Larger values enforce more sparsity.

norm_diag [boolean, optional (default=False)] Whether to normalize the output matrix to make the leading diagonal elements all 1

smallest_coef [float between 0 and 1, optional (default=0.1)] The value of the smallest coefficient.

largest_coef [float between 0 and 1, optional (default=0.9)] The value of the largest coefficient.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

prec [sparse matrix of shape (dim, dim)] The generated matrix.

See also:

make_spd_matrix

Notes

The sparsity is actually imposed on the cholesky factor of the matrix. Thus alpha does not translate directly into the filling fraction of the matrix itself.

Examples using `sklearn.datasets.make_sparse_spd_matrix`

- *Sparse inverse covariance estimation*

`sklearn.datasets`.make_sparse_uncorrelated

`sklearn.datasets.`**`make_sparse_uncorrelated`**(*n_samples=100*, *n_features=10*, *random_state=None*)

Generate a random regression problem with sparse uncorrelated design

This dataset is described in Celeux et al [1]. as:

```
X ~ N(0, 1)
y(X) = X[:, 0] + 2 * X[:, 1] - 2 * X[:, 2] - 1.5 * X[:, 3]
```

Only the first 4 features are informative. The remaining features are useless.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=10)] The number of features.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

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

y [array of shape [n_samples]] The output values.

References

[1]

sklearn.datasets.make_spd_matrix

sklearn.datasets.**make_spd_matrix**(*n_dim*, *random_state=None*)
Generate a random symmetric, positive-definite matrix.

Read more in the *User Guide*.

Parameters

n_dim [int] The matrix dimension.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_dim, n_dim]] The random symmetric, positive-definite matrix.

See also:

make_sparse_spd_matrix

sklearn.datasets.make_swiss_roll

sklearn.datasets.**make_swiss_roll**(*n_samples=100*, *noise=0.0*, *random_state=None*)
Generate a swiss roll dataset.

Read more in the *User Guide*.

Parameters

n_samples [int, optional (default=100)] The number of sample points on the S curve.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise.

random_state [int, RandomState instance or None (default)] Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See *Glossary*.

Returns

X [array of shape [n_samples, 3]] The points.

t [array of shape [n_samples]] The univariate position of the sample according to the main dimension of the points in the manifold.

Notes

The algorithm is from Marsland [1].

References

[1]

Examples using `sklearn.datasets.make_swiss_roll`

- *Hierarchical clustering: structured vs unstructured ward*
- *Swiss Roll reduction with LLE*

6.9 `sklearn.decomposition`: Matrix Decomposition

The *sklearn.decomposition* module includes matrix decomposition algorithms, including among others PCA, NMF or ICA. Most of the algorithms of this module can be regarded as dimensionality reduction techniques.

User guide: See the *Decomposing signals in components (matrix factorization problems)* section for further details.

decomposition.DictionaryLearning([...])	Dictionary learning
decomposition.FactorAnalysis([n_components, ...])	Factor Analysis (FA)
decomposition.FastICA([n_components, ...])	FastICA: a fast algorithm for Independent Component Analysis.
decomposition.IncrementalPCA([n_components, ...])	Incremental principal components analysis (IPCA).
decomposition.KernelPCA([n_components, ...])	Kernel Principal component analysis (KPCA)
decomposition.LatentDirichletAllocation([...])	Latent Dirichlet Allocation with online variational Bayes algorithm
decomposition.MiniBatchDictionaryLearning([...])	Mini-batch dictionary learning
decomposition.MiniBatchSparsePCA([...])	Mini-batch Sparse Principal Components Analysis
decomposition.NMF([n_components, init, ...])	Non-Negative Matrix Factorization (NMF)
decomposition.PCA([n_components, copy, ...])	Principal component analysis (PCA)
decomposition.SparsePCA([n_components, ...])	Sparse Principal Components Analysis (SparsePCA)
decomposition.SparseCoder(dictionary[, ...])	Sparse coding
decomposition.TruncatedSVD([n_components, ...])	Dimensionality reduction using truncated SVD (aka LSA).

6.9.1 `sklearn.decomposition`.DictionaryLearning

class sklearn.decomposition.**DictionaryLearning**(*n_components=None*, *alpha=1*, *max_iter=1000*, *tol=1e-08*, *fit_algorithm='lars'*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *n_jobs=None*, *code_init=None*, *dict_init=None*, *verbose=False*, *split_sign=False*, *random_state=None*, *positive_code=False*, *positive_dict=False*)

Dictionary learning

Finds a dictionary (a set of atoms) that can best be used to represent data using a sparse code.

Solves the optimization problem:

```
(U^*,V^*) = argmin 0.5 || Y - U V ||_2^2 + alpha * || U ||_1
            (U,V)
            with || V_k ||_2 = 1 for all  0 <= k < n_components
```

Read more in the *User Guide*.

Parameters

n_components [int,] number of dictionary elements to extract

alpha [float,] sparsity controlling parameter

max_iter [int,] maximum number of iterations to perform

tol [float,] tolerance for numerical error

fit_algorithm [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

New in version 0.17: *cd* coordinate descent method to improve speed.

transform_algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}] Algorithm used to transform the data lars: uses the least angle regression method (linear_model.lars_path) lasso_lars: uses Lars to compute the Lasso solution lasso_cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). lasso_lars will be faster if the estimated components are sparse. omp: uses orthogonal matching pursuit to estimate the sparse solution threshold: squashes to zero all coefficients less than alpha from the projection `dictionary * X'`

New in version 0.17: *lasso_cd* coordinate descent method to improve speed.

transform_n_nonzero_coefs [int, `0.1 * n_features` by default] Number of nonzero coefficients to target in each column of the solution. This is only used by `algorithm='lars'` and `algorithm='omp'` and is overridden by `alpha` in the *Orthogonal Matching Pursuit (OMP)* case.

transform_alpha [float, 1. by default] If `algorithm='lasso_lars'` or `algorithm='lasso_cd'`, `alpha` is the penalty applied to the L1 norm. If `algorithm='threshold'`, `alpha` is the absolute value of the threshold below which coefficients will be squashed to zero. If `algorithm='omp'`, `alpha` is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides `n_nonzero_coefs`.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

code_init [array of shape (n_samples, n_components),] initial value for the code, for warm restart

dict_init [array of shape (n_components, n_features),] initial values for the dictionary, for warm restart

verbose [bool, optional (default: False)] To control the verbosity of the procedure.

split_sign [bool, False by default] Whether to split the sparse feature vector into the concatenation of its negative part and its positive part. This can improve the performance of downstream classifiers.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

positive_code [bool] Whether to enforce positivity when finding the code.

New in version 0.20.

positive_dict [bool] Whether to enforce positivity when finding the dictionary

New in version 0.20.

Attributes

components_ [array, [n_components, n_features]] dictionary atoms extracted from the data

error_ [array] vector of errors at each iteration

n_iter_ [int] Number of iterations run.

See also:

SparseCoder

MiniBatchDictionaryLearning

SparsePCA

MiniBatchSparsePCA

Notes

References:

J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009: Online dictionary learning for sparse coding (https://www.di.ens.fr/sierra/pdfs/icml09.pdf)

Methods

fit(self, X[, y])	Fit the model from data in X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Encode the data as a sparse combination of the dictionary atoms.

__init__ (*self*, *n_components=None*, *alpha=1*, *max_iter=1000*, *tol=1e-08*, *fit_algorithm='lars'*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *n_jobs=None*, *code_init=None*, *dict_init=None*, *verbose=False*, *split_sign=False*, *random_state=None*, *positive_code=False*, *positive_dict=False*)

fit (*self*, *X*, *y=None*)
Fit the model from data in X.

Parameters

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

y [Ignored]

Returns

self [object] Returns the object itself

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter `transform_algorithm`.

Parameters

X [array of shape (n_samples, n_features)] Test data to be transformed, must have the same number of features as the data used to train the model.

Returns

X_new [array, shape (n_samples, n_components)] Transformed data

6.9.2 sklearn.decomposition.FactorAnalysis

class sklearn.decomposition.**FactorAnalysis**(*n_components=None*, *tol=0.01*, *copy=True*, *max_iter=1000*, *noise_variance_init=None*, *svd_method='randomized'*, *iterated_power=3*, *random_state=0*)
Factor Analysis (FA)

A simple linear generative model with Gaussian latent variables.

The observations are assumed to be caused by a linear transformation of lower dimensional latent factors and added Gaussian noise. Without loss of generality the factors are distributed according to a Gaussian with zero mean and unit covariance. The noise is also zero mean and has an arbitrary diagonal covariance matrix.

If we would restrict the model further, by assuming that the Gaussian noise is even isotropic (all diagonal entries are the same) we would obtain `PPCA`.

FactorAnalysis performs a maximum likelihood estimate of the so-called `loading` matrix, the transformation of the latent variables to the observed ones, using expectation-maximization (EM).

Read more in the *User Guide*.

Parameters

n_components [int | None] Dimensionality of latent space, the number of components of `X` that are obtained after `transform`. If None, n_components is set to the number of features.

tol [float] Stopping tolerance for EM algorithm.

copy [bool] Whether to make a copy of X. If `False`, the input X gets overwritten during fitting.

max_iter [int] Maximum number of iterations.

noise_variance_init [None | array, shape=(n_features,)] The initial guess of the noise variance for each feature. If None, it defaults to np.ones(n_features)

svd_method [{'lapack', 'randomized'}] Which SVD method to use. If 'lapack' use standard SVD from scipy.linalg, if 'randomized' use fast `randomized_svd` function. Defaults to 'randomized'. For most applications 'randomized' will be sufficiently precise while providing significant speed gains. Accuracy can also be improved by setting higher values for `iterated_power`. If this is not sufficient, for maximum precision you should choose 'lapack'.

iterated_power [int, optional] Number of iterations for the power method. 3 by default. Only used if `svd_method` equals 'randomized'

random_state [int, RandomState instance or None, optional (default=0)] 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`. Only used when `svd_method` equals 'randomized'.

Attributes

components_ [array, [n_components, n_features]] Components with maximum variance.

loglike_ [list, [n_iterations]] The log likelihood at each iteration.

noise_variance_ [array, shape=(n_features,)] The estimated noise variance for each feature.

n_iter_ [int] Number of iterations run.

See also:

PCA Principal component analysis is also a latent linear variable model which however assumes equal noise variance for each feature. This extra assumption makes probabilistic PCA faster as it can be computed in closed form.

FastICA Independent component analysis, a latent variable model with non-Gaussian latent variables.

References

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.decomposition import FactorAnalysis
>>> X, _ = load_digits(return_X_y=True)
>>> transformer = FactorAnalysis(n_components=7, random_state=0)
>>> X_transformed = transformer.fit_transform(X)
>>> X_transformed.shape
(1797, 7)
```

Methods

fit(self, X[, y])	Fit the FactorAnalysis model to X using EM
fit_transform(self, X[, y])	Fit to data, then transform it.
get_covariance(self)	Compute data covariance with the FactorAnalysis model.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Compute data precision matrix with the FactorAnalysis model.
score(self, X[, y])	Compute the average log-likelihood of the samples
score_samples(self, X)	Compute the log-likelihood of each sample
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply dimensionality reduction to X using the model.

__init__(*self*, *n_components=None*, *tol=0.01*, *copy=True*, *max_iter=1000*, *noise_variance_init=None*, *svd_method='randomized'*, *iterated_power=3*, *random_state=0*)

fit(*self*, *X*, *y=None*)
Fit the FactorAnalysis model to X using EM

Parameters

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

y [Ignored]

Returns

self

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_covariance(*self*)
Compute data covariance with the FactorAnalysis model.

```
cov = components_.T * components_ + diag(noise_variance)
```

Returns

cov [array, shape (n_features, n_features)] Estimated covariance of data.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_precision(*self*)
Compute data precision matrix with the FactorAnalysis model.

Returns

precision [array, shape (n_features, n_features)] Estimated precision of data.

score(*self*, *X*, *y=None*)
Compute the average log-likelihood of the samples

Parameters

X [array, shape (n_samples, n_features)] The data

y [Ignored]

Returns

ll [float] Average log-likelihood of the samples under the current model

score_samples(*self*, *X*)
Compute the log-likelihood of each sample

Parameters

X [array, shape (n_samples, n_features)] The data

Returns

ll [array, shape (n_samples,)] Log-likelihood of each sample under the current model

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Apply dimensionality reduction to X using the model.

Compute the expected mean of the latent variables. See Barber, 21.2.33 (or Bishop, 12.66).

Parameters

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

Returns

X_new [array-like, shape (n_samples, n_components)] The latent variables of X.

Examples using `sklearn.decomposition.FactorAnalysis`

- *Model selection with Probabilistic PCA and Factor Analysis (FA)*
- *Faces dataset decompositions*

6.9.3 `sklearn.decomposition`.FastICA

class `sklearn.decomposition.`**`FastICA`**(*n_components=None*, *algorithm='parallel'*, *whiten=True*, *fun='logcosh'*, *fun_args=None*, *max_iter=200*, *tol=0.0001*, *w_init=None*, *random_state=None*)

FastICA: a fast algorithm for Independent Component Analysis.

Read more in the *User Guide*.

Parameters

n_components [int, optional] Number of components to use. If none is passed, all are used.

algorithm [{'parallel', 'deflation'}] Apply parallel or deflational algorithm for FastICA.

whiten [boolean, optional] If whiten is false, the data is already considered to be whitened, and no whitening is performed.

fun [string or function, optional. Default: 'logcosh'] The functional form of the G function used in the approximation to neg-entropy. Could be either 'logcosh', 'exp', or 'cube'. You can also provide your own function. It should return a tuple containing the value of the function, and of its derivative, in the point. Example:

def my_g(x): return x ** 3, (3 * x ** 2).mean(axis=-1)

fun_args [dictionary, optional] Arguments to send to the functional form. If empty and if fun='logcosh', fun_args will take value {'alpha' : 1.0}.

max_iter [int, optional] Maximum number of iterations during fit.

tol [float, optional] Tolerance on update at each iteration.

w_init [None of an (n_components, n_components) ndarray] The mixing matrix to be used to initialize the algorithm.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Attributes

components_ [2D array, shape (n_components, n_features)] The unmixing matrix.

mixing_ [array, shape (n_features, n_components)] The mixing matrix.

n_iter_ [int] If the algorithm is "deflation", n_iter is the maximum number of iterations run across all components. Else they are just the number of iterations taken to converge.

Notes

Implementation based on *A. Hyvarinen and E. Oja, Independent Component Analysis: Algorithms and Applications, Neural Networks, 13(4-5), 2000, pp. 411-430*

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.decomposition import FastICA
>>> X, _ = load_digits(return_X_y=True)
>>> transformer = FastICA(n_components=7,
...         random_state=0)
>>> X_transformed = transformer.fit_transform(X)
>>> X_transformed.shape
(1797, 7)
```

Methods

fit(self, X[, y])	Fit the model to X.
fit_transform(self, X[, y])	Fit the model and recover the sources from X.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X[, copy])	Transform the sources back to the mixed data (apply mixing matrix).
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, copy])	Recover the sources from X (apply the unmixing matrix).

__init__(*self*, *n_components=None*, *algorithm='parallel'*, *whiten=True*, *fun='logcosh'*, *fun_args=None*, *max_iter=200*, *tol=0.0001*, *w_init=None*, *random_state=None*)

fit(*self*, *X*, *y=None*)
Fit the model to X.

Parameters

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

y [Ignored]

Returns

self

fit_transform(*self*, *X*, *y=None*)
Fit the model and recover the sources from X.

Parameters

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

y [Ignored]

Returns

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

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*, *copy=True*)
Transform the sources back to the mixed data (apply mixing matrix).

Parameters

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

copy [bool (optional)] If False, data passed to fit are overwritten. Defaults to True.

Returns

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

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*, *copy=True*)
Recover the sources from X (apply the unmixing matrix).

Parameters

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

copy [bool (optional)] If False, data passed to fit are overwritten. Defaults to True.

Returns

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

Examples using `sklearn.decomposition.FastICA`

- *Blind source separation using FastICA*
- *FastICA on 2D point clouds*
- *Faces dataset decompositions*

6.9.4 sklearn.decomposition.IncrementalPCA

class sklearn.decomposition.**IncrementalPCA**(*n_components=None*, *whiten=False*, *copy=True*, *batch_size=None*)

Incremental principal components analysis (IPCA).

Linear dimensionality reduction using Singular Value Decomposition of the data, keeping only the most significant singular vectors to project the data to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

Depending on the size of the input data, this algorithm can be much more memory efficient than a PCA.

This algorithm has constant memory complexity, on the order of `batch_size`, enabling use of np.memmap files without loading the entire file into memory.

The computational overhead of each SVD is `O(batch_size * n_features ** 2)`, but only 2 * batch_size samples remain in memory at a time. There will be `n_samples / batch_size` SVD computations to get the principal components, versus 1 large SVD of complexity `O(n_samples * n_features ** 2)` for PCA.

Read more in the *User Guide*.

Parameters

n_components [int or None, (default=None)] Number of components to keep. If `n_components `` is ``None`, then `n_components` is set to `min(n_samples, n_features)`.

whiten [bool, optional] When True (False by default) the `components_` vectors are divided by `n_samples` times `components_` to ensure uncorrelated outputs with unit component-wise variances.

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

copy [bool, (default=True)] If False, X will be overwritten. `copy=False` can be used to save memory but is unsafe for general use.

batch_size [int or None, (default=None)] The number of samples to use for each batch. Only used when calling `fit`. If `batch_size` is `None`, then `batch_size` is inferred from the data and set to `5 * n_features`, to provide a balance between approximation accuracy and memory consumption.

Attributes

components_ [array, shape (n_components, n_features)] Components with maximum variance.

explained_variance_ [array, shape (n_components,)] Variance explained by each of the selected components.

explained_variance_ratio_ [array, shape (n_components,)] Percentage of variance explained by each of the selected components. If all components are stored, the sum of explained variances is equal to 1.0.

singular_values_ [array, shape (n_components,)] The singular values corresponding to each of the selected components. The singular values are equal to the 2-norms of the `n_components` variables in the lower-dimensional space.

mean_ [array, shape (n_features,)] Per-feature empirical mean, aggregate over calls to `partial_fit`.

var_ [array, shape (n_features,)] Per-feature empirical variance, aggregate over calls to `partial_fit`.

noise_variance_ [float] The estimated noise covariance following the Probabilistic PCA model from Tipping and Bishop 1999. See "Pattern Recognition and Machine Learning" by C. Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf.

n_components_ [int] The estimated number of components. Relevant when `n_components=None`.

n_samples_seen_ [int] The number of samples processed by the estimator. Will be reset on new calls to fit, but increments across `partial_fit` calls.

See also:

PCA

KernelPCA

SparsePCA

TruncatedSVD

Notes

Implements the incremental PCA model from: *D. Ross, J. Lim, R. Lin, M. Yang, Incremental Learning for Robust Visual Tracking, International Journal of Computer Vision, Volume 77, Issue 1-3, pp. 125-141, May 2008.* See https://www.cs.toronto.edu/~dross/ivt/RossLimLinYang_ijcv.pdf

This model is an extension of the Sequential Karhunen-Loeve Transform from: *A. Levy and M. Lindenbaum, Sequential Karhunen-Loeve Basis Extraction and its Application to Images, IEEE Transactions on Image Processing, Volume 9, Number 8, pp. 1371-1374, August 2000.* See https://www.cs.technion.ac.il/~mic/doc/skl-ip.pdf

We have specifically abstained from an optimization used by authors of both papers, a QR decomposition used in specific situations to reduce the algorithmic complexity of the SVD. The source for this technique is *Matrix Computations, Third Edition, G. Holub and C. Van Loan, Chapter 5, section 5.4.4, pp 252-253.*. This technique has been omitted because it is advantageous only when decomposing a matrix with `n_samples` (rows) >= 5/3 * `n_features` (columns), and hurts the readability of the implemented algorithm. This would be a good opportunity for future optimization, if it is deemed necessary.

References

D. Ross, J. Lim, R. Lin, M. Yang. Incremental Learning for Robust Visual Tracking, International Journal of Computer Vision, Volume 77, Issue 1-3, pp. 125-141, May 2008.

G. Golub and C. Van Loan. Matrix Computations, Third Edition, Chapter 5, Section 5.4.4, pp. 252-253.

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.decomposition import IncrementalPCA
>>> X, _ = load_digits(return_X_y=True)
>>> transformer = IncrementalPCA(n_components=7, batch_size=200)
>>> # either partially fit on smaller batches of data
>>> transformer.partial_fit(X[:100, :])
IncrementalPCA(batch_size=200, copy=True, n_components=7, whiten=False)
```

```
>>> # or let the fit function itself divide the data into batches
>>> X_transformed = transformer.fit_transform(X)
>>> X_transformed.shape
(1797, 7)
```

Methods

fit(self, X[, y])	Fit the model with X, using minibatches of size batch_size.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_covariance(self)	Compute data covariance with the generative model.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Compute data precision matrix with the generative model.
inverse_transform(self, X)	Transform data back to its original space.
partial_fit(self, X[, y, check_input])	Incremental fit with X.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply dimensionality reduction to X.

__init__(*self*, *n_components=None*, *whiten=False*, *copy=True*, *batch_size=None*)

fit(*self*, *X*, *y=None*)
Fit the model with X, using minibatches of size batch_size.

Parameters

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

y [Ignored]

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_covariance(*self*)
Compute data covariance with the generative model.

`cov = components_.T * S**2 * components_ + sigma2 * eye(n_features)`
where S**2 contains the explained variances, and sigma2 contains the noise variances.

Returns

cov [array, shape=(n_features, n_features)] Estimated covariance of data.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_precision(*self*)
Compute data precision matrix with the generative model.

Equals the inverse of the covariance but computed with the matrix inversion lemma for efficiency.

Returns

precision [array, shape=(n_features, n_features)] Estimated precision of data.

inverse_transform(*self*, *X*)
Transform data back to its original space.

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

Parameters

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

Returns

X_original array-like, shape (n_samples, n_features)

Notes

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

partial_fit(*self*, *X*, *y=None*, *check_input=True*)
Incremental fit with X. All of X is processed as a single batch.

Parameters

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

check_input [bool] Run check_array on X.

y [Ignored]

Returns

self [object] Returns the instance itself.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)

Apply dimensionality reduction to X.

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

Parameters

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

Returns

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

Examples

```
>>> import numpy as np
>>> from sklearn.decomposition import IncrementalPCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> ipca = IncrementalPCA(n_components=2, batch_size=3)
>>> ipca.fit(X)
IncrementalPCA(batch_size=3, copy=True, n_components=2, whiten=False)
>>> ipca.transform(X)
```

Examples using `sklearn.decomposition.IncrementalPCA`

- *Incremental PCA*

6.9.5 sklearn.decomposition.KernelPCA

class sklearn.decomposition.**KernelPCA**(*n_components=None*, *kernel='linear'*, *gamma=None*, *degree=3*, *coef0=1*, *kernel_params=None*, *alpha=1.0*, *fit_inverse_transform=False*, *eigen_solver='auto'*, *tol=0*, *max_iter=None*, *remove_zero_eig=False*, *random_state=None*, *copy_X=True*, *n_jobs=None*)

Kernel Principal component analysis (KPCA)

Non-linear dimensionality reduction through the use of kernels (see *Pairwise metrics, Affinities and Kernels*).

Read more in the *User Guide*.

Parameters

n_components [int, default=None] Number of components. If None, all non-zero components are kept.

kernel ["linear" | "poly" | "rbf" | "sigmoid" | "cosine" | "precomputed"] Kernel. Default="linear".

gamma [float, default=1/n_features] Kernel coefficient for rbf, poly and sigmoid kernels. Ignored by other kernels.

degree [int, default=3] Degree for poly kernels. Ignored by other kernels.

coef0 [float, default=1] Independent term in poly and sigmoid kernels. Ignored by other kernels.

kernel_params [mapping of string to any, default=None] Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.

alpha [int, default=1.0] Hyperparameter of the ridge regression that learns the inverse transform (when fit_inverse_transform=True).

fit_inverse_transform [bool, default=False] Learn the inverse transform for non-precomputed kernels. (i.e. learn to find the pre-image of a point)

eigen_solver [string ['auto'|'dense'|'arpack'], default='auto'] Select eigensolver to use. If n_components is much less than the number of training samples, arpack may be more efficient than the dense eigensolver.

tol [float, default=0] Convergence tolerance for arpack. If 0, optimal value will be chosen by arpack.

max_iter [int, default=None] Maximum number of iterations for arpack. If None, optimal value will be chosen by arpack.

remove_zero_eig [boolean, default=False] If True, then all components with zero eigenvalues are removed, so that the number of components in the output may be < n_components (and sometimes even zero due to numerical instability). When n_components is None, this parameter is ignored and components with zero eigenvalues are removed regardless.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `eigen_solver` == 'arpack'.

New in version 0.18.

copy_X [boolean, default=True] If True, input X is copied and stored by the model in the `X_fit_` attribute. If no further changes will be done to X, setting `copy_X=False` saves memory by storing a reference.

New in version 0.18.

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

New in version 0.18.

Attributes

lambdas_ [array, (n_components,)] Eigenvalues of the centered kernel matrix in decreasing order. If *n_components* and `remove_zero_eig` are not set, then all values are stored.

alphas_ [array, (n_samples, n_components)] Eigenvectors of the centered kernel matrix. If *n_components* and `remove_zero_eig` are not set, then all components are stored.

dual_coef_ [array, (n_samples, n_features)] Inverse transform matrix. Only available when `fit_inverse_transform` is True.

X_transformed_fit_ [array, (n_samples, n_components)] Projection of the fitted data on the kernel principal components. Only available when `fit_inverse_transform` is True.

X_fit_ [(n_samples, n_features)] The data used to fit the model. If `copy_X=False`, then `X_fit_` is a reference. This attribute is used for the calls to transform.

References

Kernel PCA was introduced in: Bernhard Schoelkopf, Alexander J. Smola, and Klaus-Robert Mueller. 1999. Kernel principal component analysis. In Advances in kernel methods, MIT Press, Cambridge, MA, USA 327-352.

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.decomposition import KernelPCA
>>> X, _ = load_digits(return_X_y=True)
>>> transformer = KernelPCA(n_components=7, kernel='linear')
>>> X_transformed = transformer.fit_transform(X)
>>> X_transformed.shape
(1797, 7)
```

Methods

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

__init__(*self*, *n_components=None*, *kernel='linear'*, *gamma=None*, *degree=3*, *coef0=1*, *kernel_params=None*, *alpha=1.0*, *fit_inverse_transform=False*, *eigen_solver='auto'*, *tol=0*, *max_iter=None*, *remove_zero_eig=False*, *random_state=None*, *copy_X=True*, *n_jobs=None*)

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

Fit the model from data in X.

Parameters

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

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *X*, *y=None*, ***params*)

Fit the model from data in X and transform X.

Parameters

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

Returns

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

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*)
Transform X back to original space.

Parameters

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

Returns

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

References

"Learning to Find Pre-Images", G BakIr et al, 2004.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform X.

Parameters

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

Returns

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

Examples using `sklearn.decomposition.KernelPCA`

- *Kernel PCA*

6.9.6 sklearn.decomposition.LatentDirichletAllocation

class sklearn.decomposition.**LatentDirichletAllocation**(*n_components=10*, *doc_topic_prior=None*, *topic_word_prior=None*, *learning_method='batch'*, *learning_decay=0.7*, *learning_offset=10.0*, *max_iter=10*, *batch_size=128*, *evaluate_every=-1*, *total_samples=1000000.0*, *perp_tol=0.1*, *mean_change_tol=0.001*, *max_doc_update_iter=100*, *n_jobs=None*, *verbose=0*, *random_state=None*)

Latent Dirichlet Allocation with online variational Bayes algorithm

New in version 0.17.

Read more in the *User Guide*.

Parameters

n_components [int, optional (default=10)] Number of topics.

doc_topic_prior [float, optional (default=None)] Prior of document topic distribution *theta*. If the value is None, defaults to `1 / n_components`. In [1]_, this is called `alpha`.

topic_word_prior [float, optional (default=None)] Prior of topic word distribution `beta`. If the value is None, defaults to `1 / n_components`. In [1]_, this is called `eta`.

learning_method ['batch' | 'online', default='batch'] Method used to update `_component`. Only used in *fit* method. In general, if the data size is large, the online update will be much faster than the batch update.

Valid options:

```
'batch': Batch variational Bayes method. Use all training data in
    each EM update.
    Old `components_` will be overwritten in each iteration.
'online': Online variational Bayes method. In each EM update, use
    mini-batch of training data to update the ``components_``
    variable incrementally. The learning rate is controlled by the
    ``learning_decay`` and the ``learning_offset`` parameters.
```

Changed in version 0.20: The default learning method is now `"batch"`.

learning_decay [float, optional (default=0.7)] It is a parameter that control learning rate in the online learning method. The value should be set between (0.5, 1.0] to guarantee asymptotic convergence. When the value is 0.0 and batch_size is `n_samples`, the update method is same as batch learning. In the literature, this is called kappa.

learning_offset [float, optional (default=10.)] A (positive) parameter that downweights early iterations in online learning. It should be greater than 1.0. In the literature, this is called tau_0.

max_iter [integer, optional (default=10)] The maximum number of iterations.

batch_size [int, optional (default=128)] Number of documents to use in each EM iteration. Only used in online learning.

evaluate_every [int, optional (default=0)] How often to evaluate perplexity. Only used in *fit* method. set it to 0 or negative number to not evalute perplexity in training at all. Evaluating perplexity can help you check convergence in training process, but it will also increase total training time. Evaluating perplexity in every iteration might increase training time up to two-fold.

total_samples [int, optional (default=1e6)] Total number of documents. Only used in the *partial_fit* method.

perp_tol [float, optional (default=1e-1)] Perplexity tolerance in batch learning. Only used when `evaluate_every` is greater than 0.

mean_change_tol [float, optional (default=1e-3)] Stopping tolerance for updating document topic distribution in E-step.

max_doc_update_iter [int (default=100)] Max number of iterations for updating document topic distribution in the E-step.

n_jobs [int or None, optional (default=None)] The number of jobs to use in the E-step. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [int, optional (default=0)] Verbosity level.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Attributes

components_ [array, [n_components, n_features]] Variational parameters for topic word distribution. Since the complete conditional for topic word distribution is a Dirichlet, `components_[i, j]` can be viewed as pseudocount that represents the number of times word `j` was assigned to topic `i`. It can also be viewed as distribution over the words for each topic after normalization: `model.components_ / model.components_.sum(axis=1)[:, np.newaxis]`.

n_batch_iter_ [int] Number of iterations of the EM step.

n_iter_ [int] Number of passes over the dataset.

References

[1] "Online Learning for Latent Dirichlet Allocation", Matthew D. Hoffman, David M. Blei, Francis Bach, 2010

[2] "Stochastic Variational Inference", Matthew D. Hoffman, David M. Blei, Chong Wang, John Paisley, 2013

[3] Matthew D. Hoffman's onlineldavb code. Link: https://github.com/blei-lab/onlineldavb

Examples

```
>>> from sklearn.decomposition import LatentDirichletAllocation
>>> from sklearn.datasets import make_multilabel_classification
>>> # This produces a feature matrix of token counts, similar to what
>>> # CountVectorizer would produce on text.
```

```
>>> X, _ = make_multilabel_classification(random_state=0)
>>> lda = LatentDirichletAllocation(n_components=5,
...     random_state=0)
>>> lda.fit(X)
LatentDirichletAllocation(...)
>>> # get topics for some given samples:
>>> lda.transform(X[-2:])
array([[0.00360392, 0.25499205, 0.0036211 , 0.64236448, 0.09541846],
       [0.15297572, 0.00362644, 0.44412786, 0.39568399, 0.003586  ]])
```

Methods

fit(self, X[, y])	Learn model for the data X with variational Bayes method.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X[, y])	Online VB with Mini-Batch update.
perplexity(self, X[, sub_sampling])	Calculate approximate perplexity for data X.
score(self, X[, y])	Calculate approximate log-likelihood as score.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform data X according to the fitted model.

__init__(*self*, *n_components=10*, *doc_topic_prior=None*, *topic_word_prior=None*, *learning_method='batch'*, *learning_decay=0.7*, *learning_offset=10.0*, *max_iter=10*, *batch_size=128*, *evaluate_every=-1*, *total_samples=1000000.0*, *perp_tol=0.1*, *mean_change_tol=0.001*, *max_doc_update_iter=100*, *n_jobs=None*, *verbose=0*, *random_state=None*)

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

Learn model for the data X with variational Bayes method.

When `learning_method` is 'online', use mini-batch update. Otherwise, use batch update.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Document word matrix.

y [Ignored]

Returns

self

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X*, *y=None*)
Online VB with Mini-Batch update.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Document word matrix.

y [Ignored]

Returns

self

perplexity(*self*, *X*, *sub_sampling=False*)
Calculate approximate perplexity for data X.

Perplexity is defined as exp(-1. * log-likelihood per word)

Changed in version 0.19: *doc_topic_distr* argument has been deprecated and is ignored because user no longer has access to unnormalized distribution

Parameters

X [array-like or sparse matrix, [n_samples, n_features]] Document word matrix.

sub_sampling [bool] Do sub-sampling or not.

Returns

score [float] Perplexity score.

score(*self*, *X*, *y=None*)
Calculate approximate log-likelihood as score.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Document word matrix.

y [Ignored]

Returns

score [float] Use approximate bound as score.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform data X according to the fitted model.

Changed in version 0.18: *doc_topic_distr* is now normalized

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Document word matrix.

Returns

doc_topic_distr [shape=(n_samples, n_components)] Document topic distribution for X.

Examples using `sklearn.decomposition.LatentDirichletAllocation`

- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*

6.9.7 sklearn.decomposition.MiniBatchDictionaryLearning

class sklearn.decomposition.**MiniBatchDictionaryLearning**(*n_components=None*, *alpha=1*, *n_iter=1000*, *fit_algorithm='lars'*, *n_jobs=None*, *batch_size=3*, *shuffle=True*, *dict_init=None*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *verbose=False*, *split_sign=False*, *random_state=None*, *positive_code=False*, *positive_dict=False*)

Mini-batch dictionary learning

Finds a dictionary (a set of atoms) that can best be used to represent data using a sparse code.

Solves the optimization problem:

```
(U^*,V^*) = argmin 0.5 || Y - U V ||_2^2 + alpha * || U ||_1
             (U,V)
             with || V_k ||_2 = 1 for all  0 <= k < n_components
```

Read more in the *User Guide*.

Parameters

n_components [int,] number of dictionary elements to extract

alpha [float,] sparsity controlling parameter

n_iter [int,] total number of iterations to perform

fit_algorithm [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

batch_size [int,] number of samples in each mini-batch

shuffle [bool,] whether to shuffle the samples before forming batches

dict_init [array of shape (n_components, n_features),] initial value of the dictionary for warm restart scenarios

transform_algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}] Algorithm used to transform the data. lars: uses the least angle regression method (linear_model.lars_path) lasso_lars: uses Lars to compute the Lasso solution lasso_cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). lasso_lars will be faster if the estimated components are sparse. omp: uses orthogonal matching pursuit to estimate the sparse solution threshold: squashes to zero all coefficients less than alpha from the projection dictionary * X'

transform_n_nonzero_coefs [int, `0.1 * n_features` by default] Number of nonzero coefficients to target in each column of the solution. This is only used by `algorithm='lars'` and `algorithm='omp'` and is overridden by `alpha` in the *Orthogonal Matching Pursuit (OMP)* case.

transform_alpha [float, 1. by default] If `algorithm='lasso_lars'` or `algorithm='lasso_cd'`, `alpha` is the penalty applied to the L1 norm. If `algorithm='threshold'`, `alpha` is the absolute value of the threshold below which coefficients will be squashed to zero. If `algorithm='omp'`, `alpha` is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides `n_nonzero_coefs`.

verbose [bool, optional (default: False)] To control the verbosity of the procedure.

split_sign [bool, False by default] Whether to split the sparse feature vector into the concatenation of its negative part and its positive part. This can improve the performance of downstream classifiers.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

positive_code [bool] Whether to enforce positivity when finding the code.

New in version 0.20.

positive_dict [bool] Whether to enforce positivity when finding the dictionary.

New in version 0.20.

Attributes

components_ [array, [n_components, n_features]] components extracted from the data

inner_stats_ [tuple of (A, B) ndarrays] Internal sufficient statistics that are kept by the algorithm. Keeping them is useful in online settings, to avoid loosing the history of the evolution, but they shouldn't have any use for the end user. A (n_components, n_components) is the dictionary covariance matrix. B (n_features, n_components) is the data approximation matrix

n_iter_ [int] Number of iterations run.

See also:

SparseCoder

DictionaryLearning

SparsePCA

MiniBatchSparsePCA

Notes

References:

J. Mairal, F. Bach, J. Ponce, G. Sapiro, 2009: Online dictionary learning for sparse coding (https://www.di.ens.fr/sierra/pdfs/icml09.pdf)

Methods

fit(self, X[, y])	Fit the model from data in X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X[, y, iter_offset])	Updates the model using the data in X as a mini-batch.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Encode the data as a sparse combination of the dictionary atoms.

__init__(*self*, *n_components=None*, *alpha=1*, *n_iter=1000*, *fit_algorithm='lars'*, *n_jobs=None*, *batch_size=3*, *shuffle=True*, *dict_init=None*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *verbose=False*, *split_sign=False*, *random_state=None*, *positive_code=False*, *positive_dict=False*)

fit(*self*, *X*, *y=None*)
Fit the model from data in X.

Parameters

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

y [Ignored]

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X*, *y=None*, *iter_offset=None*)
Updates the model using the data in X as a mini-batch.

Parameters

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

y [Ignored]

iter_offset [integer, optional] The number of iteration on data batches that has been performed before this call to partial_fit. This is optional: if no number is passed, the memory of the object is used.

Returns

self [object] Returns the instance itself.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter `transform_algorithm`.

Parameters

X [array of shape (n_samples, n_features)] Test data to be transformed, must have the same number of features as the data used to train the model.

Returns

X_new [array, shape (n_samples, n_components)] Transformed data

Examples using `sklearn.decomposition.MiniBatchDictionaryLearning`

- *Image denoising using dictionary learning*
- *Faces dataset decompositions*

6.9.8 sklearn.decomposition.MiniBatchSparsePCA

class sklearn.decomposition.**MiniBatchSparsePCA**(*n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *n_iter=100*, *callback=None*, *batch_size=3*, *verbose=False*, *shuffle=True*, *n_jobs=None*, *method='lars'*, *random_state=None*, *normalize_components=False*)

Mini-batch Sparse Principal Components Analysis

Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha.

Read more in the *User Guide*.

Parameters

n_components [int,] number of sparse atoms to extract

alpha [int,] Sparsity controlling parameter. Higher values lead to sparser components.

ridge_alpha [float,] Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.

n_iter [int,] number of iterations to perform for each mini batch

callback [callable or None, optional (default: None)] callable that gets invoked every five iterations

batch_size [int,] the number of features to take in each mini batch

verbose [int] Controls the verbosity; the higher, the more messages. Defaults to 0.

shuffle [boolean,] whether to shuffle the data before splitting it in batches

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

method [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

normalize_components [boolean, optional (default=False)]

- if False, use a version of Sparse PCA without components normalization and without data centering. This is likely a bug and even though it's the default for backward compatibility, this should not be used.
- if True, use a version of Sparse PCA with components normalization and data centering.

New in version 0.20.

Deprecated since version 0.22: `normalize_components` was added and set to `False` for backward compatibility. It would be set to `True` from 0.22 onwards.

Attributes

components_ [array, [n_components, n_features]] Sparse components extracted from the data.

n_iter_ [int] Number of iterations run.

mean_ [array, shape (n_features,)] Per-feature empirical mean, estimated from the training set. Equal to `X.mean(axis=0)`.

See also:

PCA

SparsePCA

DictionaryLearning

Examples

```
>>> import numpy as np
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.decomposition import MiniBatchSparsePCA
>>> X, _ = make_friedman1(n_samples=200, n_features=30, random_state=0)
>>> transformer = MiniBatchSparsePCA(n_components=5,
...         batch_size=50,
...         normalize_components=True,
...         random_state=0)
>>> transformer.fit(X)
MiniBatchSparsePCA(...)
>>> X_transformed = transformer.transform(X)
>>> X_transformed.shape
(200, 5)
>>> # most values in the components_ are zero (sparsity)
>>> np.mean(transformer.components_ == 0)
0.94
```

Methods

fit(self, X[, y])	Fit the model from data in X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Least Squares projection of the data onto the sparse components.

__init__(*self*, *n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *n_iter=100*, *callback=None*, *batch_size=3*, *verbose=False*, *shuffle=True*, *n_jobs=None*, *method='lars'*, *random_state=None*, *normalize_components=False*)

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

Fit the model from data in X.

Parameters

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

y [Ignored]

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Least Squares projection of the data onto the sparse components.

To avoid instability issues in case the system is under-determined, regularization can be applied (Ridge regression) via the `ridge_alpha` parameter.

Note that Sparse PCA components orthogonality is not enforced as in PCA hence one cannot use a simple linear projection.

Parameters

X [array of shape (n_samples, n_features)] Test data to be transformed, must have the same number of features as the data used to train the model.

Returns

X_new array, shape (n_samples, n_components) Transformed data.

Examples using `sklearn.decomposition.MiniBatchSparsePCA`

- *Faces dataset decompositions*

6.9.9 sklearn.decomposition.NMF

class sklearn.decomposition.**NMF**(*n_components=None*, *init=None*, *solver='cd'*, *beta_loss='frobenius'*, *tol=0.0001*, *max_iter=200*, *random_state=None*, *alpha=0.0*, *l1_ratio=0.0*, *verbose=0*, *shuffle=False*)

Non-Negative Matrix Factorization (NMF)

Find two non-negative matrices (W, H) whose product approximates the non- negative matrix X. This factorization can be used for example for dimensionality reduction, source separation or topic extraction.

The objective function is:

```
0.5 * ||X - WH||_Fro^2
+ alpha * l1_ratio * ||vec(W)||_1
+ alpha * l1_ratio * ||vec(H)||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
+ 0.5 * alpha * (1 - l1_ratio) * ||H||_Fro^2
```

Where:

```
||A||_Fro^2 = \sum_{i,j} A_{ij}^2 (Frobenius norm)
||vec(A)||_1 = \sum_{i,j} abs(A_{ij}) (Elementwise L1 norm)
```

For multiplicative-update ('mu') solver, the Frobenius norm (0.5 * ||X - WH||_Fro^2) can be changed into another beta-divergence loss, by changing the beta_loss parameter.

The objective function is minimized with an alternating minimization of W and H.

Read more in the *User Guide*.

Parameters

n_components [int or None] Number of components, if n_components is not set all features are kept.

init [None | 'random' | 'nndsvd' | 'nndsvda' | 'nndsvdar' | 'custom'] Method used to initialize the procedure. Default: None. Valid options:

- **None: 'nndsvd' if n_components <= min(n_samples, n_features),** otherwise random.
- **'random': non-negative random matrices, scaled with:** sqrt(X.mean() / n_components)
- **'nndsvd': Nonnegative Double Singular Value Decomposition (NNDSVD)** initialization (better for sparseness)
- **'nndsvda': NNDSVD with zeros filled with the average of X** (better when sparsity is not desired)
- **'nndsvdar': NNDSVD with zeros filled with small random values** (generally faster, less accurate alternative to NNDSVDa for when sparsity is not desired)
- 'custom': use custom matrices W and H

solver ['cd' | 'mu'] Numerical solver to use: 'cd' is a Coordinate Descent solver. 'mu' is a Multiplicative Update solver.

New in version 0.17: Coordinate Descent solver.

New in version 0.19: Multiplicative Update solver.

beta_loss [float or string, default 'frobenius'] String must be in {'frobenius', 'kullback-leibler', 'itakura-saito'}. Beta divergence to be minimized, measuring the distance between X and the dot product WH. Note that values different from 'frobenius' (or 2) and 'kullback-leibler' (or 1) lead to significantly slower fits. Note that for beta_loss <= 0 (or 'itakura-saito'), the input matrix X cannot contain zeros. Used only in 'mu' solver.

New in version 0.19.

tol [float, default: 1e-4] Tolerance of the stopping condition.

max_iter [integer, default: 200] Maximum number of iterations before timing out.

random_state [int, RandomState instance or None, optional, default: None] 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`.

alpha [double, default: 0.] Constant that multiplies the regularization terms. Set it to zero to have no regularization.

New in version 0.17: *alpha* used in the Coordinate Descent solver.

l1_ratio [double, default: 0.] The regularization mixing parameter, with 0 <= l1_ratio <= 1. For l1_ratio = 0 the penalty is an elementwise L2 penalty (aka Frobenius Norm). For l1_ratio = 1 it is an elementwise L1 penalty. For 0 < l1_ratio < 1, the penalty is a combination of L1 and L2.

New in version 0.17: Regularization parameter *l1_ratio* used in the Coordinate Descent solver.

verbose [bool, default=False] Whether to be verbose.

shuffle [boolean, default: False] If true, randomize the order of coordinates in the CD solver.

New in version 0.17: *shuffle* parameter used in the Coordinate Descent solver.

Attributes

components_ [array, [n_components, n_features]] Factorization matrix, sometimes called 'dictionary'.

reconstruction_err_ [number] Frobenius norm of the matrix difference, or beta-divergence, between the training data `X` and the reconstructed data `WH` from the fitted model.

n_iter_ [int] Actual number of iterations.

References

Cichocki, Andrzej, and P. H. A. N. Anh-Huy. "Fast local algorithms for large scale nonnegative matrix and tensor factorizations." IEICE transactions on fundamentals of electronics, communications and computer sciences 92.3: 708-721, 2009.

Fevotte, C., & Idier, J. (2011). Algorithms for nonnegative matrix factorization with the beta-divergence. Neural Computation, 23(9).

Examples

```
>>> import numpy as np
>>> X = np.array([[1, 1], [2, 1], [3, 1.2], [4, 1], [5, 0.8], [6, 1]])
>>> from sklearn.decomposition import NMF
>>> model = NMF(n_components=2, init='random', random_state=0)
>>> W = model.fit_transform(X)
>>> H = model.components_
```

Methods

fit(self, X[, y])	Learn a NMF model for the data X.
fit_transform(self, X[, y, W, H])	Learn a NMF model for the data X and returns the transformed data.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, W)	Transform data back to its original space.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform the data X according to the fitted NMF model

__init__(*self*, *n_components=None*, *init=None*, *solver='cd'*, *beta_loss='frobenius'*, *tol=0.0001*, *max_iter=200*, *random_state=None*, *alpha=0.0*, *l1_ratio=0.0*, *verbose=0*, *shuffle=False*)

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

Learn a NMF model for the data X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Data matrix to be decomposed

y [Ignored]

Returns

self

fit_transform(*self*, *X*, *y=None*, *W=None*, *H=None*)

Learn a NMF model for the data X and returns the transformed data.

This is more efficient than calling fit followed by transform.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Data matrix to be decomposed

y [Ignored]

W [array-like, shape (n_samples, n_components)] If init='custom', it is used as initial guess for the solution.

H [array-like, shape (n_components, n_features)] If init='custom', it is used as initial guess for the solution.

Returns

W [array, shape (n_samples, n_components)] Transformed data.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *W*)
Transform data back to its original space.

Parameters

W [{array-like, sparse matrix}, shape (n_samples, n_components)] Transformed data matrix

Returns

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Data matrix of original shape

New in version 0.18: ..

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform the data X according to the fitted NMF model

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Data matrix to be transformed by the model

Returns

W [array, shape (n_samples, n_components)] Transformed data

Examples using `sklearn.decomposition.NMF`

- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *Faces dataset decompositions*

6.9.10 sklearn.decomposition.PCA

class sklearn.decomposition.**PCA**(*n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*)
Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko et al. 2009, depending on the shape of the input data and the number of components to extract.

It can also use the scipy.sparse.linalg ARPACK implementation of the truncated SVD.

Notice that this class does not support sparse input. See *TruncatedSVD* for an alternative with sparse data.

Read more in the *User Guide*.

Parameters

n_components [int, float, None or string] Number of components to keep. if n_components is not set all components are kept:

```
n_components == min(n_samples, n_features)
```

If `n_components == 'mle'` and `svd_solver == 'full'`, Minka's MLE is used to guess the dimension. Use of `n_components == 'mle'` will interpret `svd_solver == 'auto'` as `svd_solver == 'full'`.

If `0 < n_components < 1` and `svd_solver == 'full'`, select the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by n_components.

If `svd_solver == 'arpack'`, the number of components must be strictly less than the minimum of n_features and n_samples.

Hence, the None case results in:

```
n_components == min(n_samples, n_features) - 1
```

copy [bool (default True)] If False, data passed to fit are overwritten and running fit(X).transform(X) will not yield the expected results, use fit_transform(X) instead.

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

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

svd_solver [string {'auto', 'full', 'arpack', 'randomized'}]

auto : the solver is selected by a default policy based on `X.shape` and *n_components*: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient 'randomized' method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

full : run exact full SVD calling the standard LAPACK solver via `scipy.linalg.svd` and select the components by postprocessing

arpack : run SVD truncated to n_components calling ARPACK solver via `scipy.sparse.linalg.svds`. It requires strictly 0 < n_components < min(X.shape)

randomized : run randomized SVD by the method of Halko et al.

New in version 0.18.0.

tol [float >= 0, optional (default .0)] Tolerance for singular values computed by svd_solver == 'arpack'.

New in version 0.18.0.

iterated_power [int >= 0, or 'auto', (default 'auto')] Number of iterations for the power method computed by svd_solver == 'randomized'.

New in version 0.18.0.

random_state [int, RandomState instance or None, optional (default None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `svd_solver` == 'arpack' or 'randomized'.

New in version 0.18.0.

Attributes

components_ [array, shape (n_components, n_features)] Principal axes in feature space, representing the directions of maximum variance in the data. The components are sorted by `explained_variance_`.

explained_variance_ [array, shape (n_components,)] The amount of variance explained by each of the selected components.

Equal to n_components largest eigenvalues of the covariance matrix of X.

New in version 0.18.

explained_variance_ratio_ [array, shape (n_components,)] Percentage of variance explained by each of the selected components.

If `n_components` is not set then all components are stored and the sum of the ratios is equal to 1.0.

singular_values_ [array, shape (n_components,)] The singular values corresponding to each of the selected components. The singular values are equal to the 2-norms of the `n_components` variables in the lower-dimensional space.

New in version 0.19.

mean_ [array, shape (n_features,)] Per-feature empirical mean, estimated from the training set.

Equal to `X.mean(axis=0)`.

n_components_ [int] The estimated number of components. When n_components is set to 'mle' or a number between 0 and 1 (with svd_solver == 'full') this number is estimated from input data. Otherwise it equals the parameter n_components, or the lesser value of n_features and n_samples if n_components is None.

noise_variance_ [float] The estimated noise covariance following the Probabilistic PCA model from Tipping and Bishop 1999. See "Pattern Recognition and Machine Learning" by C. Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf. It is required to compute the estimated data covariance and score samples.

Equal to the average of (min(n_features, n_samples) - n_components) smallest eigenvalues of the covariance matrix of X.

See also:

KernelPCA

SparsePCA

TruncatedSVD

IncrementalPCA

References

For n_components == 'mle', this class uses the method of *Minka, T. P. "Automatic choice of dimensionality for PCA". In NIPS, pp. 598-604*

Implements the probabilistic PCA model from: Tipping, M. E., and Bishop, C. M. (1999). "Probabilistic principal component analysis". Journal of the Royal Statistical Society: Series B (Statistical Methodology), 61(3), 611-622. via the score and score_samples methods. See http://www.miketipping.com/papers/met-mppca.pdf

For svd_solver == 'arpack', refer to `scipy.sparse.linalg.svds`.

For svd_solver == 'randomized', see: *Halko, N., Martinsson, P. G., and Tropp, J. A. (2011). "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions". SIAM review, 53(2), 217-288.* and also *Martinsson, P. G., Rokhlin, V., and Tygert, M. (2011). "A randomized algorithm for the decomposition of matrices". Applied and Computational Harmonic Analysis, 30(1), 47-68.*

Examples

```
>>> import numpy as np
>>> from sklearn.decomposition import PCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> pca = PCA(n_components=2)
>>> pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
>>> print(pca.explained_variance_ratio_)
[0.9924... 0.0075...]
>>> print(pca.singular_values_)
[6.30061... 0.54980...]
```

```
>>> pca = PCA(n_components=2, svd_solver='full')
>>> pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
  svd_solver='full', tol=0.0, whiten=False)
>>> print(pca.explained_variance_ratio_)
[0.9924... 0.00755...]
>>> print(pca.singular_values_)
[6.30061... 0.54980...]
```

```
>>> pca = PCA(n_components=1, svd_solver='arpack')
>>> pca.fit(X)
PCA(copy=True, iterated_power='auto', n_components=1, random_state=None,
  svd_solver='arpack', tol=0.0, whiten=False)
>>> print(pca.explained_variance_ratio_)
[0.99244...]
>>> print(pca.singular_values_)
[6.30061...]
```

Methods

fit(self, X[, y])	Fit the model with X.

Continued on next page

Table 6.59 – continued from previous page

fit_transform(self, X[, y])	Fit the model with X and apply the dimensionality reduction on X.
get_covariance(self)	Compute data covariance with the generative model.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Compute data precision matrix with the generative model.
inverse_transform(self, X)	Transform data back to its original space.
score(self, X[, y])	Return the average log-likelihood of all samples.
score_samples(self, X)	Return the log-likelihood of each sample.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply dimensionality reduction to X.

__init__(*self*, *n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*)

fit(*self*, *X*, *y=None*)
Fit the model with X.

Parameters

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

y [Ignored]

Returns

self [object] Returns the instance itself.

fit_transform(*self*, *X*, *y=None*)
Fit the model with X and apply the dimensionality reduction on X.

Parameters

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

y [Ignored]

Returns

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

get_covariance(*self*)
Compute data covariance with the generative model.

`cov = components_.T * S**2 * components_ + sigma2 * eye(n_features)`
where S**2 contains the explained variances, and sigma2 contains the noise variances.

Returns

cov [array, shape=(n_features, n_features)] Estimated covariance of data.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_precision(*self*)
Compute data precision matrix with the generative model.

Equals the inverse of the covariance but computed with the matrix inversion lemma for efficiency.

Returns

precision [array, shape=(n_features, n_features)] Estimated precision of data.

inverse_transform(*self*, *X*)
Transform data back to its original space.

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

Parameters

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

Returns

X_original array-like, shape (n_samples, n_features)

Notes

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

score(*self*, *X*, *y=None*)
Return the average log-likelihood of all samples.

See. "Pattern Recognition and Machine Learning" by C. Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf

Parameters

X [array, shape(n_samples, n_features)] The data.

y [Ignored]

Returns

ll [float] Average log-likelihood of the samples under the current model

score_samples(*self*, *X*)
Return the log-likelihood of each sample.

See. "Pattern Recognition and Machine Learning" by C. Bishop, 12.2.1 p. 574 or http://www.miketipping.com/papers/met-mppca.pdf

Parameters

X [array, shape(n_samples, n_features)] The data.

Returns

ll [array, shape (n_samples,)] Log-likelihood of each sample under the current model

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)

Apply dimensionality reduction to X.

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

Parameters

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

Returns

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

Examples

```
>>> import numpy as np
>>> from sklearn.decomposition import IncrementalPCA
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> ipca = IncrementalPCA(n_components=2, batch_size=3)
>>> ipca.fit(X)
IncrementalPCA(batch_size=3, copy=True, n_components=2, whiten=False)
>>> ipca.transform(X)
```

Examples using `sklearn.decomposition.PCA`

- *Multilabel classification*
- *Explicit feature map approximation for RBF kernels*
- *Faces recognition example using eigenfaces and SVMs*
- *A demo of K-Means clustering on the handwritten digits data*
- *Concatenating multiple feature extraction methods*
- *Pipelining: chaining a PCA and a logistic regression*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *The Iris Dataset*
- *PCA example with Iris Data-set*
- *Incremental PCA*
- *Comparison of LDA and PCA 2D projection of Iris dataset*
- *Blind source separation using FastICA*
- *Principal components analysis (PCA)*
- *FastICA on 2D point clouds*
- *Kernel PCA*

- *Model selection with Probabilistic PCA and Factor Analysis (FA)*
- *Faces dataset decompositions*
- *Multi-dimensional scaling*
- *Balance model complexity and cross-validated score*
- *Kernel Density Estimation*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Using FunctionTransformer to select columns*
- *Importance of Feature Scaling*

6.9.11 sklearn.decomposition.SparsePCA

class sklearn.decomposition.**SparsePCA**(*n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *max_iter=1000*, *tol=1e-08*, *method='lars'*, *n_jobs=None*, *U_init=None*, *V_init=None*, *verbose=False*, *random_state=None*, *normalize_components=False*)

Sparse Principal Components Analysis (SparsePCA)

Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha.

Read more in the *User Guide*.

Parameters

n_components [int,] Number of sparse atoms to extract.

alpha [float,] Sparsity controlling parameter. Higher values lead to sparser components.

ridge_alpha [float,] Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.

max_iter [int,] Maximum number of iterations to perform.

tol [float,] Tolerance for the stopping condition.

method [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

U_init [array of shape (n_samples, n_components),] Initial values for the loadings for warm restart scenarios.

V_init [array of shape (n_components, n_features),] Initial values for the components for warm restart scenarios.

verbose [int] Controls the verbosity; the higher, the more messages. Defaults to 0.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

normalize_components [boolean, optional (default=False)]

- if False, use a version of Sparse PCA without components normalization and without data centering. This is likely a bug and even though it's the default for backward compatibility, this should not be used.
- if True, use a version of Sparse PCA with components normalization and data centering.

New in version 0.20.

Deprecated since version 0.22: `normalize_components` was added and set to `False` for backward compatibility. It would be set to `True` from 0.22 onwards.

Attributes

components_ [array, [n_components, n_features]] Sparse components extracted from the data.

error_ [array] Vector of errors at each iteration.

n_iter_ [int] Number of iterations run.

mean_ [array, shape (n_features,)] Per-feature empirical mean, estimated from the training set. Equal to `X.mean(axis=0)`.

See also:

PCA

MiniBatchSparsePCA

DictionaryLearning

Examples

```
>>> import numpy as np
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.decomposition import SparsePCA
>>> X, _ = make_friedman1(n_samples=200, n_features=30, random_state=0)
>>> transformer = SparsePCA(n_components=5,
...         normalize_components=True,
...         random_state=0)
>>> transformer.fit(X)
SparsePCA(...)
>>> X_transformed = transformer.transform(X)
>>> X_transformed.shape
(200, 5)
>>> # most values in the components_ are zero (sparsity)
>>> np.mean(transformer.components_ == 0)
0.9666...
```

Methods

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

Continued on next page

Table 6.60 – continued from previous page

transform(self, X)	Least Squares projection of the data onto the sparse components.

__init__ (*self*, *n_components=None*, *alpha=1*, *ridge_alpha=0.01*, *max_iter=1000*, *tol=1e-08*, *method='lars'*, *n_jobs=None*, *U_init=None*, *V_init=None*, *verbose=False*, *random_state=None*, *normalize_components=False*)

fit (*self*, *X*, *y=None*)
Fit the model from data in X.

Parameters

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

y [Ignored]

Returns

self [object] Returns the instance itself.

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, ***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.

Returns

self

transform (*self*, *X*)
Least Squares projection of the data onto the sparse components.

To avoid instability issues in case the system is under-determined, regularization can be applied (Ridge regression) via the `ridge_alpha` parameter.

Note that Sparse PCA components orthogonality is not enforced as in PCA hence one cannot use a simple linear projection.

Parameters

X [array of shape (n_samples, n_features)] Test data to be transformed, must have the same number of features as the data used to train the model.

Returns

X_new array, shape (n_samples, n_components) Transformed data.

6.9.12 sklearn.decomposition.SparseCoder

class sklearn.decomposition.**SparseCoder**(*dictionary*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *split_sign=False*, *n_jobs=None*, *positive_code=False*)

Sparse coding

Finds a sparse representation of data against a fixed, precomputed dictionary.

Each row of the result is the solution to a sparse coding problem. The goal is to find a sparse array `code` such that:

```
X ~= code * dictionary
```

Read more in the *User Guide*.

Parameters

dictionary [array, [n_components, n_features]] The dictionary atoms used for sparse coding. Lines are assumed to be normalized to unit norm.

transform_algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}] Algorithm used to transform the data: lars: uses the least angle regression method (linear_model.lars_path) lasso_lars: uses Lars to compute the Lasso solution lasso_cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). lasso_lars will be faster if the estimated components are sparse. omp: uses orthogonal matching pursuit to estimate the sparse solution threshold: squashes to zero all coefficients less than alpha from the projection `dictionary * X'`

transform_n_nonzero_coefs [int, `0.1 * n_features` by default] Number of nonzero coefficients to target in each column of the solution. This is only used by `algorithm='lars'` and `algorithm='omp'` and is overridden by `alpha` in the *Orthogonal Matching Pursuit (OMP)* case.

transform_alpha [float, 1. by default] If `algorithm='lasso_lars'` or `algorithm='lasso_cd'`, `alpha` is the penalty applied to the L1 norm. If `algorithm='threshold'`, `alpha` is the absolute value of the threshold below which coefficients will be squashed to zero. If `algorithm='omp'`, `alpha` is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides `n_nonzero_coefs`.

split_sign [bool, False by default] Whether to split the sparse feature vector into the concatenation of its negative part and its positive part. This can improve the performance of downstream classifiers.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

positive_code [bool] Whether to enforce positivity when finding the code.

New in version 0.20.

Attributes

components_ [array, [n_components, n_features]] The unchanged dictionary atoms

See also:

DictionaryLearning

MiniBatchDictionaryLearning

SparsePCA

MiniBatchSparsePCA

sparse_encode

Methods

fit(self, X[, y])	Do nothing and return the estimator unchanged
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Encode the data as a sparse combination of the dictionary atoms.

__init__(*self*, *dictionary*, *transform_algorithm='omp'*, *transform_n_nonzero_coefs=None*, *transform_alpha=None*, *split_sign=False*, *n_jobs=None*, *positive_code=False*)

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

Do nothing and return the estimator unchanged

This method is just there to implement the usual API and hence work in pipelines.

Parameters

X [Ignored]

y [Ignored]

Returns

self [object] Returns the object itself

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Encode the data as a sparse combination of the dictionary atoms.

Coding method is determined by the object parameter `transform_algorithm`.

Parameters

X [array of shape (n_samples, n_features)] Test data to be transformed, must have the same number of features as the data used to train the model.

Returns

X_new [array, shape (n_samples, n_components)] Transformed data

Examples using `sklearn.decomposition.SparseCoder`

- *Sparse coding with a precomputed dictionary*

6.9.13 `sklearn.decomposition`.TruncatedSVD

class `sklearn.decomposition.`**TruncatedSVD**(*n_components=2*, *algorithm='randomized'*, *n_iter=5*, *random_state=None*, *tol=0.0*)
Dimensionality reduction using truncated SVD (aka LSA).

This transformer performs linear dimensionality reduction by means of truncated singular value decomposition (SVD). Contrary to PCA, this estimator does not center the data before computing the singular value decomposition. This means it can work with scipy.sparse matrices efficiently.

In particular, truncated SVD works on term count/tf-idf matrices as returned by the vectorizers in sklearn.feature_extraction.text. In that context, it is known as latent semantic analysis (LSA).

This estimator supports two algorithms: a fast randomized SVD solver, and a "naive" algorithm that uses ARPACK as an eigensolver on (X * X.T) or (X.T * X), whichever is more efficient.

Read more in the *User Guide*.

Parameters

n_components [int, default = 2] Desired dimensionality of output data. Must be strictly less than the number of features. The default value is useful for visualisation. For LSA, a value of 100 is recommended.

algorithm [string, default = "randomized"] SVD solver to use. Either "arpack" for the ARPACK wrapper in SciPy (scipy.sparse.linalg.svds), or "randomized" for the randomized algorithm due to Halko (2009).

n_iter [int, optional (default 5)] Number of iterations for randomized SVD solver. Not used by ARPACK. The default is larger than the default in *randomized_svd* to handle sparse matrices that may have large slowly decaying spectrum.

random_state [int, RandomState instance or None, optional, default = None] 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`.

tol [float, optional] Tolerance for ARPACK. 0 means machine precision. Ignored by randomized SVD solver.

Attributes

components_ [array, shape (n_components, n_features)]

explained_variance_ [array, shape (n_components,)] The variance of the training samples transformed by a projection to each component.

explained_variance_ratio_ [array, shape (n_components,)] Percentage of variance explained by each of the selected components.

singular_values_ [array, shape (n_components,)] The singular values corresponding to each of the selected components. The singular values are equal to the 2-norms of the `n_components` variables in the lower-dimensional space.

See also:

PCA

Notes

SVD suffers from a problem called "sign indeterminacy", which means the sign of the `components_` and the output from transform depend on the algorithm and random state. To work around this, fit instances of this class to data once, then keep the instance around to do transformations.

References

Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 (arXiv:909) https://arxiv.org/pdf/0909.4061.pdf

Examples

```
>>> from sklearn.decomposition import TruncatedSVD
>>> from sklearn.random_projection import sparse_random_matrix
>>> X = sparse_random_matrix(100, 100, density=0.01, random_state=42)
>>> svd = TruncatedSVD(n_components=5, n_iter=7, random_state=42)
>>> svd.fit(X)
```

```
TruncatedSVD(algorithm='randomized', n_components=5, n_iter=7,
        random_state=42, tol=0.0)
>>> print(svd.explained_variance_ratio_)
[0.0606... 0.0584... 0.0497... 0.0434... 0.0372...]
>>> print(svd.explained_variance_ratio_.sum())
0.249...
>>> print(svd.singular_values_)
[2.5841... 2.5245... 2.3201... 2.1753... 2.0443...]
```

Methods

fit(self, X[, y])	Fit LSI model on training data X.
fit_transform(self, X[, y])	Fit LSI model to X and perform dimensionality reduction on X.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Transform X back to its original space.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Perform dimensionality reduction on X.

__init__ (*self*, *n_components=2*, *algorithm='randomized'*, *n_iter=5*, *random_state=None*, *tol=0.0*)

fit (*self*, *X*, *y=None*)
Fit LSI model on training data X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data.

y [Ignored]

Returns

self [object] Returns the transformer object.

fit_transform (*self*, *X*, *y=None*)
Fit LSI model to X and perform dimensionality reduction on X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data.

y [Ignored]

Returns

X_new [array, shape (n_samples, n_components)] Reduced version of X. This will always be a dense array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*)
Transform X back to its original space.

Returns an array X_original whose transform would be X.

Parameters

X [array-like, shape (n_samples, n_components)] New data.

Returns

X_original [array, shape (n_samples, n_features)] Note that this is always a dense array.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Perform dimensionality reduction on X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] New data.

Returns

X_new [array, shape (n_samples, n_components)] Reduced version of X. This will always be a dense array.

Examples using `sklearn.decomposition.TruncatedSVD`

- *Column Transformer with Heterogeneous Data Sources*
- *Hashing feature transformation using Totally Random Trees*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*
- *Clustering text documents using k-means*

decomposition.dict_learning(X, n_components, ...)	Solves a dictionary learning matrix factorization problem.
decomposition.dict_learning_online(X[, ...])	Solves a dictionary learning matrix factorization problem online.
decomposition.fastica(X[, n_components, ...])	Perform Fast Independent Component Analysis.
decomposition.non_negative_factorization(X)	Compute Non-negative Matrix Factorization (NMF)
decomposition.sparse_encode(X, dictionary[, ...])	Sparse coding

6.9.14 sklearn.decomposition.dict_learning

sklearn.decomposition.**dict_learning**(*X*, *n_components*, *alpha*, *max_iter=100*, *tol=1e-08*, *method='lars'*, *n_jobs=None*, *dict_init=None*, *code_init=None*, *callback=None*, *verbose=False*, *random_state=None*, *return_n_iter=False*, *positive_dict=False*, *positive_code=False*)

Solves a dictionary learning matrix factorization problem.

Finds the best dictionary and the corresponding sparse code for approximating the data matrix X by solving:

```
(U^*, V^*) = argmin 0.5 || X - U V ||_2^2 + alpha * || U ||_1
             (U,V)
            with || V_k ||_2 = 1 for all  0 <= k < n_components
```

where V is the dictionary and U is the sparse code.

Read more in the *User Guide*.

Parameters

X [array of shape (n_samples, n_features)] Data matrix.

n_components [int,] Number of dictionary atoms to extract.

alpha [int,] Sparsity controlling parameter.

max_iter [int,] Maximum number of iterations to perform.

tol [float,] Tolerance for the stopping condition.

method [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

dict_init [array of shape (n_components, n_features),] Initial value for the dictionary for warm restart scenarios.

code_init [array of shape (n_samples, n_components),] Initial value for the sparse code for warm restart scenarios.

callback [callable or None, optional (default: None)] Callable that gets invoked every five iterations

verbose [bool, optional (default: False)] To control the verbosity of the procedure.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

return_n_iter [bool] Whether or not to return the number of iterations.

positive_dict [bool] Whether to enforce positivity when finding the dictionary.

New in version 0.20.

positive_code [bool] Whether to enforce positivity when finding the code.

New in version 0.20.

Returns

code [array of shape (n_samples, n_components)] The sparse code factor in the matrix factorization.

dictionary [array of shape (n_components, n_features),] The dictionary factor in the matrix factorization.

errors [array] Vector of errors at each iteration.

n_iter [int] Number of iterations run. Returned only if `return_n_iter` is set to True.

See also:

dict_learning_online

DictionaryLearning

MiniBatchDictionaryLearning

SparsePCA

MiniBatchSparsePCA

6.9.15 sklearn.decomposition.dict_learning_online

sklearn.decomposition.**dict_learning_online**(*X*, *n_components=2*, *alpha=1*, *n_iter=100*, *return_code=True*, *dict_init=None*, *callback=None*, *batch_size=3*, *verbose=False*, *shuffle=True*, *n_jobs=None*, *method='lars'*, *iter_offset=0*, *random_state=None*, *return_inner_stats=False*, *inner_stats=None*, *return_n_iter=False*, *positive_dict=False*, *positive_code=False*)

Solves a dictionary learning matrix factorization problem online.

Finds the best dictionary and the corresponding sparse code for approximating the data matrix X by solving:

```
(U^*, V^*) = argmin 0.5 || X - U V ||_2^2 + alpha * || U ||_1
             (U,V)
             with || V_k ||_2 = 1 for all  0 <= k < n_components
```

where V is the dictionary and U is the sparse code. This is accomplished by repeatedly iterating over mini-batches by slicing the input data.

Read more in the *User Guide*.

Parameters

X [array of shape (n_samples, n_features)] Data matrix.

n_components [int,] Number of dictionary atoms to extract.

alpha [float,] Sparsity controlling parameter.

n_iter [int,] Number of iterations to perform.

return_code [boolean,] Whether to also return the code U or just the dictionary V.

dict_init [array of shape (n_components, n_features),] Initial value for the dictionary for warm restart scenarios.

callback [callable or None, optional (default: None)] callable that gets invoked every five iterations

batch_size [int,] The number of samples to take in each batch.

verbose [bool, optional (default: False)] To control the verbosity of the procedure.

shuffle [boolean,] Whether to shuffle the data before splitting it in batches.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

method [{'lars', 'cd'}] lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path) cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). Lars will be faster if the estimated components are sparse.

iter_offset [int, default 0] Number of previous iterations completed on the dictionary used for initialization.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

return_inner_stats [boolean, optional] Return the inner statistics A (dictionary covariance) and B (data approximation). Useful to restart the algorithm in an online setting. If return_inner_stats is True, return_code is ignored

inner_stats [tuple of (A, B) ndarrays] Inner sufficient statistics that are kept by the algorithm. Passing them at initialization is useful in online settings, to avoid loosing the history of the evolution. A (n_components, n_components) is the dictionary covariance matrix. B (n_features, n_components) is the data approximation matrix

return_n_iter [bool] Whether or not to return the number of iterations.

positive_dict [bool] Whether to enforce positivity when finding the dictionary.

New in version 0.20.

positive_code [bool] Whether to enforce positivity when finding the code.

New in version 0.20.

Returns

code [array of shape (n_samples, n_components),] the sparse code (only returned if `return_code=True`)

dictionary [array of shape (n_components, n_features),] the solutions to the dictionary learning problem

n_iter [int] Number of iterations run. Returned only if `return_n_iter` is set to `True`.

See also:

dict_learning

DictionaryLearning

MiniBatchDictionaryLearning

SparsePCA

MiniBatchSparsePCA

6.9.16 sklearn.decomposition.fastica

sklearn.decomposition.**fastica**(*X*, *n_components=None*, *algorithm='parallel'*, *whiten=True*, *fun='logcosh'*, *fun_args=None*, *max_iter=200*, *tol=0.0001*, *w_init=None*, *random_state=None*, *return_X_mean=False*, *compute_sources=True*, *return_n_iter=False*)

Perform Fast Independent Component Analysis.

Read more in the *User Guide*.

Parameters

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

n_components [int, optional] Number of components to extract. If None no dimension reduction is performed.

algorithm [{'parallel', 'deflation'}, optional] Apply a parallel or deflational FASTICA algorithm.

whiten [boolean, optional] If True perform an initial whitening of the data. If False, the data is assumed to have already been preprocessed: it should be centered, normed and white. Otherwise you will get incorrect results. In this case the parameter n_components will be ignored.

fun [string or function, optional. Default: 'logcosh'] The functional form of the G function used in the approximation to neg-entropy. Could be either 'logcosh', 'exp', or 'cube'. You can also provide your own function. It should return a tuple containing the value of the function, and of its derivative, in the point. The derivative should be averaged along its last dimension. Example:

def my_g(x): return x ** 3, np.mean(3 * x ** 2, axis=-1)

fun_args [dictionary, optional] Arguments to send to the functional form. If empty or None and if fun='logcosh', fun_args will take value {'alpha' : 1.0}

max_iter [int, optional] Maximum number of iterations to perform.

tol [float, optional] A positive scalar giving the tolerance at which the un-mixing matrix is considered to have converged.

w_init [(n_components, n_components) array, optional] Initial un-mixing array of dimension (n.comp,n.comp). If None (default) then an array of normal r.v.'s is used.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

return_X_mean [bool, optional] If True, X_mean is returned too.

compute_sources [bool, optional] If False, sources are not computed, but only the rotation matrix. This can save memory when working with big data. Defaults to True.

return_n_iter [bool, optional] Whether or not to return the number of iterations.

Returns

K [array, shape (n_components, n_features) | None.] If whiten is 'True', K is the pre-whitening matrix that projects data onto the first n_components principal components. If whiten is 'False', K is 'None'.

W [array, shape (n_components, n_components)] Estimated un-mixing matrix. The mixing matrix can be obtained by:

```
w = np.dot(W, K.T)
A = w.T * (w * w.T).I
```

S [array, shape (n_samples, n_components) | None] Estimated source matrix

X_mean [array, shape (n_features,)] The mean over features. Returned only if return_X_mean is True.

n_iter [int] If the algorithm is "deflation", n_iter is the maximum number of iterations run across all components. Else they are just the number of iterations taken to converge. This is returned only when return_n_iter is set to `True`.

Notes

The data matrix X is considered to be a linear combination of non-Gaussian (independent) components i.e. X = AS where columns of S contain the independent components and A is a linear mixing matrix. In short ICA attempts to `un-mix' the data by estimating an un-mixing matrix W where ``S = W K X.``

This implementation was originally made for data of shape [n_features, n_samples]. Now the input is transposed before the algorithm is applied. This makes it slightly faster for Fortran-ordered input.

Implemented using FastICA: *A. Hyvarinen and E. Oja, Independent Component Analysis: Algorithms and Applications, Neural Networks, 13(4-5), 2000, pp. 411-430*

6.9.17 sklearn.decomposition.non_negative_factorization

sklearn.decomposition.**non_negative_factorization**(*X*, *W=None*, *H=None*, *n_components=None*, *init='warn'*, *update_H=True*, *solver='cd'*, *beta_loss='frobenius'*, *tol=0.0001*, *max_iter=200*, *alpha=0.0*, *l1_ratio=0.0*, *regularization=None*, *random_state=None*, *verbose=0*, *shuffle=False*)

Compute Non-negative Matrix Factorization (NMF)

Find two non-negative matrices (W, H) whose product approximates the non- negative matrix X. This factorization can be used for example for dimensionality reduction, source separation or topic extraction.

The objective function is:

```
0.5 * ||X - WH||_Fro^2
+ alpha * l1_ratio * ||vec(W)||_1
+ alpha * l1_ratio * ||vec(H)||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
+ 0.5 * alpha * (1 - l1_ratio) * ||H||_Fro^2
```

Where:

```
||A||_Fro^2 = \sum_{i,j} A_{ij}^2 (Frobenius norm)
||vec(A)||_1 = \sum_{i,j} abs(A_{ij}) (Elementwise L1 norm)
```

For multiplicative-update ('mu') solver, the Frobenius norm (0.5 * ||X - WH||_Fro^2) can be changed into another beta-divergence loss, by changing the beta_loss parameter.

The objective function is minimized with an alternating minimization of W and H. If H is given and update_H=False, it solves for W only.

Parameters

X [array-like, shape (n_samples, n_features)] Constant matrix.

W [array-like, shape (n_samples, n_components)] If init='custom', it is used as initial guess for the solution.

H [array-like, shape (n_components, n_features)] If init='custom', it is used as initial guess for the solution. If update_H=False, it is used as a constant, to solve for W only.

n_components [integer] Number of components, if n_components is not set all features are kept.

init [None | 'random' | 'nndsvd' | 'nndsvda' | 'nndsvdar' | 'custom'] Method used to initialize the procedure. Default: 'random'.

The default value will change from 'random' to None in version 0.23 to make it consistent with decomposition.NMF.

Valid options:

- None: 'nndsvd' if n_components < n_features, otherwise 'random'.
- **'random': non-negative random matrices, scaled with:** sqrt(X.mean() / n_components)
- **'nndsvd': Nonnegative Double Singular Value Decomposition (NNDSVD)** initialization (better for sparseness)
- **'nndsvda': NNDSVD with zeros filled with the average of X** (better when sparsity is not desired)
- **'nndsvdar': NNDSVD with zeros filled with small random values** (generally faster, less accurate alternative to NNDSVDa for when sparsity is not desired)
- 'custom': use custom matrices W and H

update_H [boolean, default: True] Set to True, both W and H will be estimated from initial guesses. Set to False, only W will be estimated.

solver ['cd' | 'mu'] Numerical solver to use: 'cd' is a Coordinate Descent solver that uses Fast Hierarchical

Alternating Least Squares (Fast HALS).

'mu' is a Multiplicative Update solver.

New in version 0.17: Coordinate Descent solver.

New in version 0.19: Multiplicative Update solver.

beta_loss [float or string, default 'frobenius'] String must be in {'frobenius', 'kullback-leibler', 'itakura-saito'}. Beta divergence to be minimized, measuring the distance between X and the dot product WH. Note that values different from 'frobenius' (or 2) and 'kullback-leibler' (or 1) lead to significantly slower fits. Note that for beta_loss <= 0 (or 'itakura-saito'), the input matrix X cannot contain zeros. Used only in 'mu' solver.

New in version 0.19.

tol [float, default: 1e-4] Tolerance of the stopping condition.

max_iter [integer, default: 200] Maximum number of iterations before timing out.

alpha [double, default: 0.] Constant that multiplies the regularization terms.

l1_ratio [double, default: 0.] The regularization mixing parameter, with 0 <= l1_ratio <= 1. For l1_ratio = 0 the penalty is an elementwise L2 penalty (aka Frobenius Norm). For l1_ratio = 1 it is an elementwise L1 penalty. For 0 < l1_ratio < 1, the penalty is a combination of L1 and L2.

regularization ['both' | 'components' | 'transformation' | None] Select whether the regularization affects the components (H), the transformation (W), both or none of them.

random_state [int, RandomState instance or None, optional, default: None] 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`.

verbose [integer, default: 0] The verbosity level.

shuffle [boolean, default: False] If true, randomize the order of coordinates in the CD solver.

Returns

W [array-like, shape (n_samples, n_components)] Solution to the non-negative least squares problem.

H [array-like, shape (n_components, n_features)] Solution to the non-negative least squares problem.

n_iter [int] Actual number of iterations.

References

Cichocki, Andrzej, and P. H. A. N. Anh-Huy. "Fast local algorithms for large scale nonnegative matrix and tensor factorizations." IEICE transactions on fundamentals of electronics, communications and computer sciences 92.3: 708-721, 2009.

Fevotte, C., & Idier, J. (2011). Algorithms for nonnegative matrix factorization with the beta-divergence. Neural Computation, 23(9).

Examples

```
>>> import numpy as np
>>> X = np.array([[1,1], [2, 1], [3, 1.2], [4, 1], [5, 0.8], [6, 1]])
>>> from sklearn.decomposition import non_negative_factorization
>>> W, H, n_iter = non_negative_factorization(X, n_components=2,
... init='random', random_state=0)
```

6.9.18 sklearn.decomposition.sparse_encode

sklearn.decomposition.**sparse_encode**(*X*, *dictionary*, *gram=None*, *cov=None*, *algorithm='lasso_lars'*, *n_nonzero_coefs=None*, *alpha=None*, *copy_cov=True*, *init=None*, *max_iter=1000*, *n_jobs=None*, *check_input=True*, *verbose=0*, *positive=False*)

Sparse coding

Each row of the result is the solution to a sparse coding problem. The goal is to find a sparse array `code` such that:

```
X ~= code * dictionary
```

Read more in the *User Guide*.

Parameters

X [array of shape (n_samples, n_features)] Data matrix

dictionary [array of shape (n_components, n_features)] The dictionary matrix against which to solve the sparse coding of the data. Some of the algorithms assume normalized rows for meaningful output.

gram [array, shape=(n_components, n_components)] Precomputed Gram matrix, dictionary * dictionary'

cov [array, shape=(n_components, n_samples)] Precomputed covariance, dictionary' * X

algorithm [{'lasso_lars', 'lasso_cd', 'lars', 'omp', 'threshold'}] lars: uses the least angle regression method (linear_model.lars_path) lasso_lars: uses Lars to compute the Lasso solution lasso_cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso). lasso_lars will be faster if the estimated components are sparse. omp: uses orthogonal matching pursuit to estimate the sparse solution threshold: squashes to zero all coefficients less than alpha from the projection dictionary * X'

n_nonzero_coefs [int, 0.1 * n_features by default] Number of nonzero coefficients to target in each column of the solution. This is only used by `algorithm='lars'` and `algorithm='omp'` and is overridden by `alpha` in the *Orthogonal Matching Pursuit (OMP)* case.

alpha [float, 1. by default] If `algorithm='lasso_lars'` or `algorithm='lasso_cd'`, `alpha` is the penalty applied to the L1 norm. If `algorithm='threshold'`, `alpha` is the absolute value of the threshold below which coefficients will be squashed to zero. If `algorithm='omp'`, `alpha` is the tolerance parameter: the value of the reconstruction error targeted. In this case, it overrides `n_nonzero_coefs`.

copy_cov [boolean, optional] Whether to copy the precomputed covariance matrix; if False, it may be overwritten.

init [array of shape (n_samples, n_components)] Initialization value of the sparse codes. Only used if `algorithm='lasso_cd'`.

max_iter [int, 1000 by default] Maximum number of iterations to perform if `algorithm='lasso_cd'`.

n_jobs [int or None, optional (default=None)] Number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

check_input [boolean, optional] If False, the input arrays X and dictionary will not be checked.

verbose [int, optional] Controls the verbosity; the higher, the more messages. Defaults to 0.

positive [boolean, optional] Whether to enforce positivity when finding the encoding.

New in version 0.20.

Returns

code [array of shape (n_samples, n_components)] The sparse codes

See also:

sklearn.linear_model.lars_path

sklearn.linear_model.orthogonal_mp

sklearn.linear_model.Lasso

SparseCoder

6.10 sklearn.discriminant_analysis: Discriminant Analysis

Linear Discriminant Analysis and Quadratic Discriminant Analysis

User guide: See the *Linear and Quadratic Discriminant Analysis* section for further details.

discriminant_analysis.LinearDiscriminantAnalysis([...])	Linear Discriminant Analysis
discriminant_analysis.QuadraticDiscriminantAnalysis([...])	Quadratic Discriminant Analysis

6.10.1 sklearn.discriminant_analysis.LinearDiscriminantAnalysis

class sklearn.discriminant_analysis.**LinearDiscriminantAnalysis**(*solver='svd'*, *shrinkage=None*, *priors=None*, *n_components=None*, *store_covariance=False*, *tol=0.0001*)

Linear Discriminant Analysis

A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.

The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix.

The fitted model can also be used to reduce the dimensionality of the input by projecting it to the most discriminative directions.

New in version 0.17: *LinearDiscriminantAnalysis*.

Read more in the *User Guide*.

Parameters

solver [string, optional]

Solver to use, possible values:

- 'svd': Singular value decomposition (default). Does not compute the covariance matrix, therefore this solver is recommended for data with a large number of features.
- 'lsqr': Least squares solution, can be combined with shrinkage.
- 'eigen': Eigenvalue decomposition, can be combined with shrinkage.

shrinkage [string or float, optional]

Shrinkage parameter, possible values:

- None: no shrinkage (default).
- 'auto': automatic shrinkage using the Ledoit-Wolf lemma.
- float between 0 and 1: fixed shrinkage parameter.

Note that shrinkage works only with 'lsqr' and 'eigen' solvers.

priors [array, optional, shape (n_classes,)] Class priors.

n_components [int, optional (default=None)] Number of components (<= min(n_classes - 1, n_features)) for dimensionality reduction. If None, will be set to min(n_classes - 1, n_features).

store_covariance [bool, optional] Additionally compute class covariance matrix (default False), used only in 'svd' solver.

New in version 0.17.

tol [float, optional, (default 1.0e-4)] Threshold used for rank estimation in SVD solver.

New in version 0.17.

Attributes

coef_ [array, shape (n_features,) or (n_classes, n_features)] Weight vector(s).

intercept_ [array, shape (n_features,)] Intercept term.

covariance_ [array-like, shape (n_features, n_features)] Covariance matrix (shared by all classes).

explained_variance_ratio_ [array, shape (n_components,)] Percentage of variance explained by each of the selected components. If `n_components` is not set then all components are stored and the sum of explained variances is equal to 1.0. Only available when eigen or svd solver is used.

means_ [array-like, shape (n_classes, n_features)] Class means.

priors_ [array-like, shape (n_classes,)] Class priors (sum to 1).

scalings_ [array-like, shape (rank, n_classes - 1)] Scaling of the features in the space spanned by the class centroids.

xbar_ [array-like, shape (n_features,)] Overall mean.

classes_ [array-like, shape (n_classes,)] Unique class labels.

See also:

sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis Quadratic Discriminant Analysis

Notes

The default solver is 'svd'. It can perform both classification and transform, and it does not rely on the calculation of the covariance matrix. This can be an advantage in situations where the number of features is large. However, the 'svd' solver cannot be used with shrinkage.

The 'lsqr' solver is an efficient algorithm that only works for classification. It supports shrinkage.

The 'eigen' solver is based on the optimization of the between class scatter to within class scatter ratio. It can be used for both classification and transform, and it supports shrinkage. However, the 'eigen' solver needs to compute the covariance matrix, so it might not be suitable for situations with a high number of features.

Examples

```
>>> import numpy as np
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
>>> 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 = LinearDiscriminantAnalysis()
>>> clf.fit(X, y)
LinearDiscriminantAnalysis(n_components=None, priors=None, shrinkage=None,
              solver='svd', store_covariance=False, tol=0.0001)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
fit(self, X, y)	Fit LinearDiscriminantAnalysis model according to the given training data and parameters.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class labels for samples in X.
predict_log_proba(self, X)	Estimate log probability.
predict_proba(self, X)	Estimate probability.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Project data to maximize class separation.

__init__(*self*, *solver='svd'*, *shrinkage=None*, *priors=None*, *n_components=None*, *store_covariance=False*, *tol=0.0001*)

decision_function(*self*, *X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

fit(*self*, *X*, *y*)

Fit LinearDiscriminantAnalysis model according to the given training data and parameters.

Changed in version 0.19: *store_covariance* has been moved to main constructor.

Changed in version 0.19: *tol* has been moved to main constructor.

Parameters

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

y [array, shape (n_samples,)] Target values.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

predict_log_proba(*self*, *X*)
Estimate log probability.

Parameters

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

Returns

C [array, shape (n_samples, n_classes)] Estimated log probabilities.

predict_proba(*self*, *X*)
Estimate probability.

Parameters

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

Returns

C [array, shape (n_samples, n_classes)] Estimated probabilities.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Project data to maximize class separation.

Parameters

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

Returns

X_new [array, shape (n_samples, n_components)] Transformed data.

Examples using `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`

- *Normal and Shrinkage Linear Discriminant Analysis for classification*
- *Linear and Quadratic Discriminant Analysis with covariance ellipsoid*
- *Comparison of LDA and PCA 2D projection of Iris dataset*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*
- *Dimensionality Reduction with Neighborhood Components Analysis*

6.10.2 sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis

class sklearn.discriminant_analysis.**QuadraticDiscriminantAnalysis**(*priors=None*, *reg_param=0.0*, *store_covariance=False*, *tol=0.0001*)

Quadratic Discriminant Analysis

A classifier with a quadratic decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.

The model fits a Gaussian density to each class.

New in version 0.17: *QuadraticDiscriminantAnalysis*

Read more in the *User Guide*.

Parameters

priors [array, optional, shape = [n_classes]] Priors on classes

reg_param [float, optional] Regularizes the covariance estimate as `(1-reg_param)*Sigma + reg_param*np.eye(n_features)`

store_covariance [boolean] If True the covariance matrices are computed and stored in the `self.covariance_` attribute.

New in version 0.17.

tol [float, optional, default 1.0e-4] Threshold used for rank estimation.

New in version 0.17.

Attributes

covariance_ [list of array-like, shape = [n_features, n_features]] Covariance matrices of each class.

means_ [array-like, shape = [n_classes, n_features]] Class means.

priors_ [array-like, shape = [n_classes]] Class priors (sum to 1).

rotations_ [list of arrays] For each class k an array of shape [n_features, n_k], with `n_k = min(n_features, number of elements in class k)` It is the rotation of the Gaussian distribution, i.e. its principal axis.

scalings_ [list of arrays] For each class k an array of shape [n_k]. It contains the scaling of the Gaussian distributions along its principal axes, i.e. the variance in the rotated coordinate system.

See also:

sklearn.discriminant_analysis.LinearDiscriminantAnalysis Linear Discriminant Analysis

Examples

```
>>> from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
>>> 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 = QuadraticDiscriminantAnalysis()
>>> clf.fit(X, y)
...
QuadraticDiscriminantAnalysis(priors=None, reg_param=0.0,
                              store_covariance=False, tol=0.0001)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(self, X)	Apply decision function to an array of samples.
fit(self, X, y)	Fit the model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.

Continued on next page

Table 6.66 – continued from previous page

predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return posterior probabilities of classification.
predict_proba(self, X)	Return posterior probabilities of classification.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *priors=None*, *reg_param=0.0*, *store_covariance=False*, *tol=0.0001*)

decision_function (*self*, *X*)

Apply decision function to an array of samples.

Parameters

X [array-like, shape = [n_samples, n_features]] Array of samples (test vectors).

Returns

C [array, shape = [n_samples, n_classes] or [n_samples,]] Decision function values related to each class, per sample. In the two-class case, the shape is [n_samples,], giving the log likelihood ratio of the positive class.

fit (*self*, *X*, *y*)

Fit the model according to the given training data and parameters.

Changed in version 0.19: `store_covariances` has been moved to main constructor as `store_covariance`

Changed in version 0.19: `tol` has been moved to main constructor.

Parameters

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

y [array, shape = [n_samples]] Target values (integers)

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *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, shape = [n_samples, n_features]]

Returns

C [array, shape = [n_samples]]

predict_log_proba(*self*, *X*)
Return posterior probabilities of classification.

Parameters

X [array-like, shape = [n_samples, n_features]] Array of samples/test vectors.

Returns

C [array, shape = [n_samples, n_classes]] Posterior log-probabilities of classification per class.

predict_proba(*self*, *X*)
Return posterior probabilities of classification.

Parameters

X [array-like, shape = [n_samples, n_features]] Array of samples/test vectors.

Returns

C [array, shape = [n_samples, n_classes]] Posterior probabilities of classification per class.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis`

- *Classifier comparison*
- *Linear and Quadratic Discriminant Analysis with covariance ellipsoid*

6.11 `sklearn.dummy`: Dummy estimators

User guide: See the *Model evaluation: quantifying the quality of predictions* section for further details.

dummy.DummyClassifier([strategy, ...])	DummyClassifier is a classifier that makes predictions using simple rules.
dummy.DummyRegressor([strategy, constant, ...])	DummyRegressor is a regressor that makes predictions using simple rules.

6.11.1 sklearn.dummy.DummyClassifier

class sklearn.dummy.**DummyClassifier**(*strategy='stratified'*, *random_state=None*, *constant=None*)

DummyClassifier is a classifier that makes predictions using simple rules.

This classifier is useful as a simple baseline to compare with other (real) classifiers. Do not use it for real problems.

Read more in the *User Guide*.

Parameters

strategy [str, default="stratified"] Strategy to use to generate predictions.

- "stratified": generates predictions by respecting the training set's class distribution.
- "most_frequent": always predicts the most frequent label in the training set.
- "prior": always predicts the class that maximizes the class prior (like "most_frequent") and `predict_proba` returns the class prior.
- "uniform": generates predictions uniformly at random.
- "constant": always predicts a constant label that is provided by the user. This is useful for metrics that evaluate a non-majority class

  New in version 0.17: Dummy Classifier now supports prior fitting strategy using parameter *prior*.

random_state [int, RandomState instance or None, optional, default=None] 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`.

constant [int or str or array of shape = [n_outputs]] The explicit constant as predicted by the "constant" strategy. This parameter is useful only for the "constant" strategy.

Attributes

classes_ [array or list of array of shape = [n_classes]] Class labels for each output.

n_classes_ [array or list of array of shape = [n_classes]] Number of label for each output.

class_prior_ [array or list of array of shape = [n_classes]] Probability of each class for each output.

n_outputs_ [int,] Number of outputs.

sparse_output_ [bool,] True if the array returned from predict is to be in sparse CSC format. Is automatically set to True if the input y is passed in sparse format.

Methods

fit(self, X, y[, sample_weight])	Fit the random classifier.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform classification on test vectors X.
predict_log_proba(self, X)	Return log probability estimates for the test vectors X.
predict_proba(self, X)	Return probability estimates for the test vectors X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *strategy='stratified'*, *random_state=None*, *constant=None*)

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the random classifier.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] Target values.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Perform classification on test vectors X.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

Returns

y [array, shape = [n_samples] or [n_samples, n_outputs]] Predicted target values for X.

predict_log_proba (*self*, *X*)
Return log probability estimates for the test vectors X.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

Returns

P [array-like or list of array-like of shape = [n_samples, n_classes]] Returns the log probability of the sample for each class in the model, where classes are ordered arithmetically for each output.

predict_proba(*self*, *X*)
Return probability estimates for the test vectors X.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

Returns

P [array-like or list of array-lke of shape = [n_samples, n_classes]] Returns the probability of the sample for each class in the model, where classes are ordered arithmetically, for each output.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, None}] Test samples with shape = (n_samples, n_features) or None. Passing None as test samples gives the same result as passing real test samples, since DummyClassifier operates independently of the sampled observations.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

6.11.2 sklearn.dummy.DummyRegressor

class sklearn.dummy.**DummyRegressor**(*strategy='mean'*, *constant=None*, *quantile=None*)
DummyRegressor is a regressor that makes predictions using simple rules.

This regressor is useful as a simple baseline to compare with other (real) regressors. Do not use it for real problems.

Read more in the *User Guide*.

Parameters

strategy [str] Strategy to use to generate predictions.

- "mean": always predicts the mean of the training set
- "median": always predicts the median of the training set
- "quantile": always predicts a specified quantile of the training set, provided with the quantile parameter.

- "constant": always predicts a constant value that is provided by the user.

constant [int or float or array of shape = [n_outputs]] The explicit constant as predicted by the "constant" strategy. This parameter is useful only for the "constant" strategy.

quantile [float in [0.0, 1.0]] The quantile to predict using the "quantile" strategy. A quantile of 0.5 corresponds to the median, while 0.0 to the minimum and 1.0 to the maximum.

Attributes

constant_ [float or array of shape [n_outputs]] Mean or median or quantile of the training targets or constant value given by the user.

n_outputs_ [int,] Number of outputs.

Methods

fit(self, X, y[, sample_weight])	Fit the random regressor.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, return_std])	Perform classification on test vectors X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *strategy='mean'*, *constant=None*, *quantile=None*)

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the random regressor.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] Target values.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*, *return_std=False*)
Perform classification on test vectors X.

Parameters

X [{array-like, object with finite length or shape}] Training data, requires length = n_samples

return_std [boolean, optional] Whether to return the standard deviation of posterior prediction. All zeros in this case.

Returns

y [array, shape = [n_samples] or [n_samples, n_outputs]] Predicted target values for X.

y_std [array, shape = [n_samples] or [n_samples, n_outputs]] Standard deviation of predictive distribution of query points.

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

Returns 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, None}] Test samples with shape = (n_samples, n_features) or None. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator. Passing None as test samples gives the same result as passing real test samples, since DummyRegressor operates independently of the sampled observations.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

—

6.12 `sklearn.ensemble`: Ensemble Methods

The *`sklearn.ensemble`* module includes ensemble-based methods for classification, regression and anomaly detection.

User guide: See the *Ensemble methods* section for further details.

`ensemble.AdaBoostClassifier`([...])	An AdaBoost classifier.
`ensemble.AdaBoostRegressor`([base_estimator, ...])	An AdaBoost regressor.
`ensemble.BaggingClassifier`([base_estimator, ...])	A Bagging classifier.

Continued on next page

Table 6.71 – continued from previous page

ensemble.BaggingRegressor([base_estimator, ...])	A Bagging regressor.
ensemble.ExtraTreesClassifier([...])	An extra-trees classifier.
ensemble.ExtraTreesRegressor([n_estimators, ...])	An extra-trees regressor.
ensemble.GradientBoostingClassifier([loss, ...])	Gradient Boosting for classification.
ensemble.GradientBoostingRegressor([loss, ...])	Gradient Boosting for regression.
ensemble.IsolationForest([n_estimators, ...])	Isolation Forest Algorithm
ensemble.RandomForestClassifier([...])	A random forest classifier.
ensemble.RandomForestRegressor([...])	A random forest regressor.
ensemble.RandomTreesEmbedding([...])	An ensemble of totally random trees.
ensemble.VotingClassifier(estimators[, ...])	Soft Voting/Majority Rule classifier for unfitted estimators.
ensemble.VotingRegressor(estimators[, ...])	Prediction voting regressor for unfitted estimators.
ensemble.HistGradientBoostingRegressor([...])	Histogram-based Gradient Boosting Regression Tree.
ensemble.HistGradientBoostingClassifier([...])	Histogram-based Gradient Boosting Classification Tree.

6.12.1 sklearn.ensemble.AdaBoostClassifier

class sklearn.ensemble.**AdaBoostClassifier**(*base_estimator=None*, *n_estimators=50*, *learning_rate=1.0*, *algorithm='SAMME.R'*, *random_state=None*)

An AdaBoost classifier.

An AdaBoost [1] classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

This class implements the algorithm known as AdaBoost-SAMME [2].

Read more in the *User Guide*.

Parameters

base_estimator [object, optional (default=None)] The base estimator from which the boosted ensemble is built. Support for sample weighting is required, as well as proper `classes_` and `n_classes_` attributes. If `None`, then the base estimator is `DecisionTreeClassifier(max_depth=1)`

n_estimators [integer, optional (default=50)] The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

learning_rate [float, optional (default=1.)] Learning rate shrinks the contribution of each classifier by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

algorithm [{'SAMME', 'SAMME.R'}, optional (default='SAMME.R')] If 'SAMME.R' then use the SAMME.R real boosting algorithm. `base_estimator` must support calculation of class probabilities. If 'SAMME' then use the SAMME discrete boosting algorithm. The SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Attributes

estimators_ [list of classifiers] The collection of fitted sub-estimators.

classes_ [array of shape = [n_classes]] The classes labels.

n_classes_ [int] The number of classes.

estimator_weights_ [array of floats] Weights for each estimator in the boosted ensemble.

estimator_errors_ [array of floats] Classification error for each estimator in the boosted ensemble.

feature_importances_ [array of shape = [n_features]] Return the feature importances (the higher, the more important the feature).

See also:

AdaBoostRegressor, ***GradientBoostingClassifier***

sklearn.tree.DecisionTreeClassifier

References

[R33e4ec8c4ad5-1], *[R33e4ec8c4ad5-2]*

Examples

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = AdaBoostClassifier(n_estimators=100, random_state=0)
>>> clf.fit(X, y)
AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
        learning_rate=1.0, n_estimators=100, random_state=0)
>>> clf.feature_importances_
array([0.28..., 0.42..., 0.14..., 0.16...])
>>> clf.predict([[0, 0, 0, 0]])
array([1])
>>> clf.score(X, y)
0.983...
```

Methods

decision_function(self, X)	Compute the decision function of X.
fit(self, X, y[, sample_weight])	Build a boosted classifier from the training set (X, y).
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict classes for X.
predict_log_proba(self, X)	Predict class log-probabilities for X.

Continued on next page

Table 6.72 – continued from previous page

predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
staged_decision_function(self, X)	Compute decision function of X for each boosting iteration.
staged_predict(self, X)	Return staged predictions for X.
staged_predict_proba(self, X)	Predict class probabilities for X.
staged_score(self, X, y[, sample_weight])	Return staged scores for X, y.

__init__ (*self*, *base_estimator=None*, *n_estimators=50*, *learning_rate=1.0*, *algorithm='SAMME.R'*, *random_state=None*)

decision_function (*self*, *X*)
Compute the decision function of X.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

score [array, shape = [n_samples, k]] The decision function of the input samples. The order of outputs is the same of that of the *classes_* attribute. Binary classification is a special cases with `k == 1`, otherwise `k==n_classes`. For binary classification, values closer to -1 or 1 mean more like the first or second class in `classes_`, respectively.

feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]]

fit (*self*, *X*, *y*, *sample_weight=None*)
Build a boosted classifier from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

y [array-like of shape = [n_samples]] The target values (class labels).

sample_weight [array-like of shape = [n_samples], optional] Sample weights. If None, the sample weights are initialized to `1 / n_samples`.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)

Predict classes for X.

The predicted class of an input sample is computed as the weighted mean prediction of the classifiers in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

y [array of shape = [n_samples]] The predicted classes.

predict_log_proba (*self*, *X*)

Predict class log-probabilities for X.

The predicted class log-probabilities of an input sample is computed as the weighted mean predicted class log-probabilities of the classifiers in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

p [array of shape = [n_samples, n_classes]] The class probabilities of the input samples. The order of outputs is the same of that of the *classes_* attribute.

predict_proba (*self*, *X*)

Predict class probabilities for X.

The predicted class probabilities of an input sample is computed as the weighted mean predicted class probabilities of the classifiers in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

p [array of shape = [n_samples, n_classes]] The class probabilities of the input samples. The order of outputs is the same of that of the *classes_* attribute.

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

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

staged_decision_function(*self*, *X*)
Compute decision function of `X` for each boosting iteration.

This method allows monitoring (i.e. determine error on testing set) after each boosting iteration.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

score [generator of array, shape = [n_samples, k]] The decision function of the input samples. The order of outputs is the same of that of the *classes_* attribute. Binary classification is a special cases with `k == 1`, otherwise `k==n_classes`. For binary classification, values closer to -1 or 1 mean more like the first or second class in `classes_`, respectively.

staged_predict(*self*, *X*)
Return staged predictions for X.

The predicted class of an input sample is computed as the weighted mean prediction of the classifiers in the ensemble.

This generator method yields the ensemble prediction after each iteration of boosting and therefore allows monitoring, such as to determine the prediction on a test set after each boost.

Parameters

X [array-like of shape = [n_samples, n_features]] The input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

y [generator of array, shape = [n_samples]] The predicted classes.

staged_predict_proba(*self*, *X*)
Predict class probabilities for X.

The predicted class probabilities of an input sample is computed as the weighted mean predicted class probabilities of the classifiers in the ensemble.

This generator method yields the ensemble predicted class probabilities after each iteration of boosting and therefore allows monitoring, such as to determine the predicted class probabilities on a test set after each boost.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

p [generator of array, shape = [n_samples]] The class probabilities of the input samples. The order of outputs is the same of that of the *classes_* attribute.

staged_score(*self*, *X*, *y*, *sample_weight=None*)

Return staged scores for X, y.

This generator method yields the ensemble score after each iteration of boosting and therefore allows monitoring, such as to determine the score on a test set after each boost.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

y [array-like, shape = [n_samples]] Labels for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

z [float]

Examples using `sklearn.ensemble.AdaBoostClassifier`

- *Classifier comparison*
- *Two-class AdaBoost*
- *Multi-class AdaBoosted Decision Trees*
- *Discrete versus Real AdaBoost*
- *Plot the decision surfaces of ensembles of trees on the iris dataset*

6.12.2 `sklearn.ensemble`.AdaBoostRegressor

class `sklearn.ensemble.`**AdaBoostRegressor**(*base_estimator=None*, *n_estimators=50*, *learning_rate=1.0*, *loss='linear'*, *random_state=None*)

An AdaBoost regressor.

An AdaBoost [1] regressor is a meta-estimator that begins by fitting a regressor on the original dataset and then fits additional copies of the regressor on the same dataset but where the weights of instances are adjusted according to the error of the current prediction. As such, subsequent regressors focus more on difficult cases.

This class implements the algorithm known as AdaBoost.R2 [2].

Read more in the *User Guide*.

Parameters

base_estimator [object, optional (default=None)] The base estimator from which the boosted ensemble is built. Support for sample weighting is required. If `None`, then the base estimator is `DecisionTreeRegressor(max_depth=3)`

n_estimators [integer, optional (default=50)] The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

learning_rate [float, optional (default=1.)] Learning rate shrinks the contribution of each regressor by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.

loss [{'linear', 'square', 'exponential'}, optional (default='linear')] The loss function to use when updating the weights after each boosting iteration.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Attributes

estimators_ [list of classifiers] The collection of fitted sub-estimators.

estimator_weights_ [array of floats] Weights for each estimator in the boosted ensemble.

estimator_errors_ [array of floats] Regression error for each estimator in the boosted ensemble.

feature_importances_ [array of shape = [n_features]] Return the feature importances (the higher, the more important the feature).

See also:

AdaBoostClassifier, ***GradientBoostingRegressor***

sklearn.tree.DecisionTreeRegressor

References

[R0c261b7dee9d-1], *[R0c261b7dee9d-2]*

Examples

```
>>> from sklearn.ensemble import AdaBoostRegressor
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_features=4, n_informative=2,
...                        random_state=0, shuffle=False)
>>> regr = AdaBoostRegressor(random_state=0, n_estimators=100)
>>> regr.fit(X, y)
AdaBoostRegressor(base_estimator=None, learning_rate=1.0, loss='linear',
        n_estimators=100, random_state=0)
>>> regr.feature_importances_
array([0.2788..., 0.7109..., 0.0065..., 0.0036...])
>>> regr.predict([[0, 0, 0, 0]])
array([4.7972...])
>>> regr.score(X, y)
0.9771...
```

Methods

fit(self, X, y[, sample_weight])	Build a boosted regressor from the training set (X, y).

Continued on next page

Table 6.73 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict regression value for X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
staged_predict(self, X)	Return staged predictions for X.
staged_score(self, X, y[, sample_weight])	Return staged scores for X, y.

__init__ (*self*, *base_estimator=None*, *n_estimators=50*, *learning_rate=1.0*, *loss='linear'*, *random_state=None*)

feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]]

fit (*self*, *X*, *y*, *sample_weight=None*)
Build a boosted regressor from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

y [array-like of shape = [n_samples]] The target values (real numbers).

sample_weight [array-like of shape = [n_samples], optional] Sample weights. If None, the sample weights are initialized to 1 / n_samples.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict regression value for X.

The predicted regression value of an input sample is computed as the weighted median prediction of the classifiers in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

Returns

y [array of shape = [n_samples]] The predicted regression values.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

staged_predict (*self*, *X*)

Return staged predictions for X.

The predicted regression value of an input sample is computed as the weighted median prediction of the classifiers in the ensemble.

This generator method yields the ensemble prediction after each iteration of boosting and therefore allows monitoring, such as to determine the prediction on a test set after each boost.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples.

Returns

y [generator of array, shape = [n_samples]] The predicted regression values.

staged_score(*self*, *X*, *y*, *sample_weight=None*)
Return staged scores for X, y.

This generator method yields the ensemble score after each iteration of boosting and therefore allows monitoring, such as to determine the score on a test set after each boost.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrix can be CSC, CSR, COO, DOK, or LIL. COO, DOK, and LIL are converted to CSR.

y [array-like, shape = [n_samples]] Labels for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

z [float]

Examples using sklearn.ensemble.AdaBoostRegressor

- *Decision Tree Regression with AdaBoost*

6.12.3 sklearn.ensemble.BaggingClassifier

class sklearn.ensemble.**BaggingClassifier**(*base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *warm_start=False*, *n_jobs=None*, *random_state=None*, *verbose=0*)
A Bagging classifier.

A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

This algorithm encompasses several works from the literature. When random subsets of the dataset are drawn as random subsets of the samples, then this algorithm is known as Pasting *[Rb1846455d0e5-1]*. If samples are drawn with replacement, then the method is known as Bagging *[Rb1846455d0e5-2]*. When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces *[Rb1846455d0e5-3]*. Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches *[Rb1846455d0e5-4]*.

Read more in the *User Guide*.

Parameters

base_estimator [object or None, optional (default=None)] The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a decision tree.

n_estimators [int, optional (default=10)] The number of base estimators in the ensemble.

max_samples [int or float, optional (default=1.0)] The number of samples to draw from X to train each base estimator.

- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples.

max_features [int or float, optional (default=1.0)] The number of features to draw from X to train each base estimator.

- If int, then draw `max_features` features.
- If float, then draw `max_features * X.shape[1]` features.

bootstrap [boolean, optional (default=True)] Whether samples are drawn with replacement. If False, sampling without replacement is performed.

bootstrap_features [boolean, optional (default=False)] Whether features are drawn with replacement.

oob_score [bool, optional (default=False)] Whether to use out-of-bag samples to estimate the generalization error.

warm_start [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new ensemble. See *the Glossary*.

New in version 0.17: *warm_start* constructor parameter.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both *fit* and *predict*. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

Attributes

base_estimator_ [estimator] The base estimator from which the ensemble is grown.

estimators_ [list of estimators] The collection of fitted base estimators.

estimators_samples_ [list of arrays] The subset of drawn samples for each base estimator.

estimators_features_ [list of arrays] The subset of drawn features for each base estimator.

classes_ [array of shape = [n_classes]] The classes labels.

n_classes_ [int or list] The number of classes.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_decision_function_ [array of shape = [n_samples, n_classes]] Decision function computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, `oob_decision_function_` might contain NaN.

References

[Rb1846455d0e5-1], *[Rb1846455d0e5-2]*, *[Rb1846455d0e5-3]*, *[Rb1846455d0e5-4]*

Methods

decision_function(self, X)	Average of the decision functions of the base classifiers.
fit(self, X, y[, sample_weight])	Build a Bagging ensemble of estimators from the training set (X, y).
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class for X.
predict_log_proba(self, X)	Predict class log-probabilities for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *warm_start=False*, *n_jobs=None*, *random_state=None*, *verbose=0*)

decision_function (*self*, *X*)

Average of the decision functions of the base classifiers.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

Returns

score [array, shape = [n_samples, k]] The decision function of the input samples. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`. Regression and binary classification are special cases with `k == 1`, otherwise `k==n_classes`.

estimators_samples_

The subset of drawn samples for each base estimator.

Returns a dynamically generated list of indices identifying the samples used for fitting each member of the ensemble, i.e., the in-bag samples.

Note: the list is re-created at each call to the property in order to reduce the object memory footprint by not storing the sampling data. Thus fetching the property may be slower than expected.

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

Build a Bagging ensemble of estimators from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

y [array-like, shape = [n_samples]] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Note that this is supported only if the base estimator supports sample weighting.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)
Predict class for X.

The predicted class of an input sample is computed as the class with the highest mean predicted probability. If base estimators do not implement a `predict_proba` method, then it resorts to voting.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

Returns

y [array of shape = [n_samples]] The predicted classes.

predict_log_proba(*self*, *X*)
Predict class log-probabilities for X.

The predicted class log-probabilities of an input sample is computed as the log of the mean predicted class probabilities of the base estimators in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

Returns

p [array of shape = [n_samples, n_classes]] The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba(*self*, *X*)
Predict class probabilities for X.

The predicted class probabilities of an input sample is computed as the mean predicted class probabilities of the base estimators in the ensemble. If base estimators do not implement a `predict_proba` method, then it resorts to voting and the predicted class probabilities of an input sample represents the proportion of estimators predicting each class.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

Returns

p [array of shape = [n_samples, n_classes]] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

6.12.4 sklearn.ensemble.BaggingRegressor

class sklearn.ensemble.**BaggingRegressor**(*base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *warm_start=False*, *n_jobs=None*, *random_state=None*, *verbose=0*)

A Bagging regressor.

A Bagging regressor is an ensemble meta-estimator that fits base regressors each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

This algorithm encompasses several works from the literature. When random subsets of the dataset are drawn as random subsets of the samples, then this algorithm is known as Pasting *[R4d113ba76fc0-1]*. If samples are drawn with replacement, then the method is known as Bagging *[R4d113ba76fc0-2]*. When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces *[R4d113ba76fc0-3]*. Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches *[R4d113ba76fc0-4]*.

Read more in the *User Guide*.

Parameters

base_estimator [object or None, optional (default=None)] The base estimator to fit on random subsets of the dataset. If None, then the base estimator is a decision tree.

n_estimators [int, optional (default=10)] The number of base estimators in the ensemble.

max_samples [int or float, optional (default=1.0)] The number of samples to draw from X to train each base estimator.

- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples.

max_features [int or float, optional (default=1.0)] The number of features to draw from X to train each base estimator.

- If int, then draw `max_features` features.
- If float, then draw `max_features * X.shape[1]` features.

bootstrap [boolean, optional (default=True)] Whether samples are drawn with replacement. If False, sampling without replacement is performed.

bootstrap_features [boolean, optional (default=False)] Whether features are drawn with replacement.

oob_score [bool] Whether to use out-of-bag samples to estimate the generalization error.

warm_start [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new ensemble. See *the Glossary*.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both *fit* and *predict*. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

Attributes

estimators_ [list of estimators] The collection of fitted sub-estimators.

estimators_samples_ [list of arrays] The subset of drawn samples for each base estimator.

estimators_features_ [list of arrays] The subset of drawn features for each base estimator.

oob_score_ [float] Score of the training dataset obtained using an out-of-bag estimate.

oob_prediction_ [array of shape = [n_samples]] Prediction computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, `oob_prediction_` might contain NaN.

References

[R4d113ba76fc0-1], *[R4d113ba76fc0-2]*, *[R4d113ba76fc0-3]*, *[R4d113ba76fc0-4]*

Methods

fit(self, X, y[, sample_weight])	Build a Bagging ensemble of estimators from the training set (X, y).
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict regression target for X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *base_estimator=None*, *n_estimators=10*, *max_samples=1.0*, *max_features=1.0*, *bootstrap=True*, *bootstrap_features=False*, *oob_score=False*, *warm_start=False*, *n_jobs=None*, *random_state=None*, *verbose=0*)

estimators_samples_
The subset of drawn samples for each base estimator.

Returns a dynamically generated list of indices identifying the samples used for fitting each member of the ensemble, i.e., the in-bag samples.

Note: the list is re-created at each call to the property in order to reduce the object memory footprint by not storing the sampling data. Thus fetching the property may be slower than expected.

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

Build a Bagging ensemble of estimators from the training set (X, y).

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

y [array-like, shape = [n_samples]] The target values (class labels in classification, real numbers in regression).

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Note that this is supported only if the base estimator supports sample weighting.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict regression target for X.

The predicted regression target of an input sample is computed as the mean predicted regression targets of the estimators in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape = [n_samples, n_features]] The training input samples. Sparse matrices are accepted only if they are supported by the base estimator.

Returns

y [array of shape = [n_samples]] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.ensemble.BaggingRegressor`

- *Single estimator versus bagging: bias-variance decomposition*

6.12.5 `sklearn.ensemble`.IsolationForest

class `sklearn.ensemble.`**IsolationForest**(*n_estimators=100*, *max_samples='auto'*, *contamination='legacy'*, *max_features=1.0*, *bootstrap=False*, *n_jobs=None*, *behaviour='old'*, *random_state=None*, *verbose=0*, *warm_start=False*)

Isolation Forest Algorithm

Return the anomaly score of each sample using the IsolationForest algorithm

The IsolationForest 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node.

This path length, averaged over a forest of such random trees, is a measure of normality and our decision function.

Random partitioning produces noticeably shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.

Read more in the *User Guide*.

New in version 0.18.

Parameters

n_estimators [int, optional (default=100)] The number of base estimators in the ensemble.

max_samples [int or float, optional (default="auto")]

The number of samples to draw from X to train each base estimator.

- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples.
- If "auto", then `max_samples=min(256, n_samples)`.

If max_samples is larger than the number of samples provided, all samples will be used for all trees (no sampling).

contamination [float in (0., 0.5), optional (default=0.1)] The amount of contamination of the data set, i.e. the proportion of outliers in the data set. Used when fitting to define the threshold on the decision function. If 'auto', the decision function threshold is determined as in the original paper.

Changed in version 0.20: The default value of `contamination` will change from 0.1 in 0.20 to `'auto'` in 0.22.

max_features [int or float, optional (default=1.0)] The number of features to draw from X to train each base estimator.

- If int, then draw `max_features` features.
- If float, then draw `max_features * X.shape[1]` features.

bootstrap [boolean, optional (default=False)] If True, individual trees are fit on random subsets of the training data sampled with replacement. If False, sampling without replacement is performed.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both *fit* and *predict*. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

behaviour [str, default='old'] Behaviour of the `decision_function` which can be either 'old' or 'new'. Passing `behaviour='new'` makes the `decision_function` change to match other anomaly detection algorithm API which will be the default behaviour in the future. As explained in details in the `offset_` attribute documentation, the `decision_function` becomes dependent on the contamination parameter, in such a way that 0 becomes its natural threshold to detect outliers.

New in version 0.20: `behaviour` is added in 0.20 for back-compatibility purpose.

Deprecated since version 0.20: `behaviour='old'` is deprecated in 0.20 and will not be possible in 0.22.

Deprecated since version 0.22: `behaviour` parameter will be deprecated in 0.22 and removed in 0.24.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

verbose [int, optional (default=0)] Controls the verbosity of the tree building process.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See *the Glossary*.

New in version 0.21.

Attributes

estimators_ [list of DecisionTreeClassifier] The collection of fitted sub-estimators.

estimators_samples_ [list of arrays] The subset of drawn samples for each base estimator.

max_samples_ [integer] The actual number of samples

offset_ [float] Offset used to define the decision function from the raw scores. We have the relation: `decision_function = score_samples - offset_`. Assuming behaviour == 'new', `offset_` is defined as follows. When the contamination parameter is set to "auto", the offset is equal to -0.5 as the scores of inliers are close to 0 and the scores of outliers are close to -1. When a contamination parameter different than "auto" is provided, the offset is defined in such a way we obtain the expected number of outliers (samples with decision function < 0) in training. Assuming the behaviour parameter is set to 'old', we always have `offset_ = -0.5`, making the decision function independent from the contamination parameter.

Notes

The implementation is based on an ensemble of ExtraTreeRegressor. The maximum depth of each tree is set to `ceil(log_2(n))` where n is the number of samples used to build the tree (see (Liu et al., 2008) for more details).

References

[Rd7ae0a2ae688-1], *[Rd7ae0a2ae688-2]*

Methods

decision_function(self, X)	Average anomaly score of X of the base classifiers.
fit(self, X[, y, sample_weight])	Fit estimator.
fit_predict(self, X[, y])	Performs fit on X and returns labels for X.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict if a particular sample is an outlier or not.
score_samples(self, X)	Opposite of the anomaly score defined in the original paper.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_estimators=100*, *max_samples='auto'*, *contamination='legacy'*, *max_features=1.0*, *bootstrap=False*, *n_jobs=None*, *behaviour='old'*, *random_state=None*, *verbose=0*, *warm_start=False*)

decision_function(*self*, *X*)
Average anomaly score of X of the base classifiers.

The anomaly score of an input sample is computed as the mean anomaly score of the trees in the forest.

The measure of normality of an observation given a tree is the depth of the leaf containing this observation, which is equivalent to the number of splittings required to isolate this point. In case of several observations n_left in the leaf, the average path length of a n_left samples isolation tree is added.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

scores [array, shape (n_samples,)] The anomaly score of the input samples. The lower, the more abnormal. Negative scores represent outliers, positive scores represent inliers.

estimators_samples_
The subset of drawn samples for each base estimator.

Returns a dynamically generated list of indices identifying the samples used for fitting each member of the ensemble, i.e., the in-bag samples.

Note: the list is re-created at each call to the property in order to reduce the object memory footprint by not storing the sampling data. Thus fetching the property may be slower than expected.

fit(*self*, *X*, *y=None*, *sample_weight=None*)
Fit estimator.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The input samples. Use `dtype=np.float32` for maximum efficiency. Sparse matrices are also supported, use sparse `csc_matrix` for maximum efficiency.

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted.

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

Returns

self [object]

fit_predict(*self*, *X*, *y=None*)
Performs fit on X and returns labels for X.

Returns -1 for outliers and 1 for inliers.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

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

Returns

y [ndarray, shape (n_samples,)] 1 for inliers, -1 for outliers.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)

Predict if a particular sample is an outlier or not.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

is_inlier [array, shape (n_samples,)] For each observation, tells whether or not (+1 or -1) it should be considered as an inlier according to the fitted model.

score_samples(*self*, *X*)

Opposite of the anomaly score defined in the original paper.

The anomaly score of an input sample is computed as the mean anomaly score of the trees in the forest.

The measure of normality of an observation given a tree is the depth of the leaf containing this observation, which is equivalent to the number of splittings required to isolate this point. In case of several observations n_left in the leaf, the average path length of a n_left samples isolation tree is added.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The input samples.

Returns

scores [array, shape (n_samples,)] The anomaly score of the input samples. The lower, the more abnormal.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.ensemble.IsolationForest`

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *IsolationForest example*

6.12.6 sklearn.ensemble.RandomTreesEmbedding

class sklearn.ensemble.**RandomTreesEmbedding**(*n_estimators='warn'*, *max_depth=5*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *sparse_output=True*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*)

An ensemble of totally random trees.

An unsupervised transformation of a dataset to a high-dimensional sparse representation. A datapoint is coded according to which leaf of each tree it is sorted into. Using a one-hot encoding of the leaves, this leads to a binary coding with as many ones as there are trees in the forest.

The dimensionality of the resulting representation is `n_out <= n_estimators * max_leaf_nodes`. If `max_leaf_nodes == None`, the number of leaf nodes is at most `n_estimators * 2 ** max_depth`.

Read more in the *User Guide*.

Parameters

n_estimators [integer, optional (default=10)] Number of trees in the forest.

Changed in version 0.20: The default value of `n_estimators` will change from 10 in version 0.20 to 100 in version 0.22.

max_depth [integer, optional (default=5)] The maximum depth of each tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` is the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` is the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_leaf_nodes [int or None, optional (default=None)] Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

sparse_output [bool, optional (default=True)] Whether or not to return a sparse CSR matrix, as default behavior, or to return a dense array compatible with dense pipeline operators.

n_jobs [int or None, optional (default=None)] The number of jobs to run in parallel for both *fit* and *predict*. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

verbose [int, optional (default=0)] Controls the verbosity when fitting and predicting.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See *the Glossary*.

Attributes

estimators_ [list of DecisionTreeClassifier] The collection of fitted sub-estimators.

References

[R6e47e53bacbd-1], *[R6e47e53bacbd-2]*

Methods

apply(self, X)	Apply trees in the forest to X, return leaf indices.
decision_path(self, X)	Return the decision path in the forest
fit(self, X[, y, sample_weight])	Fit estimator.
fit_transform(self, X[, y, sample_weight])	Fit estimator and transform dataset.

Continued on next page

Table 6.77 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform dataset.

__init__ (*self*, *n_estimators='warn'*, *max_depth=5*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *sparse_output=True*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*)

apply (*self*, *X*)
Apply trees in the forest to X, return leaf indices.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

X_leaves [array_like, shape = [n_samples, n_estimators]] For each datapoint x in X and for each tree in the forest, return the index of the leaf x ends up in.

decision_path (*self*, *X*)
Return the decision path in the forest

New in version 0.18.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, its dtype will be converted to `dtype=np.float32`. If a sparse matrix is provided, it will be converted into a sparse `csr_matrix`.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

n_nodes_ptr [array of size (n_estimators + 1,)] The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]] gives the indicator value for the i-th estimator.

feature_importances_

Return the feature importances (the higher, the more important the feature).

Returns

feature_importances_ [array, shape = [n_features]] The values of this array sum to 1, unless all trees are single node trees consisting of only the root node, in which case it will be an array of zeros.

fit (*self*, *X*, *y=None*, *sample_weight=None*)
Fit estimator.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] The input samples. Use `dtype=np.float32` for maximum efficiency. Sparse matrices are also supported, use sparse `csc_matrix` for maximum efficiency.

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

self [object]

fit_transform(*self*, *X*, *y=None*, *sample_weight=None*)
Fit estimator and transform dataset.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Input data used to build forests. Use `dtype=np.float32` for maximum efficiency.

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. In the case of classification, splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns

X_transformed [sparse matrix, shape=(n_samples, n_out)] Transformed dataset.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform dataset.

Parameters

X [array-like or sparse matrix, shape=(n_samples, n_features)] Input data to be transformed. Use `dtype=np.float32` for maximum efficiency. Sparse matrices are also supported, use sparse `csr_matrix` for maximum efficiency.

Returns

X_transformed [sparse matrix, shape=(n_samples, n_out)] Transformed dataset.

Examples using `sklearn.ensemble.RandomTreesEmbedding`

- *Hashing feature transformation using Totally Random Trees*
- *Feature transformations with ensembles of trees*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

6.12.7 `sklearn.ensemble`.VotingClassifier

class `sklearn.ensemble.`**`VotingClassifier`**(*estimators*, *voting='hard'*, *weights=None*, *n_jobs=None*, *flatten_transform=True*)

Soft Voting/Majority Rule classifier for unfitted estimators.

New in version 0.17.

Read more in the *User Guide*.

Parameters

estimators [list of (string, estimator) tuples] Invoking the `fit` method on the `VotingClassifier` will fit clones of those original estimators that will be stored in the class attribute `self.estimators_`. An estimator can be set to `None` or `'drop'` using `set_params`.

voting [str, {'hard', 'soft'} (default='hard')] If 'hard', uses predicted class labels for majority rule voting. Else if 'soft', predicts the class label based on the argmax of the sums of the predicted probabilities, which is recommended for an ensemble of well-calibrated classifiers.

weights [array-like, shape (n_classifiers,), optional (default='None')] Sequence of weights (`float` or `int`) to weight the occurrences of predicted class labels (`hard` voting) or class probabilities before averaging (`soft` voting). Uses uniform weights if `None`.

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

flatten_transform [bool, optional (default=True)] Affects shape of transform output only when voting='soft' If voting='soft' and flatten_transform=True, transform method returns matrix with shape (n_samples, n_classifiers * n_classes). If flatten_transform=False, it returns (n_classifiers, n_samples, n_classes).

Attributes

estimators_ [list of classifiers] The collection of fitted sub-estimators as defined in `estimators` that are not `None`.

named_estimators_ [Bunch object, a dictionary with attribute access] Attribute to access any fitted sub-estimators by name.

New in version 0.20.

classes_ [array-like, shape (n_predictions,)] The classes labels.

See also:

`VotingRegressor` Prediction voting regressor.

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.ensemble import RandomForestClassifier, VotingClassifier
>>> clf1 = LogisticRegression(solver='lbfgs', multi_class='multinomial',
...                           random_state=1)
>>> clf2 = RandomForestClassifier(n_estimators=50, random_state=1)
>>> clf3 = GaussianNB()
>>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
>>> y = np.array([1, 1, 1, 2, 2, 2])
>>> eclf1 = VotingClassifier(estimators=[
...         ('lr', clf1), ('rf', clf2), ('gnb', clf3)], voting='hard')
>>> eclf1 = eclf1.fit(X, y)
>>> print(eclf1.predict(X))
[1 1 1 2 2 2]
>>> np.array_equal(eclf1.named_estimators_.lr.predict(X),
...                eclf1.named_estimators_['lr'].predict(X))
True
>>> eclf2 = VotingClassifier(estimators=[
...         ('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...         voting='soft')
>>> eclf2 = eclf2.fit(X, y)
>>> print(eclf2.predict(X))
[1 1 1 2 2 2]
>>> eclf3 = VotingClassifier(estimators=[
...        ('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...        voting='soft', weights=[2,1,1],
...        flatten_transform=True)
>>> eclf3 = eclf3.fit(X, y)
>>> print(eclf3.predict(X))
[1 1 1 2 2 2]
>>> print(eclf3.transform(X).shape)
(6, 6)
```

Methods

fit(self, X, y[, sample_weight])	Fit the estimators.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get the parameters of the ensemble estimator
predict(self, X)	Predict class labels for X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Setting the parameters for the ensemble estimator
transform(self, X)	Return class labels or probabilities for X for each estimator.

__init__(*self*, *estimators*, *voting='hard'*, *weights=None*, *n_jobs=None*, *flatten_transform=True*)

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

Fit the estimators.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

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

sample_weight [array-like, shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Note that this is supported only if all underlying estimators support sample weights.

Returns

self [object]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get the parameters of the ensemble estimator

Parameters

deep [bool] Setting it to True gets the various estimators and the parameters of the estimators as well

predict(*self*, *X*)
Predict class labels for X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples.

Returns

maj [array-like, shape (n_samples,)] Predicted class labels.

predict_proba
Compute probabilities of possible outcomes for samples in X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples.

Returns

avg [array-like, shape (n_samples, n_classes)] Weighted average probability for each class per sample.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***params*)
Setting the parameters for the ensemble estimator

Valid parameter keys can be listed with get_params().

Parameters

****params** [keyword arguments] Specific parameters using e.g. set_params(parameter_name=new_value) In addition, to setting the parameters of the ensemble estimator, the individual estimators of the ensemble estimator can also be set or replaced by setting them to None.

Examples

In this example, the RandomForestClassifier is removed clf1 = LogisticRegression() clf2 = RandomForestClassifier() eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2)] eclf.set_params(rf=None)

transform (*self*, *X*)
Return class labels or probabilities for X for each estimator.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

Returns

probabilities_or_labels

If `voting='soft'` and `flatten_transform=True`: returns array-like of shape (n_classifiers, n_samples * n_classes), being class probabilities calculated by each classifier.

If `voting='soft' and `flatten_transform=False`: array-like of shape (n_classifiers, n_samples, n_classes)

If `voting='hard'`: array-like of shape (n_samples, n_classifiers), being class labels predicted by each classifier.

Examples using `sklearn.ensemble.VotingClassifier`

- *Plot the decision boundaries of a VotingClassifier*
- *Plot class probabilities calculated by the VotingClassifier*

6.12.8 `sklearn.ensemble`.VotingRegressor

class sklearn.ensemble.**VotingRegressor** (*estimators*, *weights=None*, *n_jobs=None*)
Prediction voting regressor for unfitted estimators.

New in version 0.21.

A voting regressor is an ensemble meta-estimator that fits base regressors each on the whole dataset. It, then, averages the individual predictions to form a final prediction.

Read more in the *User Guide*.

Parameters

estimators [list of (string, estimator) tuples] Invoking the `fit` method on the `VotingRegressor` will fit clones of those original estimators that will be stored in the class attribute `self.estimators_`. An estimator can be set to `None` or `'drop'` using `set_params`.

weights [array-like, shape (n_regressors,), optional (default=‘None‘)] Sequence of weights (`float` or `int`) to weight the occurrences of predicted values before averaging. Uses uniform weights if `None`.

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

Attributes

estimators_ [list of regressors] The collection of fitted sub-estimators as defined in `estimators` that are not `None`.

named_estimators_ [Bunch object, a dictionary with attribute access] Attribute to access any fitted sub-estimators by name.

See also:

VotingClassifier Soft Voting/Majority Rule classifier.

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.ensemble import VotingRegressor
>>> r1 = LinearRegression()
>>> r2 = RandomForestRegressor(n_estimators=10, random_state=1)
>>> X = np.array([[1, 1], [2, 4], [3, 9], [4, 16], [5, 25], [6, 36]])
>>> y = np.array([2, 6, 12, 20, 30, 42])
>>> er = VotingRegressor([('lr', r1), ('rf', r2)])
>>> print(er.fit(X, y).predict(X))
[ 3.3  5.7 11.8 19.7 28.  40.3]
```

Methods

fit(self, X, y[, sample_weight])	Fit the estimators.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get the parameters of the ensemble estimator
predict(self, X)	Predict regression target for X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Setting the parameters for the ensemble estimator

Continued on next page

Table 6.79 – continued from previous page

transform(self, X)	Return predictions for X for each estimator.

__init__ (*self*, *estimators*, *weights=None*, *n_jobs=None*)

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the estimators.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features.

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

sample_weight [array-like, shape (n_samples,) or None] Sample weights. If None, then samples are equally weighted. Note that this is supported only if all underlying estimators support sample weights.

Returns

self [object]

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get the parameters of the ensemble estimator

Parameters

deep [bool] Setting it to True gets the various estimators and the parameters of the estimators as well

predict (*self*, *X*)
Predict regression target for X.

The predicted regression target of an input sample is computed as the mean predicted regression targets of the estimators in the ensemble.

Parameters

X [{array-like, sparse matrix} of shape (n_samples, n_features)] The input samples.

Returns

y [array of shape (n_samples,)] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *`metrics.r2_score`*. This will influence the `score` method of all the multioutput regressors (except for *`multioutput.MultiOutputRegressor`*). To specify the default value manually and avoid the warning, please either call *`metrics.r2_score`* directly or make a custom scorer with *`metrics.make_scorer`* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***params*)

Setting the parameters for the ensemble estimator

Valid parameter keys can be listed with get_params().

Parameters

****params** [keyword arguments] Specific parameters using e.g. set_params(parameter_name=new_value) In addition, to setting the parameters of the ensemble estimator, the individual estimators of the ensemble estimator can also be set or replaced by setting them to None.

Examples

In this example, the RandomForestClassifier is removed clf1 = LogisticRegression() clf2 = RandomForestClassifier() eclf = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2)] eclf.set_params(rf=None)

transform(*self*, *X*)

Return predictions for X for each estimator.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input samples.

Returns

predictions array-like of shape (n_samples, n_classifiers), being values predicted by each regressor.

Examples using `sklearn.ensemble.VotingRegressor`

- *Plot individual and voting regression predictions*

6.12.9 sklearn.ensemble.HistGradientBoostingRegressor

class sklearn.ensemble.**HistGradientBoostingRegressor**(*loss='least_squares'*, *learning_rate=0.1*, *max_iter=100*, *max_leaf_nodes=31*, *max_depth=None*, *min_samples_leaf=20*, *l2_regularization=0.0*, *max_bins=256*, *scoring=None*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=1e-07*, *verbose=0*, *random_state=None*)

Histogram-based Gradient Boosting Regression Tree.

This estimator is much faster than *GradientBoostingRegressor* for big datasets (n_samples >= 10 000). The input data `X` is pre-binned into integer-valued bins, which considerably reduces the number of splitting points to consider, and allows the algorithm to leverage integer-based data structures. For small sample sizes, *GradientBoostingRegressor* might be preferred since binning may lead to split points that are too approximate in this setting.

This implementation is inspired by LightGBM.

Note: This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import `enable_hist_gradient_boosting`:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> # now you can import normally from ensemble
>>> from sklearn.ensemble import HistGradientBoostingClassifier
```

Parameters

loss [{'least_squares'}, optional (default='least_squares')] The loss function to use in the boosting process. Note that the "least squares" loss actually implements an "half least squares loss" to simplify the computation of the gradient.

learning_rate [float, optional (default=0.1)] The learning rate, also known as *shrinkage*. This is used as a multiplicative factor for the leaves values. Use `1` for no shrinkage.

max_iter [int, optional (default=100)] The maximum number of iterations of the boosting process, i.e. the maximum number of trees.

max_leaf_nodes [int or None, optional (default=31)] The maximum number of leaves for each tree. Must be strictly greater than 1. If None, there is no maximum limit.

max_depth [int or None, optional (default=None)] The maximum depth of each tree. The depth of a tree is the number of nodes to go from the root to the deepest leaf. Must be strictly greater than 1. Depth isn't constrained by default.

min_samples_leaf [int, optional (default=20)] The minimum number of samples per leaf. For small datasets with less than a few hundred samples, it is recommended to lower this value since only very shallow trees would be built.

l2_regularization [float, optional (default=0)] The L2 regularization parameter. Use `0` for no regularization (default).

max_bins [int, optional (default=256)] The maximum number of bins to use. Before training, each feature of the input array `X` is binned into at most `max_bins` bins, which allows for a much faster training stage. Features with a small number of unique values may use less than `max_bins` bins. Must be no larger than 256.

scoring [str or callable or None, optional (default=None)] Scoring parameter to use for early stopping. It can be a single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*). If None, the estimator's default scorer is used. If `scoring='loss'`, early stopping is checked w.r.t the loss value. Only used if `n_iter_no_change` is not None.

validation_fraction [int or float or None, optional (default=0.1)] Proportion (or absolute size) of training data to set aside as validation data for early stopping. If None, early stopping is done on the training data. Only used if `n_iter_no_change` is not None.

n_iter_no_change [int or None, optional (default=None)] Used to determine when to "early stop". The fitting process is stopped when none of the last `n_iter_no_change` scores are better than the ‘‘n_iter_no_change - 1‘‘th-to-last one, up to some tolerance. If None or 0, no early-stopping is done.

tol [float or None, optional (default=1e-7)] The absolute tolerance to use when comparing scores during early stopping. The higher the tolerance, the more likely we are to early stop: higher tolerance means that it will be harder for subsequent iterations to be considered an improvement upon the reference score.

verbose: int, optional (default=0) The verbosity level. If not zero, print some information about the fitting process.

random_state [int, np.random.RandomStateInstance or None, optional (default=None)] Pseudo-random number generator to control the subsampling in the binning process, and the train/validation data split if early stopping is enabled. See *random_state*.

Attributes

n_iter_ [int] The number of iterations as selected by early stopping (if n_iter_no_change is not None). Otherwise it corresponds to max_iter.

n_trees_per_iteration_ [int] The number of tree that are built at each iteration. For regressors, this is always 1.

train_score_ [ndarray, shape (max_iter + 1,)] The scores at each iteration on the training data. The first entry is the score of the ensemble before the first iteration. Scores are computed according to the `scoring` parameter. If `scoring` is not 'loss', scores are computed on a subset of at most 10 000 samples. Empty if no early stopping.

validation_score_ [ndarray, shape (max_iter + 1,)] The scores at each iteration on the held-out validation data. The first entry is the score of the ensemble before the first iteration. Scores are computed according to the `scoring` parameter. Empty if no early stopping or if `validation_fraction` is None.

Examples

```
>>> # To use this experimental feature, we need to explicitly ask for it:
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> from sklearn.ensemble import HistGradientBoostingRegressor
>>> from sklearn.datasets import load_boston
>>> X, y = load_boston(return_X_y=True)
>>> est = HistGradientBoostingRegressor().fit(X, y)
```

```
>>> est.score(X, y)
0.98...
```

Methods

fit(self, X, y)	Fit the gradient boosting model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict values for X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *loss='least_squares'*, *learning_rate=0.1*, *max_iter=100*, *max_leaf_nodes=31*, *max_depth=None*, *min_samples_leaf=20*, *l2_regularization=0.0*, *max_bins=256*, *scoring=None*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=1e-07*, *verbose=0*, *random_state=None*)

fit (*self*, *X*, *y*)
Fit the gradient boosting model.

Parameters

X [array-like, shape=(n_samples, n_features)] The input samples.

y [array-like, shape=(n_samples,)] Target values.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict values for X.

Parameters

X [array-like, shape (n_samples, n_features)] The input samples.

Returns

y [ndarray, shape (n_samples,)] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

6.12.10 sklearn.ensemble.HistGradientBoostingClassifier

class sklearn.ensemble.**HistGradientBoostingClassifier**(*loss='auto'*, *learning_rate=0.1*, *max_iter=100*, *max_leaf_nodes=31*, *max_depth=None*, *min_samples_leaf=20*, *l2_regularization=0.0*, *max_bins=256*, *scoring=None*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=1e-07*, *verbose=0*, *random_state=None*)

Histogram-based Gradient Boosting Classification Tree.

This estimator is much faster than *GradientBoostingClassifier* for big datasets (n_samples >= 10 000). The input data `X` is pre-binned into integer-valued bins, which considerably reduces the number of splitting points to consider, and allows the algorithm to leverage integer-based data structures. For small sample sizes, *GradientBoostingClassifier* might be preferred since binning may lead to split points that are too approximate in this setting.

This implementation is inspired by LightGBM.

Note: This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import `enable_hist_gradient_boosting`:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> # now you can import normally from ensemble
>>> from sklearn.ensemble import HistGradientBoostingClassifier
```

Parameters

loss [{'auto', 'binary_crossentropy', 'categorical_crossentropy'}, optional (default='auto')] The loss function to use in the boosting process. 'binary_crossentropy' (also known as logistic loss) is used for binary classification and generalizes to 'categorical_crossentropy' for multiclass classification. 'auto' will automatically choose either loss depending on the nature of the problem.

learning_rate [float, optional (default=0.1)] The learning rate, also known as *shrinkage*. This is used as a multiplicative factor for the leaves values. Use `1` for no shrinkage.

max_iter [int, optional (default=100)] The maximum number of iterations of the boosting process, i.e. the maximum number of trees for binary classification. For multiclass classification, `n_classes` trees per iteration are built.

max_leaf_nodes [int or None, optional (default=31)] The maximum number of leaves for each tree. Must be strictly greater than 1. If None, there is no maximum limit.

max_depth [int or None, optional (default=None)] The maximum depth of each tree. The depth of a tree is the number of nodes to go from the root to the deepest leaf. Must be strictly greater than 1. Depth isn't constrained by default.

min_samples_leaf [int, optional (default=20)] The minimum number of samples per leaf. For small datasets with less than a few hundred samples, it is recommended to lower this value since only very shallow trees would be built.

l2_regularization [float, optional (default=0)] The L2 regularization parameter. Use 0 for no regularization.

max_bins [int, optional (default=256)] The maximum number of bins to use. Before training, each feature of the input array `X` is binned into at most `max_bins` bins, which allows for a much faster training stage. Features with a small number of unique values may use less than `max_bins` bins. Must be no larger than 256.

scoring [str or callable or None, optional (default=None)] Scoring parameter to use for early stopping. It can be a single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*). If None, the estimator's default scorer is used. If `scoring='loss'`, early stopping is checked w.r.t the loss value. Only used if `n_iter_no_change` is not None.

validation_fraction [int or float or None, optional (default=0.1)] Proportion (or absolute size) of training data to set aside as validation data for early stopping. If None, early stopping is done on the training data.

n_iter_no_change [int or None, optional (default=None)] Used to determine when to "early stop". The fitting process is stopped when none of the last `n_iter_no_change` scores are better than the ''n_iter_no_change - 1''th-to-last one, up to some tolerance. If None or 0, no early-stopping is done.

tol [float or None, optional (default=1e-7)] The absolute tolerance to use when comparing scores. The higher the tolerance, the more likely we are to early stop: higher tolerance means that it will be harder for subsequent iterations to be considered an improvement upon the reference score.

verbose: int, optional (default=0) The verbosity level. If not zero, print some information about the fitting process.

random_state [int, np.random.RandomStateInstance or None, optional (default=None)] Pseudo-random number generator to control the subsampling in the binning process, and the train/validation data split if early stopping is enabled. See *random_state*.

Attributes

n_iter_ [int] The number of estimators as selected by early stopping (if n_iter_no_change is not None). Otherwise it corresponds to max_iter.

n_trees_per_iteration_ [int] The number of tree that are built at each iteration. This is equal to 1 for binary classification, and to `n_classes` for multiclass classification.

train_score_ [ndarray, shape (max_iter + 1,)] The scores at each iteration on the training data. The first entry is the score of the ensemble before the first iteration. Scores are computed according to the `scoring` parameter. If `scoring` is not 'loss', scores are computed on a subset of at most 10 000 samples. Empty if no early stopping.

validation_score_ [ndarray, shape (max_iter + 1,)] The scores at each iteration on the held-out validation data. The first entry is the score of the ensemble before the first iteration. Scores are computed according to the `scoring` parameter. Empty if no early stopping or if `validation_fraction` is None.

Examples

```
>>> # To use this experimental feature, we need to explicitly ask for it:
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> from sklearn.ensemble import HistGradientBoostingRegressor
>>> from sklearn.datasets import load_iris
>>> X, y = load_iris(return_X_y=True)
>>> clf = HistGradientBoostingClassifier().fit(X, y)
>>> clf.score(X, y)
1.0
```

Methods

decision_function(self, X)	Compute the decision function of X.
fit(self, X, y)	Fit the gradient boosting model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict classes for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *loss='auto'*, *learning_rate=0.1*, *max_iter=100*, *max_leaf_nodes=31*, *max_depth=None*, *min_samples_leaf=20*, *l2_regularization=0.0*, *max_bins=256*, *scoring=None*, *validation_fraction=0.1*, *n_iter_no_change=None*, *tol=1e-07*, *verbose=0*, *random_state=None*)

decision_function(*self*, *X*)

Compute the decision function of X.

Parameters

X [array-like, shape (n_samples, n_features)] The input samples.

Returns

decision [ndarray, shape (n_samples,) or (n_samples, n_trees_per_iteration)] The raw predicted values (i.e. the sum of the trees leaves) for each sample. n_trees_per_iteration is equal to the number of classes in multiclass classification.

fit(*self*, *X*, *y*)

Fit the gradient boosting model.

Parameters

X [array-like, shape=(n_samples, n_features)] The input samples.

y [array-like, shape=(n_samples,)] Target values.

Returns

self [object]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)

Predict classes for X.

Parameters

X [array-like, shape (n_samples, n_features)] The input samples.

Returns

y [ndarray, shape (n_samples,)] The predicted classes.

predict_proba(*self*, *X*)

Predict class probabilities for X.

Parameters

X [array-like, shape (n_samples, n_features)] The input samples.

Returns

p [ndarray, shape (n_samples, n_classes)] The class probabilities of the input samples.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

—

6.13 sklearn.exceptions: Exceptions and warnings

The *sklearn.exceptions* module includes all custom warnings and error classes used across scikit-learn.

exceptions.ChangedBehaviorWarning	Warning class used to notify the user of any change in the behavior.
exceptions.ConvergenceWarning	Custom warning to capture convergence problems
exceptions.DataConversionWarning	Warning used to notify implicit data conversions happening in the code.
exceptions.DataDimensionalityWarning	Custom warning to notify potential issues with data dimensionality.
exceptions.EfficiencyWarning	Warning used to notify the user of inefficient computation.
exceptions.FitFailedWarning	Warning class used if there is an error while fitting the estimator.
exceptions.NotFittedError	Exception class to raise if estimator is used before fitting.
exceptions.NonBLASDotWarning	Warning used when the dot operation does not use BLAS.
exceptions.UndefinedMetricWarning	Warning used when the metric is invalid

6.13.1 sklearn.exceptions.ChangedBehaviorWarning

class sklearn.exceptions.**ChangedBehaviorWarning**
Warning class used to notify the user of any change in the behavior.

Changed in version 0.18: Moved from sklearn.base.

Attributes

args

Methods

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

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

6.13.2 sklearn.exceptions.ConvergenceWarning

class sklearn.exceptions.**ConvergenceWarning**
Custom warning to capture convergence problems

Changed in version 0.18: Moved from sklearn.utils.

Attributes

args

Methods

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

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

Examples using sklearn.exceptions.ConvergenceWarning

- *Multiclass sparse logisitic regression on newgroups20*
- *Early stopping of Stochastic Gradient Descent*
- *Compare Stochastic learning strategies for MLPClassifier*
- *Feature discretization*

6.13.3 sklearn.exceptions.DataConversionWarning

class sklearn.exceptions.**DataConversionWarning**
Warning used to notify implicit data conversions happening in the code.

This warning occurs when some input data needs to be converted or interpreted in a way that may not match the user's expectations.

For example, this warning may occur when the user

- passes an integer array to a function which expects float input and will convert the input

- requests a non-copying operation, but a copy is required to meet the implementation's data-type expectations;
- passes an input whose shape can be interpreted ambiguously.

Changed in version 0.18: Moved from sklearn.utils.validation.

Attributes

args

Methods

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

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

6.13.4 sklearn.exceptions.DataDimensionalityWarning

class sklearn.exceptions.**DataDimensionalityWarning**
Custom warning to notify potential issues with data dimensionality.

For example, in random projection, this warning is raised when the number of components, which quantifies the dimensionality of the target projection space, is higher than the number of features, which quantifies the dimensionality of the original source space, to imply that the dimensionality of the problem will not be reduced.

Changed in version 0.18: Moved from sklearn.utils.

Attributes

args

Methods

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

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

6.13.5 sklearn.exceptions.EfficiencyWarning

class sklearn.exceptions.**EfficiencyWarning**
Warning used to notify the user of inefficient computation.

This warning notifies the user that the efficiency may not be optimal due to some reason which may be included as a part of the warning message. This may be subclassed into a more specific Warning class.

New in version 0.18.

Attributes

args

Methods

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

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

6.13.6 sklearn.exceptions.FitFailedWarning

class sklearn.exceptions.**FitFailedWarning**
Warning class used if there is an error while fitting the estimator.

This Warning is used in meta estimators GridSearchCV and RandomizedSearchCV and the cross-validation helper function cross_val_score to warn when there is an error while fitting the estimator.

Attributes

args

Examples

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> from sklearn.exceptions import FitFailedWarning
>>> import warnings
>>> warnings.simplefilter('always', FitFailedWarning)
>>> gs = GridSearchCV(LinearSVC(), {'C': [-1, -2]}, error_score=0, cv=2)
>>> X, y = [[1, 2], [3, 4], [5, 6], [7, 8]], [0, 0, 1, 1]
>>> with warnings.catch_warnings(record=True) as w:
...     try:
...         gs.fit(X, y)  # This will raise a ValueError since C is < 0
...     except ValueError:
...         pass
...     print(repr(w[-1].message))
...
FitFailedWarning('Estimator fit failed. The score on this train-test
partition for these parameters will be set to 0.000000.
Details: \nValueError: Penalty term must be positive; got (C=-2)\n'...)
```

Changed in version 0.18: Moved from sklearn.cross_validation.

Methods

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

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

6.13.7 sklearn.exceptions.NotFittedError

class sklearn.exceptions.**NotFittedError**
Exception class to raise if estimator is used before fitting.

This class inherits from both ValueError and AttributeError to help with exception handling and backward compatibility.

Attributes

args

Examples

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.exceptions import NotFittedError
>>> try:
...     LinearSVC().predict([[1, 2], [2, 3], [3, 4]])
... except NotFittedError as e:
...     print(repr(e))
...
NotFittedError('This LinearSVC instance is not fitted yet'...)
```

Changed in version 0.18: Moved from sklearn.utils.validation.

Methods

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

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

6.13.8 sklearn.exceptions.NonBLASDotWarning

class sklearn.exceptions.**NonBLASDotWarning**
Warning used when the dot operation does not use BLAS.

This warning is used to notify the user that BLAS was not used for dot operation and hence the efficiency may be affected.

Changed in version 0.18: Moved from sklearn.utils.validation, extends EfficiencyWarning.

Attributes

args

Methods

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

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

6.13.9 sklearn.exceptions.UndefinedMetricWarning

class sklearn.exceptions.**UndefinedMetricWarning**
Warning used when the metric is invalid

Changed in version 0.18: Moved from sklearn.base.

Attributes

args

Methods

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

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

6.14 sklearn.experimental: Experimental

The *sklearn.experimental* module provides importable modules that enable the use of experimental features or estimators.

The features and estimators that are experimental aren't subject to deprecation cycles. Use them at your own risks!

experimental.enable_hist_gradient_boosting	Enables histogram-based gradient boosting estimators.
experimental.enable_iterative_imputer	Enables IterativeImputer

6.14.1 sklearn.experimental.enable_hist_gradient_boosting

Enables histogram-based gradient boosting estimators.

The API and results of these estimators might change without any deprecation cycle.

Importing this file dynamically sets the *sklearn.ensemble.HistGradientBoostingClassifier* and *sklearn.ensemble.HistGradientBoostingRegressor* as attributes of the ensemble module:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_hist_gradient_boosting  # noqa
>>> # now you can import normally from ensemble
```

```
>>> from sklearn.ensemble import HistGradientBoostingClassifier
>>> from sklearn.ensemble import HistGradientBoostingRegressor
```

The `# noqa` comment comment can be removed: it just tells linters like flake8 to ignore the import, which appears as unused.

6.14.2 sklearn.experimental.enable_iterative_imputer

Enables IterativeImputer

The API and results of this estimator might change without any deprecation cycle.

Importing this file dynamically sets *sklearn.impute.IterativeImputer* as an attribute of the impute module:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_iterative_imputer  # noqa
>>> # now you can import normally from impute
>>> from sklearn.impute import IterativeImputer
```

6.15 sklearn.feature_extraction: Feature Extraction

The *sklearn.feature_extraction* module deals with feature extraction from raw data. It currently includes methods to extract features from text and images.

User guide: See the *Feature extraction* section for further details.

feature_extraction.DictVectorizer([dtype, ...])	Transforms lists of feature-value mappings to vectors.
feature_extraction.FeatureHasher([...])	Implements feature hashing, aka the hashing trick.

6.15.1 sklearn.feature_extraction.DictVectorizer

class sklearn.feature_extraction.**DictVectorizer**(*dtype=<class 'numpy.float64'>*, *separator='='*, *sparse=True*, *sort=True*)

Transforms lists of feature-value mappings to vectors.

This transformer turns lists of mappings (dict-like objects) of feature names to feature values into Numpy arrays or scipy.sparse matrices for use with scikit-learn estimators.

When feature values are strings, this transformer will do a binary one-hot (aka one-of-K) coding: one boolean-valued feature is constructed for each of the possible string values that the feature can take on. For instance, a feature "f" that can take on the values "ham" and "spam" will become two features in the output, one signifying "f=ham", the other "f=spam".

However, note that this transformer will only do a binary one-hot encoding when feature values are of type string. If categorical features are represented as numeric values such as int, the DictVectorizer can be followed by *sklearn.preprocessing.OneHotEncoder* to complete binary one-hot encoding.

Features that do not occur in a sample (mapping) will have a zero value in the resulting array/matrix.

Read more in the *User Guide*.

Parameters

dtype [callable, optional] The type of feature values. Passed to Numpy array/scipy.sparse matrix constructors as the dtype argument.

separator [string, optional] Separator string used when constructing new features for one-hot coding.

sparse [boolean, optional.] Whether transform should produce scipy.sparse matrices. True by default.

sort [boolean, optional.] Whether `feature_names_` and `vocabulary_` should be sorted when fitting. True by default.

Attributes

vocabulary_ [dict] A dictionary mapping feature names to feature indices.

feature_names_ [list] A list of length n_features containing the feature names (e.g., "f=ham" and "f=spam").

See also:

FeatureHasher performs vectorization using only a hash function.

sklearn.preprocessing.OrdinalEncoder handles nominal/categorical features encoded as columns of arbitrary data types.

Examples

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> v = DictVectorizer(sparse=False)
>>> D = [{'foo': 1, 'bar': 2}, {'foo': 3, 'baz': 1}]
>>> X = v.fit_transform(D)
>>> X
array([[2., 0., 1.],
       [0., 1., 3.]])
>>> v.inverse_transform(X) ==         [{'bar': 2.0, 'foo': 1.0}, {'baz': 1.0, 'foo
↪': 3.0}]
True
>>> v.transform({'foo': 4, 'unseen_feature': 3})
array([[0., 0., 4.]])
```

Methods

fit(self, X[, y])	Learn a list of feature name -> indices mappings.
fit_transform(self, X[, y])	Learn a list of feature name -> indices mappings and transform X.
get_feature_names(self)	Returns a list of feature names, ordered by their indices.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X[, dict_type])	Transform array or sparse matrix X back to feature mappings.
restrict(self, support[, indices])	Restrict the features to those in support using feature selection.
set_params(self, **params)	Set the parameters of this estimator.

Continued on next page

Table 6.95 – continued from previous page

transform(self, X)	Transform feature->value dicts to array or sparse matrix.

__init__ (*self*, *dtype=<class 'numpy.float64'>*, *separator='='*, *sparse=True*, *sort=True*)

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

Learn a list of feature name -> indices mappings.

Parameters

X [Mapping or iterable over Mappings] Dict(s) or Mapping(s) from feature names (arbitrary Python objects) to feature values (strings or convertible to dtype).

y [(ignored)]

Returns

self

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

Learn a list of feature name -> indices mappings and transform X.

Like fit(X) followed by transform(X), but does not require materializing X in memory.

Parameters

X [Mapping or iterable over Mappings] Dict(s) or Mapping(s) from feature names (arbitrary Python objects) to feature values (strings or convertible to dtype).

y [(ignored)]

Returns

Xa [{array, sparse matrix}] Feature vectors; always 2-d.

get_feature_names (*self*)

Returns a list of feature names, ordered by their indices.

If one-of-K coding is applied to categorical features, this will include the constructed feature names but not the original ones.

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform (*self*, *X*, *dict_type=<class 'dict'>*)

Transform array or sparse matrix X back to feature mappings.

X must have been produced by this DictVectorizer's transform or fit_transform method; it may only have passed through transformers that preserve the number of features and their order.

In the case of one-hot/one-of-K coding, the constructed feature names and values are returned rather than the original ones.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Sample matrix.

dict_type [callable, optional] Constructor for feature mappings. Must conform to the collections.Mapping API.

Returns

D [list of dict_type objects, length = n_samples] Feature mappings for the samples in X.

restrict(*self*, *support*, *indices=False*)

Restrict the features to those in support using feature selection.

This function modifies the estimator in-place.

Parameters

support [array-like] Boolean mask or list of indices (as returned by the get_support member of feature selectors).

indices [boolean, optional] Whether support is a list of indices.

Returns

self

Examples

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> from sklearn.feature_selection import SelectKBest, chi2
>>> v = DictVectorizer()
>>> D = [{'foo': 1, 'bar': 2}, {'foo': 3, 'baz': 1}]
>>> X = v.fit_transform(D)
>>> support = SelectKBest(chi2, k=2).fit(X, [0, 1])
>>> v.get_feature_names()
['bar', 'baz', 'foo']
>>> v.restrict(support.get_support())
...
DictVectorizer(dtype=..., separator='=', sort=True,
        sparse=True)
>>> v.get_feature_names()
['bar', 'foo']
```

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)

Transform feature->value dicts to array or sparse matrix.

Named features not encountered during fit or fit_transform will be silently ignored.

Parameters

X [Mapping or iterable over Mappings, length = n_samples] Dict(s) or Mapping(s) from feature names (arbitrary Python objects) to feature values (strings or convertible to dtype).

Returns

Xa [{array, sparse matrix}] Feature vectors; always 2-d.

Examples using `sklearn.feature_extraction.DictVectorizer`

- *Column Transformer with Heterogeneous Data Sources*
- *FeatureHasher and DictVectorizer Comparison*

6.15.2 `sklearn.feature_extraction`.FeatureHasher

class `sklearn.feature_extraction.`**`FeatureHasher`**(*n_features=1048576*, *input_type='dict'*, *dtype=<class 'numpy.float64'>*, *alternate_sign=True*)

Implements feature hashing, aka the hashing trick.

This class turns sequences of symbolic feature names (strings) into scipy.sparse matrices, using a hash function to compute the matrix column corresponding to a name. The hash function employed is the signed 32-bit version of Murmurhash3.

Feature names of type byte string are used as-is. Unicode strings are converted to UTF-8 first, but no Unicode normalization is done. Feature values must be (finite) numbers.

This class is a low-memory alternative to DictVectorizer and CountVectorizer, intended for large-scale (online) learning and situations where memory is tight, e.g. when running prediction code on embedded devices.

Read more in the *User Guide*.

Parameters

n_features [integer, optional] The number of features (columns) in the output matrices. Small numbers of features are likely to cause hash collisions, but large numbers will cause larger coefficient dimensions in linear learners.

input_type [string, optional, default "dict"] Either "dict" (the default) to accept dictionaries over (feature_name, value); "pair" to accept pairs of (feature_name, value); or "string" to accept single strings. feature_name should be a string, while value should be a number. In the case of "string", a value of 1 is implied. The feature_name is hashed to find the appropriate column for the feature. The value's sign might be flipped in the output (but see non_negative, below).

dtype [numpy type, optional, default np.float64] The type of feature values. Passed to scipy.sparse matrix constructors as the dtype argument. Do not set this to bool, np.boolean or any unsigned integer type.

alternate_sign [boolean, optional, default True] When True, an alternating sign is added to the features as to approximately conserve the inner product in the hashed space even for small n_features. This approach is similar to sparse random projection.

See also:

`DictVectorizer` vectorizes string-valued features using a hash table.

`sklearn.preprocessing.OneHotEncoder` handles nominal/categorical features.

Examples

```
>>> from sklearn.feature_extraction import FeatureHasher
>>> h = FeatureHasher(n_features=10)
>>> D = [{'dog': 1, 'cat':2, 'elephant':4},{'dog': 2, 'run': 5}]
>>> f = h.transform(D)
>>> f.toarray()
array([[ 0.,  0., -4., -1.,  0.,  0.,  0.,  0.,  0.,  2.],
       [ 0.,  0.,  0., -2., -5.,  0.,  0.,  0.,  0.,  0.]])
```

Methods

fit(self[, X, y])	No-op.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, raw_X)	Transform a sequence of instances to a scipy.sparse matrix.

__init__(*self*, *n_features=1048576*, *input_type='dict'*, *dtype=<class 'numpy.float64'>*, *alternate_sign=True*)

fit(*self*, *X=None*, *y=None*)
No-op.

This method doesn't do anything. It exists purely for compatibility with the scikit-learn transformer API.

Parameters

X [array-like]

Returns

self [FeatureHasher]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *raw_X*)
Transform a sequence of instances to a scipy.sparse matrix.

Parameters

raw_X [iterable over iterable over raw features, length = n_samples] Samples. Each sample must be iterable an (e.g., a list or tuple) containing/generating feature names (and optionally values, see the input_type constructor argument) which will be hashed. raw_X need not support the len function, so it can be the result of a generator; n_samples is determined on the fly.

Returns

X [scipy.sparse matrix, shape = (n_samples, self.n_features)] Feature matrix, for use with estimators or further transformers.

Examples using `sklearn.feature_extraction.FeatureHasher`

- *FeatureHasher and DictVectorizer Comparison*

6.15.3 From images

The *sklearn.feature_extraction.image* submodule gathers utilities to extract features from images.

feature_extraction.image.extract_patches_2d(...)	Reshape a 2D image into a collection of patches
feature_extraction.image.grid_to_graph(n_x, n_y)	Graph of the pixel-to-pixel connections
feature_extraction.image.img_to_graph(img[, ...])	Graph of the pixel-to-pixel gradient connections
feature_extraction.image.reconstruct_from_patches_2d(...)	Reconstruct the image from all of its patches.
feature_extraction.image.PatchExtractor([...])	Extracts patches from a collection of images

sklearn.feature_extraction.image.extract_patches_2d

sklearn.feature_extraction.image.**extract_patches_2d**(*image*, *patch_size*, *max_patches=None*, *random_state=None*)

Reshape a 2D image into a collection of patches

The resulting patches are allocated in a dedicated array.

Read more in the *User Guide*.

Parameters

image [array, shape = (image_height, image_width) or] (image_height, image_width, n_channels) The original image data. For color images, the last dimension specifies the channel: a RGB image would have `n_channels=3`.

patch_size [tuple of ints (patch_height, patch_width)] the dimensions of one patch

max_patches [integer or float, optional default is None] The maximum number of patches to extract. If max_patches is a float between 0 and 1, it is taken to be a proportion of the total number of patches.

random_state [int, RandomState instance or None, optional (default=None)] Pseudo number generator state used for random sampling to use if `max_patches` is not None. 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`.

Returns

patches [array, shape = (n_patches, patch_height, patch_width) or] (n_patches, patch_height, patch_width, n_channels) The collection of patches extracted from the image, where `n_patches` is either `max_patches` or the total number of patches that can be extracted.

Examples

```
>>> from sklearn.datasets import load_sample_image
>>> from sklearn.feature_extraction import image
>>> # Use the array data from the first image in this dataset:
>>> one_image = load_sample_image("china.jpg")
>>> print('Image shape: {}'.format(one_image.shape))
Image shape: (427, 640, 3)
>>> patches = image.extract_patches_2d(one_image, (2, 2))
>>> print('Patches shape: {}'.format(patches.shape))
Patches shape: (272214, 2, 2, 3)
>>> # Here are just two of these patches:
>>> print(patches[1])
[[[174 201 231]
  [174 201 231]]
 [[173 200 230]
  [173 200 230]]]
>>> print(patches[800])
[[[187 214 243]
  [188 215 244]]
 [[187 214 243]
  [188 215 244]]]
```

Examples using `sklearn.feature_extraction.image.extract_patches_2d`

- *Online learning of a dictionary of parts of faces*
- *Image denoising using dictionary learning*

sklearn.feature_extraction.image.grid_to_graph

sklearn.feature_extraction.image.**grid_to_graph**(*n_x*, *n_y*, *n_z=1*, *mask=None*, *return_as=<class 'scipy.sparse.coo.coo_matrix'>*, *dtype=<class 'int'>*)

Graph of the pixel-to-pixel connections

Edges exist if 2 voxels are connected.

Parameters

n_x [int] Dimension in x axis

n_y [int] Dimension in y axis

n_z [int, optional, default 1] Dimension in z axis

mask [ndarray of booleans, optional] An optional mask of the image, to consider only part of the pixels.

return_as [np.ndarray or a sparse matrix class, optional] The class to use to build the returned adjacency matrix.

dtype [dtype, optional, default int] The data of the returned sparse matrix. By default it is int

Notes

For scikit-learn versions 0.14.1 and prior, return_as=np.ndarray was handled by returning a dense np.matrix instance. Going forward, np.ndarray returns an np.ndarray, as expected.

For compatibility, user code relying on this method should wrap its calls in `np.asarray` to avoid type issues.

sklearn.feature_extraction.image.img_to_graph

sklearn.feature_extraction.image.**img_to_graph**(*img*, *mask=None*, *return_as=<class 'scipy.sparse.coo.coo_matrix'>*, *dtype=None*)

Graph of the pixel-to-pixel gradient connections

Edges are weighted with the gradient values.

Read more in the *User Guide*.

Parameters

img [ndarray, 2D or 3D] 2D or 3D image

mask [ndarray of booleans, optional] An optional mask of the image, to consider only part of the pixels.

return_as [np.ndarray or a sparse matrix class, optional] The class to use to build the returned adjacency matrix.

dtype [None or dtype, optional] The data of the returned sparse matrix. By default it is the dtype of img

Notes

For scikit-learn versions 0.14.1 and prior, return_as=np.ndarray was handled by returning a dense np.matrix instance. Going forward, np.ndarray returns an np.ndarray, as expected.

For compatibility, user code relying on this method should wrap its calls in `np.asarray` to avoid type issues.

sklearn.feature_extraction.image.reconstruct_from_patches_2d

sklearn.feature_extraction.image.**reconstruct_from_patches_2d**(*patches*, *image_size*)

Reconstruct the image from all of its patches.

Patches are assumed to overlap and the image is constructed by filling in the patches from left to right, top to bottom, averaging the overlapping regions.

Read more in the *User Guide*.

Parameters

patches [array, shape = (n_patches, patch_height, patch_width) or] (n_patches, patch_height, patch_width, n_channels) The complete set of patches. If the patches contain colour information, channels are indexed along the last dimension: RGB patches would have `n_channels=3`.

image_size [tuple of ints (image_height, image_width) or] (image_height, image_width, n_channels) the size of the image that will be reconstructed

Returns

image [array, shape = image_size] the reconstructed image

Examples using sklearn.feature_extraction.image.reconstruct_from_patches_2d

- *Image denoising using dictionary learning*

sklearn.feature_extraction.image.PatchExtractor

class sklearn.feature_extraction.image.**PatchExtractor**(*patch_size=None*, *max_patches=None*, *random_state=None*)

Extracts patches from a collection of images

Read more in the *User Guide*.

Parameters

patch_size [tuple of ints (patch_height, patch_width)] the dimensions of one patch

max_patches [integer or float, optional default is None] The maximum number of patches per image to extract. If max_patches is a float in (0, 1), it is taken to mean a proportion of the total number of patches.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Examples

```
>>> from sklearn.datasets import load_sample_images
>>> from sklearn.feature_extraction import image
>>> # Use the array data from the second image in this dataset:
>>> X = load_sample_images().images[1]
>>> print('Image shape: {}'.format(X.shape))
Image shape: (427, 640, 3)
>>> pe = image.PatchExtractor(patch_size=(2, 2))
>>> pe_fit = pe.fit(X)
>>> pe_trans = pe.transform(X)
>>> print('Patches shape: {}'.format(pe_trans.shape))
Patches shape: (545706, 2, 2)
```

Methods

fit(self, X[, y])	Do nothing and return the estimator unchanged
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transforms the image samples in X into a matrix of patch data.

__init__ (*self*, *patch_size=None*, *max_patches=None*, *random_state=None*)

fit (*self*, *X*, *y=None*)
Do nothing and return the estimator unchanged

This method is just there to implement the usual API and hence work in pipelines.

Parameters

X [array-like, shape [n_samples, n_features]] Training data.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, ***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.

Returns

self

transform (*self*, *X*)
Transforms the image samples in X into a matrix of patch data.

Parameters

X [array, shape = (n_samples, image_height, image_width) or] (n_samples, image_height, image_width, n_channels) Array of images from which to extract patches. For color images, the last dimension specifies the channel: a RGB image would have `n_channels=3`.

Returns

patches [array, shape = (n_patches, patch_height, patch_width) or] (n_patches, patch_height, patch_width, n_channels) The collection of patches extracted from the images, where `n_patches` is either `n_samples * max_patches` or the total number of patches that can be extracted.

6.15.4 From text

The *sklearn.feature_extraction.text* submodule gathers utilities to build feature vectors from text documents.

feature_extraction.text.CountVectorizer([...])	Convert a collection of text documents to a matrix of token counts
feature_extraction.text.HashingVectorizer([...])	Convert a collection of text documents to a matrix of token occurrences
feature_extraction.text.TfidfTransformer([...])	Transform a count matrix to a normalized tf or tf-idf representation
feature_extraction.text.TfidfVectorizer([...])	Convert a collection of raw documents to a matrix of TF-IDF features.

sklearn.feature_extraction.text.CountVectorizer

class sklearn.feature_extraction.text.**CountVectorizer**(*input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *analyzer='word'*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.int64'>*)

Convert a collection of text documents to a matrix of token counts

This implementation produces a sparse representation of the counts using scipy.sparse.csr_matrix.

If you do not provide an a-priori dictionary and you do not use an analyzer that does some kind of feature selection then the number of features will be equal to the vocabulary size found by analyzing the data.

Read more in the *User Guide*.

Parameters

input [string {'filename', 'file', 'content'}] If 'filename', the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze.

If 'file', the sequence items must have a 'read' method (file-like object) that is called to fetch the bytes in memory.

Otherwise the input is expected to be the sequence strings or bytes items are expected to be analyzed directly.

encoding [string, 'utf-8' by default.] If bytes or files are given to analyze, this encoding is used to decode.

decode_error [{'strict', 'ignore', 'replace'}] Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given `encoding`. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

strip_accents [{'ascii', 'unicode', None}] Remove accents and perform other character normalization during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.

Both 'ascii' and 'unicode' use NFKD normalization from `unicodedata.normalize`.

lowercase [boolean, True by default] Convert all characters to lowercase before tokenizing.

preprocessor [callable or None (default)] Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps.

tokenizer [callable or None (default)] Override the string tokenization step while preserving the preprocessing and n-grams generation steps. Only applies if `analyzer == 'word'`.

stop_words [string {'english'}, list, or None (default)] If 'english', a built-in stop word list for English is used. There are several known issues with 'english' and you should consider an alternative (see *Using stop words*).

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if `analyzer == 'word'`.

If None, no stop words will be used. max_df can be set to a value in the range [0.7, 1.0) to automatically detect and filter stop words based on intra corpus document frequency of terms.

token_pattern [string] Regular expression denoting what constitutes a "token", only used if `analyzer == 'word'`. The default regexp select tokens of 2 or more alphanumeric characters (punctuation is completely ignored and always treated as a token separator).

ngram_range [tuple (min_n, max_n)] The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.

analyzer [string, {'word', 'char', 'char_wb'} or callable] Whether the feature should be made of word or character n-grams. Option 'char_wb' creates character n-grams only from text inside word boundaries; n-grams at the edges of words are padded with space.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

Changed in version 0.21.

Since v0.21, if `input` is `filename` or `file`, the data is first read from the file and then passed to the given callable analyzer.

max_df [float in range [0.0, 1.0] or int, default=1.0] When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop

words). If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

min_df [float in range [0.0, 1.0] or int, default=1] When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

max_features [int or None, default=None] If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus.

This parameter is ignored if vocabulary is not None.

vocabulary [Mapping or iterable, optional] Either a Mapping (e.g., a dict) where keys are terms and values are indices in the feature matrix, or an iterable over terms. If not given, a vocabulary is determined from the input documents. Indices in the mapping should not be repeated and should not have any gap between 0 and the largest index.

binary [boolean, default=False] If True, all non zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts.

dtype [type, optional] Type of the matrix returned by fit_transform() or transform().

Attributes

vocabulary_ [dict] A mapping of terms to feature indices.

stop_words_ [set] Terms that were ignored because they either:

- occurred in too many documents (`max_df`)
- occurred in too few documents (`min_df`)
- were cut off by feature selection (`max_features`).

This is only available if no vocabulary was given.

See also:

HashingVectorizer, *TfidfVectorizer*

Notes

The `stop_words_` attribute can get large and increase the model size when pickling. This attribute is provided only for introspection and can be safely removed using delattr or set to None before pickling.

Examples

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This document is the second document.',
...     'And this is the third one.',
...     'Is this the first document?',
... ]
>>> vectorizer = CountVectorizer()
>>> X = vectorizer.fit_transform(corpus)
>>> print(vectorizer.get_feature_names())
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

```
>>> print(X.toarray())
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
```

Methods

build_analyzer(self)	Return a callable that handles preprocessing and tokenization
build_preprocessor(self)	Return a function to preprocess the text before tokenization
build_tokenizer(self)	Return a function that splits a string into a sequence of tokens
decode(self, doc)	Decode the input into a string of unicode symbols
fit(self, raw_documents[, y])	Learn a vocabulary dictionary of all tokens in the raw documents.
fit_transform(self, raw_documents[, y])	Learn the vocabulary dictionary and return term-document matrix.
get_feature_names(self)	Array mapping from feature integer indices to feature name
get_params(self[, deep])	Get parameters for this estimator.
get_stop_words(self)	Build or fetch the effective stop words list
inverse_transform(self, X)	Return terms per document with nonzero entries in X.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, raw_documents)	Transform documents to document-term matrix.

__init__(*self*, *input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *analyzer='word'*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.int64'>*)

build_analyzer(*self*)
Return a callable that handles preprocessing and tokenization

build_preprocessor(*self*)
Return a function to preprocess the text before tokenization

build_tokenizer(*self*)
Return a function that splits a string into a sequence of tokens

decode(*self*, *doc*)
Decode the input into a string of unicode symbols

The decoding strategy depends on the vectorizer parameters.

Parameters

doc [string] The string to decode

fit(*self*, *raw_documents*, *y=None*)
Learn a vocabulary dictionary of all tokens in the raw documents.

Parameters

raw_documents [iterable] An iterable which yields either str, unicode or file objects.

Returns

self

fit_transform(*self*, *raw_documents*, *y=None*)
Learn the vocabulary dictionary and return term-document matrix.

This is equivalent to fit followed by transform, but more efficiently implemented.

Parameters

raw_documents [iterable] An iterable which yields either str, unicode or file objects.

Returns

X [array, [n_samples, n_features]] Document-term matrix.

get_feature_names(*self*)
Array mapping from feature integer indices to feature name

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_stop_words(*self*)
Build or fetch the effective stop words list

inverse_transform(*self*, *X*)
Return terms per document with nonzero entries in X.

Parameters

X [{array, sparse matrix}, shape = [n_samples, n_features]]

Returns

X_inv [list of arrays, len = n_samples] List of arrays of terms.

set_params(*self*, ***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.

Returns

self

transform(*self*, *raw_documents*)
Transform documents to document-term matrix.

Extract token counts out of raw text documents using the vocabulary fitted with fit or the one provided to the constructor.

Parameters

raw_documents [iterable] An iterable which yields either str, unicode or file objects.

Returns

X [sparse matrix, [n_samples, n_features]] Document-term matrix.

Examples using `sklearn.feature_extraction.text.CountVectorizer`

- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*
- *Sample pipeline for text feature extraction and evaluation*

`sklearn.feature_extraction.text.HashingVectorizer`

class `sklearn.feature_extraction.text.`**`HashingVectorizer`**(*input='content', encoding='utf-8', decode_error='strict', strip_accents=None, lowercase=True, preprocessor=None, tokenizer=None, stop_words=None, token_pattern='(?u)\b\w\w+\b', ngram_range=(1, 1), analyzer='word', n_features=1048576, binary=False, norm='l2', alternate_sign=True, dtype=<class 'numpy.float64'>*)

Convert a collection of text documents to a matrix of token occurrences

It turns a collection of text documents into a scipy.sparse matrix holding token occurrence counts (or binary occurrence information), possibly normalized as token frequencies if norm='l1' or projected on the euclidean unit sphere if norm='l2'.

This text vectorizer implementation uses the hashing trick to find the token string name to feature integer index mapping.

This strategy has several advantages:

- it is very low memory scalable to large datasets as there is no need to store a vocabulary dictionary in memory
- it is fast to pickle and un-pickle as it holds no state besides the constructor parameters
- it can be used in a streaming (partial fit) or parallel pipeline as there is no state computed during fit.

There are also a couple of cons (vs using a CountVectorizer with an in-memory vocabulary):

- there is no way to compute the inverse transform (from feature indices to string feature names) which can be a problem when trying to introspect which features are most important to a model.
- there can be collisions: distinct tokens can be mapped to the same feature index. However in practice this is rarely an issue if n_features is large enough (e.g. 2 ** 18 for text classification problems).
- no IDF weighting as this would render the transformer stateful.

The hash function employed is the signed 32-bit version of Murmurhash3.

Read more in the *User Guide*.

Parameters

input [string {'filename', 'file', 'content'}] If 'filename', the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze.

If 'file', the sequence items must have a 'read' method (file-like object) that is called to fetch the bytes in memory.

Otherwise the input is expected to be the sequence strings or bytes items are expected to be analyzed directly.

encoding [string, default='utf-8'] If bytes or files are given to analyze, this encoding is used to decode.

decode_error [{'strict', 'ignore', 'replace'}] Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given `encoding`. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

strip_accents [{'ascii', 'unicode', None}] Remove accents and perform other character normalization during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.

Both 'ascii' and 'unicode' use NFKD normalization from `unicodedata.normalize`.

lowercase [boolean, default=True] Convert all characters to lowercase before tokenizing.

preprocessor [callable or None (default)] Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps.

tokenizer [callable or None (default)] Override the string tokenization step while preserving the preprocessing and n-grams generation steps. Only applies if `analyzer == 'word'`.

stop_words [string {'english'}, list, or None (default)] If 'english', a built-in stop word list for English is used. There are several known issues with 'english' and you should consider an alternative (see *Using stop words*).

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if `analyzer == 'word'`.

token_pattern [string] Regular expression denoting what constitutes a "token", only used if `analyzer == 'word'`. The default regexp selects tokens of 2 or more alphanumeric characters (punctuation is completely ignored and always treated as a token separator).

ngram_range [tuple (min_n, max_n), default=(1, 1)] The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.

analyzer [string, {'word', 'char', 'char_wb'} or callable] Whether the feature should be made of word or character n-grams. Option 'char_wb' creates character n-grams only from text inside word boundaries; n-grams at the edges of words are padded with space.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

Changed in version 0.21.

Since v0.21, if `input` is `filename` or `file`, the data is first read from the file and then passed to the given callable analyzer.

n_features [integer, default=(2 ** 20)] The number of features (columns) in the output matrices. Small numbers of features are likely to cause hash collisions, but large numbers will cause larger coefficient dimensions in linear learners.

binary [boolean, default=False.] If True, all non zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts.

norm ['l1', 'l2' or None, optional] Norm used to normalize term vectors. None for no normalization.

alternate_sign [boolean, optional, default True] When True, an alternating sign is added to the features as to approximately conserve the inner product in the hashed space even for small n_features. This approach is similar to sparse random projection.

New in version 0.19.

dtype [type, optional] Type of the matrix returned by fit_transform() or transform().

See also:

CountVectorizer, *TfidfVectorizer*

Examples

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This document is the second document.',
...     'And this is the third one.',
...     'Is this the first document?',
... ]
>>> vectorizer = HashingVectorizer(n_features=2**4)
>>> X = vectorizer.fit_transform(corpus)
>>> print(X.shape)
(4, 16)
```

Methods

build_analyzer(self)	Return a callable that handles preprocessing and tokenization
build_preprocessor(self)	Return a function to preprocess the text before tokenization
build_tokenizer(self)	Return a function that splits a string into a sequence of tokens
decode(self, doc)	Decode the input into a string of unicode symbols
fit(self, X[, y])	Does nothing: this transformer is stateless.
fit_transform(self, X[, y])	Transform a sequence of documents to a document-term matrix.
get_params(self[, deep])	Get parameters for this estimator.
get_stop_words(self)	Build or fetch the effective stop words list
partial_fit(self, X[, y])	Does nothing: this transformer is stateless.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform a sequence of documents to a document-term matrix.

__init__(*self*, *input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *analyzer='word'*, *n_features=1048576*, *binary=False*, *norm='l2'*, *alternate_sign=True*, *dtype=<class 'numpy.float64'>*)

build_analyzer(*self*)
Return a callable that handles preprocessing and tokenization

build_preprocessor(*self*)
Return a function to preprocess the text before tokenization

build_tokenizer(*self*)
Return a function that splits a string into a sequence of tokens

decode(*self*, *doc*)
Decode the input into a string of unicode symbols

The decoding strategy depends on the vectorizer parameters.

Parameters

doc [string] The string to decode

fit(*self*, *X*, *y=None*)
Does nothing: this transformer is stateless.

Parameters

X [array-like, shape [n_samples, n_features]] Training data.

fit_transform(*self*, *X*, *y=None*)
Transform a sequence of documents to a document-term matrix.

Parameters

X [iterable over raw text documents, length = n_samples] Samples. Each sample must be a text document (either bytes or unicode strings, file name or file object depending on the constructor argument) which will be tokenized and hashed.

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

Returns

X [scipy.sparse matrix, shape = (n_samples, self.n_features)] Document-term matrix.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_stop_words(*self*)
Build or fetch the effective stop words list

partial_fit(*self*, *X*, *y=None*)
Does nothing: this transformer is stateless.

This method is just there to mark the fact that this transformer can work in a streaming setup.

Parameters

X [array-like, shape [n_samples, n_features]] Training data.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform a sequence of documents to a document-term matrix.

Parameters

X [iterable over raw text documents, length = n_samples] Samples. Each sample must be a text document (either bytes or unicode strings, file name or file object depending on the constructor argument) which will be tokenized and hashed.

Returns

X [scipy.sparse matrix, shape = (n_samples, self.n_features)] Document-term matrix.

Examples using `sklearn.feature_extraction.text.HashingVectorizer`

- *Out-of-core classification of text documents*
- *Clustering text documents using k-means*
- *Classification of text documents using sparse features*

`sklearn.feature_extraction.text.`TfidfTransformer

class `sklearn.feature_extraction.text.`**TfidfTransformer**(*norm='l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*)
Transform a count matrix to a normalized tf or tf-idf representation

Tf means term-frequency while tf-idf means term-frequency times inverse document-frequency. This is a common term weighting scheme in information retrieval, that has also found good use in document classification.

The goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus.

The formula that is used to compute the tf-idf for a term t of a document d in a document set is tf-idf(t, d) = tf(t, d) * idf(t), and the idf is computed as idf(t) = log [n / df(t)] + 1 (if `smooth_idf=False`), where n is the total number of documents in the document set and df(t) is the document frequency of t; the document frequency is the number of documents in the document set that contain the term t. The effect of adding "1" to the idf in the equation above is that terms with zero idf, i.e., terms that occur in all documents in a training set, will not be entirely ignored. (Note that the idf formula above differs from the standard textbook notation that defines the idf as idf(t) = log [n / (df(t) + 1)]).

If `smooth_idf=True` (the default), the constant "1" is added to the numerator and denominator of the idf as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions: idf(d, t) = log [(1 + n) / (1 + df(d, t))] + 1.

Furthermore, the formulas used to compute tf and idf depend on parameter settings that correspond to the SMART notation used in IR as follows:

Tf is "n" (natural) by default, "l" (logarithmic) when `sublinear_tf=True`. Idf is "t" when use_idf is given, "n" (none) otherwise. Normalization is "c" (cosine) when `norm='l2'`, "n" (none) when `norm=None`.

Read more in the *User Guide*.

Parameters

norm ['l1', 'l2' or None, optional (default='l2')] Each output row will have unit norm, either: * 'l2': Sum of squares of vector elements is 1. The cosine similarity between two vectors is their dot product when l2 norm has been applied. * 'l1': Sum of absolute values of vector elements is 1. See `preprocessing.normalize`

use_idf [boolean (default=True)] Enable inverse-document-frequency reweighting.

smooth_idf [boolean (default=True)] Smooth idf weights by adding one to document frequencies, as if an extra document was seen containing every term in the collection exactly once. Prevents zero divisions.

sublinear_tf [boolean (default=False)] Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).

Attributes

idf_ [array, shape (n_features)] The inverse document frequency (IDF) vector; only defined if `use_idf` is True.

References

[R1b90ac3ca370-Yates2011], *[R1b90ac3ca370-MRS2008]*

Methods

fit(self, X[, y])	Learn the idf vector (global term weights)
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, copy])	Transform a count matrix to a tf or tf-idf representation

__init__(*self*, *norm='l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*)

fit(*self*, *X*, *y=None*)
Learn the idf vector (global term weights)

Parameters

X [sparse matrix, [n_samples, n_features]] a matrix of term/token counts

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*, *copy=True*)
Transform a count matrix to a tf or tf-idf representation

Parameters

X [sparse matrix, [n_samples, n_features]] a matrix of term/token counts

copy [boolean, default True] Whether to copy X and operate on the copy or perform in-place operations.

Returns

vectors [sparse matrix, [n_samples, n_features]]

Examples using `sklearn.feature_extraction.text.TfidfTransformer`

- *Sample pipeline for text feature extraction and evaluation*
- *Clustering text documents using k-means*

sklearn.feature_extraction.text.TfidfVectorizer

class sklearn.feature_extraction.text.**TfidfVectorizer**(*input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *analyzer='word'*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.float64'>*, *norm='l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*)

Convert a collection of raw documents to a matrix of TF-IDF features.

Equivalent to `CountVectorizer` followed by `TfidfTransformer`.

Read more in the *User Guide*.

Parameters

input [string {'filename', 'file', 'content'}] If 'filename', the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze.

If 'file', the sequence items must have a 'read' method (file-like object) that is called to fetch the bytes in memory.

Otherwise the input is expected to be the sequence strings or bytes items are expected to be analyzed directly.

encoding [string, 'utf-8' by default.] If bytes or files are given to analyze, this encoding is used to decode.

decode_error [{'strict', 'ignore', 'replace'} (default='strict')] Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given `encoding`. By default, it is 'strict', meaning that a UnicodeDecodeError will be raised. Other values are 'ignore' and 'replace'.

strip_accents [{'ascii', 'unicode', None} (default=None)] Remove accents and perform other character normalization during the preprocessing step. 'ascii' is a fast method that only works on characters that have an direct ASCII mapping. 'unicode' is a slightly slower method that works on any characters. None (default) does nothing.

Both 'ascii' and 'unicode' use NFKD normalization from `unicodedata.normalize`.

lowercase [boolean (default=True)] Convert all characters to lowercase before tokenizing.

preprocessor [callable or None (default=None)] Override the preprocessing (string transformation) stage while preserving the tokenizing and n-grams generation steps.

tokenizer [callable or None (default=None)] Override the string tokenization step while preserving the preprocessing and n-grams generation steps. Only applies if `analyzer == 'word'`.

analyzer [string, {'word', 'char', 'char_wb'} or callable] Whether the feature should be made of word or character n-grams. Option 'char_wb' creates character n-grams only from text inside word boundaries; n-grams at the edges of words are padded with space.

If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

Changed in version 0.21.

Since v0.21, if `input` is `filename` or `file`, the data is first read from the file and then passed to the given callable analyzer.

stop_words [string {'english'}, list, or None (default=None)] If a string, it is passed to _check_stop_list and the appropriate stop list is returned. 'english' is currently the only supported string value. There are several known issues with 'english' and you should consider an alternative (see *Using stop words*).

If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if `analyzer == 'word'`.

If None, no stop words will be used. max_df can be set to a value in the range [0.7, 1.0) to automatically detect and filter stop words based on intra corpus document frequency of terms.

token_pattern [string] Regular expression denoting what constitutes a "token", only used if `analyzer == 'word'`. The default regexp selects tokens of 2 or more alphanumeric characters (punctuation is completely ignored and always treated as a token separator).

ngram_range [tuple (min_n, max_n) (default=(1, 1))] The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.

max_df [float in range [0.0, 1.0] or int (default=1.0)] When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words). If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

min_df [float in range [0.0, 1.0] or int (default=1)] When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

max_features [int or None (default=None)] If not None, build a vocabulary that only consider the top max_features ordered by term frequency across the corpus.

This parameter is ignored if vocabulary is not None.

vocabulary [Mapping or iterable, optional (default=None)] Either a Mapping (e.g., a dict) where keys are terms and values are indices in the feature matrix, or an iterable over terms. If not given, a vocabulary is determined from the input documents.

binary [boolean (default=False)] If True, all non-zero term counts are set to 1. This does not mean outputs will have only 0/1 values, only that the tf term in tf-idf is binary. (Set idf and normalization to False to get 0/1 outputs.)

dtype [type, optional (default=float64)] Type of the matrix returned by fit_transform() or transform().

norm ['l1', 'l2' or None, optional (default='l2')] Each output row will have unit norm, either: * 'l2': Sum of squares of vector elements is 1. The cosine similarity between two vectors is their dot product when l2 norm has been applied. * 'l1': Sum of absolute values of vector elements is 1. See `preprocessing.normalize`

use_idf [boolean (default=True)] Enable inverse-document-frequency reweighting.

smooth_idf [boolean (default=True)] Smooth idf weights by adding one to document frequencies, as if an extra document was seen containing every term in the collection exactly once. Prevents zero divisions.

sublinear_tf [boolean (default=False)] Apply sublinear tf scaling, i.e. replace tf with 1 + log(tf).

Attributes

vocabulary_ [dict] A mapping of terms to feature indices.

idf_ [array, shape (n_features)] The inverse document frequency (IDF) vector; only defined if `use_idf` is True.

stop_words_ [set] Terms that were ignored because they either:

- occurred in too many documents (`max_df`)
- occurred in too few documents (`min_df`)
- were cut off by feature selection (`max_features`).

This is only available if no vocabulary was given.

See also:

CountVectorizer Transforms text into a sparse matrix of n-gram counts.

TfidfTransformer Performs the TF-IDF transformation from a provided matrix of counts.

Notes

The `stop_words_` attribute can get large and increase the model size when pickling. This attribute is provided only for introspection and can be safely removed using delattr or set to None before pickling.

Examples

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This document is the second document.',
...     'And this is the third one.',
...     'Is this the first document?',
... ]
>>> vectorizer = TfidfVectorizer()
>>> X = vectorizer.fit_transform(corpus)
>>> print(vectorizer.get_feature_names())
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
>>> print(X.shape)
(4, 9)
```

Methods

build_analyzer(self)	Return a callable that handles preprocessing and tokenization

Continued on next page

Table 6.103 – continued from previous page

build_preprocessor(self)	Return a function to preprocess the text before tokenization
build_tokenizer(self)	Return a function that splits a string into a sequence of tokens
decode(self, doc)	Decode the input into a string of unicode symbols
fit(self, raw_documents[, y])	Learn vocabulary and idf from training set.
fit_transform(self, raw_documents[, y])	Learn vocabulary and idf, return term-document matrix.
get_feature_names(self)	Array mapping from feature integer indices to feature name
get_params(self[, deep])	Get parameters for this estimator.
get_stop_words(self)	Build or fetch the effective stop words list
inverse_transform(self, X)	Return terms per document with nonzero entries in X.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, raw_documents[, copy])	Transform documents to document-term matrix.

__init__(*self*, *input='content'*, *encoding='utf-8'*, *decode_error='strict'*, *strip_accents=None*, *lowercase=True*, *preprocessor=None*, *tokenizer=None*, *analyzer='word'*, *stop_words=None*, *token_pattern='(?u)\b\w\w+\b'*, *ngram_range=(1, 1)*, *max_df=1.0*, *min_df=1*, *max_features=None*, *vocabulary=None*, *binary=False*, *dtype=<class 'numpy.float64'>*, *norm='l2'*, *use_idf=True*, *smooth_idf=True*, *sublinear_tf=False*)

build_analyzer(*self*)
Return a callable that handles preprocessing and tokenization

build_preprocessor(*self*)
Return a function to preprocess the text before tokenization

build_tokenizer(*self*)
Return a function that splits a string into a sequence of tokens

decode(*self*, *doc*)
Decode the input into a string of unicode symbols

The decoding strategy depends on the vectorizer parameters.

Parameters

doc [string] The string to decode

fit(*self*, *raw_documents*, *y=None*)
Learn vocabulary and idf from training set.

Parameters

raw_documents [iterable] an iterable which yields either str, unicode or file objects

Returns

self [TfidfVectorizer]

fit_transform(*self*, *raw_documents*, *y=None*)
Learn vocabulary and idf, return term-document matrix.

This is equivalent to fit followed by transform, but more efficiently implemented.

Parameters

raw_documents [iterable] an iterable which yields either str, unicode or file objects

Returns

X [sparse matrix, [n_samples, n_features]] Tf-idf-weighted document-term matrix.

get_feature_names(*self*)
Array mapping from feature integer indices to feature name

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_stop_words(*self*)
Build or fetch the effective stop words list

inverse_transform(*self*, *X*)
Return terms per document with nonzero entries in X.

Parameters

X [{array, sparse matrix}, shape = [n_samples, n_features]]

Returns

X_inv [list of arrays, len = n_samples] List of arrays of terms.

set_params(*self*, ***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.

Returns

self

transform(*self*, *raw_documents*, *copy=True*)
Transform documents to document-term matrix.

Uses the vocabulary and document frequencies (df) learned by fit (or fit_transform).

Parameters

raw_documents [iterable] an iterable which yields either str, unicode or file objects

copy [boolean, default True] Whether to copy X and operate on the copy or perform in-place operations.

Returns

X [sparse matrix, [n_samples, n_features]] Tf-idf-weighted document-term matrix.

Examples using `sklearn.feature_extraction.text.TfidfVectorizer`

- *Topic extraction with Non-negative Matrix Factorization and Latent Dirichlet Allocation*
- *Biclustering documents with the Spectral Co-clustering algorithm*
- *Column Transformer with Heterogeneous Data Sources*
- *Clustering text documents using k-means*

- *Classification of text documents using sparse features*

6.16 sklearn.feature_selection: Feature Selection

The *sklearn.feature_selection* module implements feature selection algorithms. It currently includes univariate filter selection methods and the recursive feature elimination algorithm.

User guide: See the *Feature selection* section for further details.

feature_selection.GenericUnivariateSelect([...])	Univariate feature selector with configurable strategy.
feature_selection.SelectPercentile([...])	Select features according to a percentile of the highest scores.
feature_selection.SelectKBest([score_func, k])	Select features according to the k highest scores.
feature_selection.SelectFpr([score_func, alpha])	Filter: Select the pvalues below alpha based on a FPR test.
feature_selection.SelectFdr([score_func, alpha])	Filter: Select the p-values for an estimated false discovery rate
feature_selection.SelectFromModel(estimator)	Meta-transformer for selecting features based on importance weights.
feature_selection.SelectFwe([score_func, alpha])	Filter: Select the p-values corresponding to Family-wise error rate
feature_selection.RFE(estimator[, ...])	Feature ranking with recursive feature elimination.
feature_selection.RFECV(estimator[, step, ...])	Feature ranking with recursive feature elimination and cross-validated selection of the best number of features.
feature_selection.VarianceThreshold([threshold])	Feature selector that removes all low-variance features.

6.16.1 sklearn.feature_selection.GenericUnivariateSelect

class sklearn.feature_selection.**GenericUnivariateSelect**(*score_func=<function f_classif>*, *mode='percentile'*, *param=1e-05*)

Univariate feature selector with configurable strategy.

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues). For modes 'percentile' or 'kbest' it can return a single array scores.

mode [{'percentile', 'k_best', 'fpr', 'fdr', 'fwe'}] Feature selection mode.

param [float or int depending on the feature selection mode] Parameter of the corresponding mode.

Attributes

scores_ [array-like, shape=(n_features,)] Scores of features.

pvalues_ [array-like, shape=(n_features,)] p-values of feature scores, None if score_func returned scores only.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

mutual_info_classif Mutual information for a discrete target.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

mutual_info_regression Mutual information for a continuous target.

SelectPercentile Select features based on percentile of the highest scores.

SelectKBest Select features based on the k highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFdr Select features based on an estimated false discovery rate.

SelectFwe Select features based on family-wise error rate.

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.feature_selection import GenericUnivariateSelect, chi2
>>> X, y = load_breast_cancer(return_X_y=True)
>>> X.shape
(569, 30)
>>> transformer = GenericUnivariateSelect(chi2, 'k_best', param=20)
>>> X_new = transformer.fit_transform(X, y)
>>> X_new.shape
(569, 20)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *score_func=<function f_classif at 0x7efe30bb2158>*, *mode='percentile'*, *param=1e-05*)

fit (*self*, *X*, *y*)

Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like, shape = [n_samples, n_features]] The training input samples.

y [array-like, shape = [n_samples]] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

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

Returns

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

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

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

Returns

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

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

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

Returns

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

6.16.2 sklearn.feature_selection.SelectPercentile

class sklearn.feature_selection.**SelectPercentile**(*score_func=<function f_classif>*, *percentile=10*)
Select features according to a percentile of the highest scores.

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues) or a single array with scores. Default is f_classif (see below "See also"). The default function only works with classification tasks.

percentile [int, optional, default=10] Percent of features to keep.

Attributes

scores_ [array-like, shape=(n_features,)] Scores of features.

pvalues_ [array-like, shape=(n_features,)] p-values of feature scores, None if `score_func` returned only scores.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

mutual_info_classif Mutual information for a discrete target.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

mutual_info_regression Mutual information for a continuous target.

SelectKBest Select features based on the k highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFdr Select features based on an estimated false discovery rate.

SelectFwe Select features based on family-wise error rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

Notes

Ties between features with equal scores will be broken in an unspecified way.

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.feature_selection import SelectPercentile, chi2
>>> X, y = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> X_new = SelectPercentile(chi2, percentile=10).fit_transform(X, y)
>>> X_new.shape
(1797, 7)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *score_func=<function f_classif at 0x7f3c10e93840>*, *percentile=10*)

fit (*self*, *X*, *y*)

Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like, shape = [n_samples, n_features]] The training input samples.

y [array-like, shape = [n_samples]] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

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

Returns

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

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

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

Returns

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

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

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

Returns

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

Examples using `sklearn.feature_selection.SelectPercentile`

- *Feature agglomeration vs. univariate selection*
- *Univariate Feature Selection*
- *SVM-Anova: SVM with univariate feature selection*

6.16.3 sklearn.feature_selection.SelectKBest

class sklearn.feature_selection.**SelectKBest**(*score_func=<function f_classif>*, *k=10*)

Select features according to the k highest scores.

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues) or a single array with scores. Default is f_classif (see below "See also"). The default function only works with classification tasks.

k [int or "all", optional, default=10] Number of top features to select. The "all" option bypasses selection, for use in a parameter search.

Attributes

scores_ [array-like, shape=(n_features,)] Scores of features.

pvalues_ [array-like, shape=(n_features,)] p-values of feature scores, None if score_func returned only scores.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

mutual_info_classif Mutual information for a discrete target.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

mutual_info_regression Mutual information for a continuous target.

SelectPercentile Select features based on percentile of the highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFdr Select features based on an estimated false discovery rate.

SelectFwe Select features based on family-wise error rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

Notes

Ties between features with equal scores will be broken in an unspecified way.

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.feature_selection import SelectKBest, chi2
>>> X, y = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> X_new = SelectKBest(chi2, k=20).fit_transform(X, y)
>>> X_new.shape
(1797, 20)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *score_func=<function f_classif at 0x7f3c10e93840>*, *k=10*)

fit (*self*, *X*, *y*)
Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like, shape = [n_samples, n_features]] The training input samples.

y [array-like, shape = [n_samples]] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_support (*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

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

Returns

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

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

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

Returns

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

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

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

Returns

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

Examples using `sklearn.feature_selection.SelectKBest`

- *Concatenating multiple feature extraction methods*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *Pipeline Anova SVM*
- *Classification of text documents using sparse features*

6.16.4 `sklearn.feature_selection`.SelectFpr

class `sklearn.feature_selection.`**SelectFpr**(*score_func=<function f_classif>*, *alpha=0.05*)
Filter: Select the pvalues below alpha based on a FPR test.

FPR test stands for False Positive Rate test. It controls the total amount of false detections.

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues). Default is f_classif (see below "See also"). The default function only works with classification tasks.

alpha [float, optional] The highest p-value for features to be kept.

Attributes

scores_ [array-like, shape=(n_features,)] Scores of features.

pvalues_ [array-like, shape=(n_features,)] p-values of feature scores.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

chi2 Chi-squared stats of non-negative features for classification tasks.

mutual_info_classif

f_regression F-value between label/feature for regression tasks.

mutual_info_regression Mutual information between features and the target.

SelectPercentile Select features based on percentile of the highest scores.

SelectKBest Select features based on the k highest scores.

SelectFdr Select features based on an estimated false discovery rate.

SelectFwe Select features based on family-wise error rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.feature_selection import SelectFpr, chi2
>>> X, y = load_breast_cancer(return_X_y=True)
>>> X.shape
(569, 30)
>>> X_new = SelectFpr(chi2, alpha=0.01).fit_transform(X, y)
>>> X_new.shape
(569, 16)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__(*self*, *score_func=<function f_classif at 0x7efe30bb2158>*, *alpha=0.05*)

fit(*self*, *X*, *y*)
Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like, shape = [n_samples, n_features]] The training input samples.

y [array-like, shape = [n_samples]] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

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

Returns

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

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

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

Returns

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

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

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

Returns

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

6.16.5 sklearn.feature_selection.SelectFdr

class sklearn.feature_selection.**SelectFdr**(*score_func=<function f_classif>*, *alpha=0.05*)
Filter: Select the p-values for an estimated false discovery rate

This uses the Benjamini-Hochberg procedure. `alpha` is an upper bound on the expected false discovery rate.

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues). Default is f_classif (see below "See also"). The default function only works with classification tasks.

alpha [float, optional] The highest uncorrected p-value for features to keep.

Attributes

scores_ [array-like, shape=(n_features,)] Scores of features.

pvalues_ [array-like, shape=(n_features,)] p-values of feature scores.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

mutual_info_classif Mutual information for a discrete target.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

mutual_info_regression Mutual information for a contnuous target.

SelectPercentile Select features based on percentile of the highest scores.

SelectKBest Select features based on the k highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFwe Select features based on family-wise error rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

References

https://en.wikipedia.org/wiki/False_discovery_rate

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.feature_selection import SelectFdr, chi2
>>> X, y = load_breast_cancer(return_X_y=True)
>>> X.shape
(569, 30)
>>> X_new = SelectFdr(chi2, alpha=0.01).fit_transform(X, y)
>>> X_new.shape
(569, 16)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *score_func=<function f_classif at 0x7efe30bb2158>*, *alpha=0.05*)

fit (*self*, *X*, *y*)

Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like, shape = [n_samples, n_features]] The training input samples.

y [array-like, shape = [n_samples]] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

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

Returns

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

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

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

Returns

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

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

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

Returns

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

6.16.6 sklearn.feature_selection.SelectFromModel

class sklearn.feature_selection.**SelectFromModel**(*estimator*, *threshold=None*, *prefit=False*, *norm_order=1*, *max_features=None*)

Meta-transformer for selecting features based on importance weights.

New in version 0.17.

Parameters

estimator [object] The base estimator from which the transformer is built. This can be both a fitted (if `prefit` is set to True) or a non-fitted estimator. The estimator must have either a `feature_importances_` or `coef_` attribute after fitting.

threshold [string, float, optional default None] The threshold value to use for feature selection. Features whose importance is greater or equal are kept while the others are discarded. If "median" (resp. "mean"), then the `threshold` value is the median (resp. the mean) of the feature importances. A scaling factor (e.g., "1.25*mean") may also be used. If None and if the estimator has a parameter penalty set to l1, either explicitly or implicitly (e.g, Lasso), the threshold used is 1e-5. Otherwise, "mean" is used by default.

prefit [bool, default False] Whether a prefit model is expected to be passed into the constructor directly or not. If True, `transform` must be called directly and SelectFromModel cannot be used with `cross_val_score`, `GridSearchCV` and similar utilities that clone the estimator. Otherwise train the model using `fit` and then `transform` to do feature selection.

norm_order [non-zero int, inf, -inf, default 1] Order of the norm used to filter the vectors of coefficients below `threshold` in the case where the `coef_` attribute of the estimator is of dimension 2.

max_features [int or None, optional] The maximum number of features selected scoring above `threshold`. To disable `threshold` and only select based on `max_features`, set `threshold=-np.inf`.

New in version 0.20.

Attributes

estimator_ [an estimator] The base estimator from which the transformer is built. This is stored only when a non-fitted estimator is passed to the `SelectFromModel`, i.e when prefit is False.

threshold_ [float] The threshold value used for feature selection.

Methods

fit(self, X[, y])	Fit the SelectFromModel meta-transformer.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
partial_fit(self, X[, y])	Fit the SelectFromModel meta-transformer only once.

Continued on next page

Table 6.110 – continued from previous page

set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *estimator*, *threshold=None*, *prefit=False*, *norm_order=1*, *max_features=None*)

fit (*self*, *X*, *y=None*, ***fit_params*)
Fit the SelectFromModel meta-transformer.

Parameters

X [array-like of shape (n_samples, n_features)] The training input samples.

y [array-like, shape (n_samples,)] The target values (integers that correspond to classes in classification, real numbers in regression).

****fit_params** [Other estimator specific parameters]

Returns

self [object]

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_support (*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

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

Returns

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

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

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

Returns

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

partial_fit(*self*, *X*, *y=None*, ***fit_params*)
Fit the SelectFromModel meta-transformer only once.

Parameters

X [array-like of shape (n_samples, n_features)] The training input samples.

y [array-like, shape (n_samples,)] The target values (integers that correspond to classes in classification, real numbers in regression).

****fit_params** [Other estimator specific parameters]

Returns

self [object]

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

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

Returns

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

Examples using sklearn.feature_selection.SelectFromModel

- *Feature selection using SelectFromModel and LassoCV*
- *Classification of text documents using sparse features*

6.16.7 sklearn.feature_selection.SelectFwe

class sklearn.feature_selection.**SelectFwe**(*score_func=<function f_classif>*, *alpha=0.05*)
Filter: Select the p-values corresponding to Family-wise error rate

Read more in the *User Guide*.

Parameters

score_func [callable] Function taking two arrays X and y, and returning a pair of arrays (scores, pvalues). Default is f_classif (see below "See also"). The default function only works with classification tasks.

alpha [float, optional] The highest uncorrected p-value for features to keep.

Attributes

scores_ [array-like, shape=(n_features,)] Scores of features.

pvalues_ [array-like, shape=(n_features,)] p-values of feature scores.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

SelectPercentile Select features based on percentile of the highest scores.

SelectKBest Select features based on the k highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFdr Select features based on an estimated false discovery rate.

GenericUnivariateSelect Univariate feature selector with configurable mode.

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.feature_selection import SelectFwe, chi2
>>> X, y = load_breast_cancer(return_X_y=True)
>>> X.shape
(569, 30)
>>> X_new = SelectFwe(chi2, alpha=0.01).fit_transform(X, y)
>>> X_new.shape
(569, 15)
```

Methods

fit(self, X, y)	Run score function on (X, y) and get the appropriate features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *score_func=<function f_classif at 0x7efe30bb2158>*, *alpha=0.05*)

fit (*self*, *X*, *y*)

Run score function on (X, y) and get the appropriate features.

Parameters

X [array-like, shape = [n_samples, n_features]] The training input samples.

y [array-like, shape = [n_samples]] The target values (class labels in classification, real numbers in regression).

Returns

self [object]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

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

Returns

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

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

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

Returns

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

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

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

Returns

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

6.16.8 sklearn.feature_selection.RFE

class sklearn.feature_selection.**RFE**(*estimator*, *n_features_to_select=None*, *step=1*, *verbose=0*)
Feature ranking with recursive feature elimination.

Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), the goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and the importance of each feature is obtained either through a `coef_` attribute or through a `feature_importances_` attribute. Then, the least important features are pruned from current set of features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

Read more in the *User Guide*.

Parameters

estimator [object] A supervised learning estimator with a `fit` method that provides information about feature importance either through a `coef_` attribute or through a `feature_importances_` attribute.

n_features_to_select [int or None (default=None)] The number of features to select. If `None`, half of the features are selected.

step [int or float, optional (default=1)] If greater than or equal to 1, then `step` corresponds to the (integer) number of features to remove at each iteration. If within (0.0, 1.0), then `step` corresponds to the percentage (rounded down) of features to remove at each iteration.

verbose [int, (default=0)] Controls verbosity of output.

Attributes

n_features_ [int] The number of selected features.

support_ [array of shape [n_features]] The mask of selected features.

ranking_ [array of shape [n_features]] The feature ranking, such that `ranking_[i]` corresponds to the ranking position of the i-th feature. Selected (i.e., estimated best) features are assigned rank 1.

estimator_ [object] The external estimator fit on the reduced dataset.

See also:

RFECV Recursive feature elimination with built-in cross-validated selection of the best number of features

References

[Re310f679c81e-1]

Examples

The following example shows how to retrieve the 5 right informative features in the Friedman #1 dataset.

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.feature_selection import RFE
>>> from sklearn.svm import SVR
>>> X, y = make_friedman1(n_samples=50, n_features=10, random_state=0)
>>> estimator = SVR(kernel="linear")
>>> selector = RFE(estimator, 5, step=1)
>>> selector = selector.fit(X, y)
>>> selector.support_
array([ True,  True,  True,  True,  True, False, False, False, False,
       False])
>>> selector.ranking_
array([1, 1, 1, 1, 1, 6, 4, 3, 2, 5])
```

Methods

decision_function(self, X)	Compute the decision function of X.
fit(self, X, y)	Fit the RFE model and then the underlying estimator on the selected features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
predict(self, X)	Reduce X to the selected features and then predict using the underlying estimator.
predict_log_proba(self, X)	Predict class log-probabilities for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y)	Reduce X to the selected features and then return the score of the underlying estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__ (*self*, *estimator*, *n_features_to_select=None*, *step=1*, *verbose=0*)

decision_function (*self*, *X*)

Compute the decision function of X.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to

a sparse `csr_matrix`.

Returns

score [array, shape = [n_samples, n_classes] or [n_samples]] The decision function of the input samples. The order of the classes corresponds to that in the attribute *classes_*. Regression and binary classification produce an array of shape [n_samples].

fit(*self*, *X*, *y*)

Fit the RFE model and then the underlying estimator on the selected features.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] The training input samples.

y [array-like, shape = [n_samples]] The target values.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

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

Returns

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

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

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

Returns

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

predict(*self*, *X*)

Reduce X to the selected features and then predict using the underlying estimator.

Parameters

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

Returns

y [array of shape [n_samples]] The predicted target values.

predict_log_proba(*self*, *X*)

Predict class log-probabilities for X.

Parameters

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

Returns

p [array of shape = [n_samples, n_classes]] The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba(*self*, *X*)

Predict class probabilities for X.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

p [array of shape = [n_samples, n_classes]] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*)

Reduce X to the selected features and then return the score of the underlying estimator.

Parameters

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

y [array of shape [n_samples]] The target values.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

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

Returns

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

Examples using `sklearn.feature_selection.RFE`

- *Recursive feature elimination*

6.16.9 `sklearn.feature_selection`.RFECV

class `sklearn.feature_selection.`**RFECV**(*estimator*, *step=1*, *min_features_to_select=1*, *cv='warn'*, *scoring=None*, *verbose=0*, *n_jobs=None*)
Feature ranking with recursive feature elimination and cross-validated selection of the best number of features.

See glossary entry for *cross-validation estimator*.

Read more in the *User Guide*.

Parameters

estimator [object] A supervised learning estimator with a `fit` method that provides information about feature importance either through a `coef_` attribute or through a `feature_importances_` attribute.

step [int or float, optional (default=1)] If greater than or equal to 1, then `step` corresponds to the (integer) number of features to remove at each iteration. If within (0.0, 1.0), then `step` corresponds to the percentage (rounded down) of features to remove at each iteration. Note that the last iteration may remove fewer than `step` features in order to reach `min_features_to_select`.

min_features_to_select [int, (default=1)] The minimum number of features to be selected. This number of features will always be scored, even if the difference between the original feature count and `min_features_to_select` isn't divisible by `step`.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if `y` is binary or multiclass, *`sklearn.model_selection.StratifiedKFold`* is used. If the estimator is a classifier or if `y` is neither binary nor multiclass, *`sklearn.model_selection.KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value of None will change from 3-fold to 5-fold in v0.22.

scoring [string, callable or None, optional, (default=None)] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

verbose [int, (default=0)] Controls verbosity of output.

n_jobs [int or None, optional (default=None)] Number of cores to run in parallel while fitting across folds. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

n_features_ [int] The number of selected features with cross-validation.

support_ [array of shape [n_features]] The mask of selected features.

ranking_ [array of shape [n_features]] The feature ranking, such that `ranking_[i]` corresponds to the ranking position of the i-th feature. Selected (i.e., estimated best) features are assigned rank 1.

grid_scores_ [array of shape [n_subsets_of_features]] The cross-validation scores such that `grid_scores_[i]` corresponds to the CV score of the i-th subset of features.

estimator_ [object] The external estimator fit on the reduced dataset.

See also:

RFE Recursive feature elimination

Notes

The size of `grid_scores_` is equal to `ceil((n_features - min_features_to_select) / step) + 1`, where step is the number of features removed at each iteration.

References

[R6f4d61ceb411-1]

Examples

The following example shows how to retrieve the a-priori not known 5 informative features in the Friedman #1 dataset.

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.feature_selection import RFECV
>>> from sklearn.svm import SVR
>>> X, y = make_friedman1(n_samples=50, n_features=10, random_state=0)
>>> estimator = SVR(kernel="linear")
>>> selector = RFECV(estimator, step=1, cv=5)
>>> selector = selector.fit(X, y)
>>> selector.support_
array([ True,  True,  True,  True,  True, False, False, False, False,
       False])
>>> selector.ranking_
array([1, 1, 1, 1, 1, 6, 4, 3, 2, 5])
```

Methods

decision_function(self, X)	Compute the decision function of X.
fit(self, X, y[, groups])	Fit the RFE model and automatically tune the number of selected features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
get_support(self[, indices])	Get a mask, or integer index, of the features selected
inverse_transform(self, X)	Reverse the transformation operation
predict(self, X)	Reduce X to the selected features and then predict using the underlying estimator.
predict_log_proba(self, X)	Predict class log-probabilities for X.
predict_proba(self, X)	Predict class probabilities for X.
score(self, X, y)	Reduce X to the selected features and then return the score of the underlying estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Reduce X to the selected features.

__init__(*self*, *estimator*, *step=1*, *min_features_to_select=1*, *cv='warn'*, *scoring=None*, *verbose=0*, *n_jobs=None*)

decision_function(*self*, *X*)

Compute the decision function of X.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

score [array, shape = [n_samples, n_classes] or [n_samples]] The decision function of the input samples. The order of the classes corresponds to that in the attribute *classes_*. Regression and binary classification produce an array of shape [n_samples].

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

Fit the RFE model and automatically tune the number of selected features.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vector, where *n_samples* is the number of samples and *n_features* is the total number of features.

y [array-like, shape = [n_samples]] Target values (integers for classification, real numbers for regression).

groups [array-like, shape = [n_samples], optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

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

Returns

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

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

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

Returns

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

predict(*self*, *X*)

Reduce X to the selected features and then predict using the underlying estimator.

Parameters

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

Returns

y [array of shape [n_samples]] The predicted target values.

predict_log_proba(*self*, *X*)
Predict class log-probabilities for X.

Parameters

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

Returns

p [array of shape = [n_samples, n_classes]] The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba(*self*, *X*)
Predict class probabilities for X.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

p [array of shape = [n_samples, n_classes]] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*)

Reduce X to the selected features and then return the score of the underlying estimator.

Parameters

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

y [array of shape [n_samples]] The target values.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

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

Returns

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

Examples using `sklearn.feature_selection.RFECV`

- *Recursive feature elimination with cross-validation*

6.16.10 `sklearn.feature_selection`.VarianceThreshold

class `sklearn.feature_selection.`**VarianceThreshold**(*threshold=0.0*)
Feature selector that removes all low-variance features.

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

Read more in the *User Guide*.

Parameters

threshold [float, optional] Features with a training-set variance lower than this threshold will be removed. The default is to keep all features with non-zero variance, i.e. remove the features that have the same value in all samples.

Attributes

variances_ [array, shape (n_features,)] Variances of individual features.

Examples

The following dataset has integer features, two of which are the same in every sample. These are removed with the default setting for threshold:

```
>>> X = [[0, 2, 0, 3], [0, 1, 4, 3], [0, 1, 1, 3]]
>>> selector = VarianceThreshold()
>>> selector.fit_transform(X)
array([[2, 0],
       [1, 4],
       [1, 1]])
```

Methods

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

__init__ (*self*, *threshold=0.0*)

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

Learn empirical variances from X.

Parameters

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

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

Returns

self

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_support(*self*, *indices=False*)
Get a mask, or integer index, of the features selected

Parameters

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

Returns

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

inverse_transform(*self*, *X*)
Reverse the transformation operation

Parameters

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

Returns

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

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Reduce X to the selected features.

Parameters

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

Returns

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

feature_selection.chi2(X, y)	Compute chi-squared stats between each non-negative feature and class.
feature_selection.f_classif(X, y)	Compute the ANOVA F-value for the provided sample.
feature_selection.f_regression(X, y[, center])	Univariate linear regression tests.
feature_selection.mutual_info_classif(X, y)	Estimate mutual information for a discrete target variable.
feature_selection.mutual_info_regression(X, y)	Estimate mutual information for a continuous target variable.

6.16.11 sklearn.feature_selection.chi2

sklearn.feature_selection.**chi2**(*X*, *y*)

Compute chi-squared stats between each non-negative feature and class.

This score can be used to select the n_features features with the highest values for the test chi-squared statistic from X, which must contain only non-negative features such as booleans or frequencies (e.g., term counts in document classification), relative to the classes.

Recall that the chi-square test measures dependence between stochastic variables, so using this function "weeds out" the features that are the most likely to be independent of class and therefore irrelevant for classification.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape = (n_samples, n_features_in)] Sample vectors.

y [array-like, shape = (n_samples,)] Target vector (class labels).

Returns

chi2 [array, shape = (n_features,)] chi2 statistics of each feature.

pval [array, shape = (n_features,)] p-values of each feature.

See also:

f_classif ANOVA F-value between label/feature for classification tasks.

f_regression F-value between label/feature for regression tasks.

Notes

Complexity of this algorithm is O(n_classes * n_features).

Examples using sklearn.feature_selection.chi2

- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *SVM-Anova: SVM with univariate feature selection*
- *Classification of text documents using sparse features*

6.16.12 sklearn.feature_selection.f_classif

sklearn.feature_selection.**f_classif**(*X*, *y*)
Compute the ANOVA F-value for the provided sample.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix} shape = [n_samples, n_features]] The set of regressors that will be tested sequentially.

y [array of shape(n_samples)] The data matrix.

Returns

F [array, shape = [n_features,]] The set of F values.

pval [array, shape = [n_features,]] The set of p-values.

See also:

chi2 Chi-squared stats of non-negative features for classification tasks.

f_regression F-value between label/feature for regression tasks.

Examples using sklearn.feature_selection.f_classif

- *Univariate Feature Selection*

6.16.13 sklearn.feature_selection.f_regression

sklearn.feature_selection.**f_regression**(*X*, *y*, *center=True*)
Univariate linear regression tests.

Linear model for testing the individual effect of each of many regressors. This is a scoring function to be used in a feature selection procedure, not a free standing feature selection procedure.

This is done in 2 steps:

1. The correlation between each regressor and the target is computed, that is, ((X[:, i] - mean(X[:, i])) * (y - mean_y)) / (std(X[:, i]) * std(y)).
2. It is converted to an F score then to a p-value.

For more on usage see the *User Guide*.

Parameters

X [{array-like, sparse matrix} shape = (n_samples, n_features)] The set of regressors that will be tested sequentially.

y [array of shape(n_samples).] The data matrix

center [True, bool,] If true, X and y will be centered.

Returns

F [array, shape=(n_features,)] F values of features.

pval [array, shape=(n_features,)] p-values of F-scores.

See also:

mutual_info_regression Mutual information for a continuous target.

f_classif ANOVA F-value between label/feature for classification tasks.

chi2 Chi-squared stats of non-negative features for classification tasks.

SelectKBest Select features based on the k highest scores.

SelectFpr Select features based on a false positive rate test.

SelectFdr Select features based on an estimated false discovery rate.

SelectFwe Select features based on family-wise error rate.

SelectPercentile Select features based on percentile of the highest scores.

Examples using `sklearn.feature_selection.f_regression`

- *Feature agglomeration vs. univariate selection*
- *Comparison of F-test and mutual information*
- *Pipeline Anova SVM*

6.16.14 `sklearn.feature_selection`.mutual_info_classif

`sklearn.feature_selection.`**`mutual_info_classif`**(*X*, *y*, *discrete_features='auto'*, *n_neighbors=3*, *copy=True*, *random_state=None*)

Estimate mutual information for a discrete target variable.

Mutual information (MI) *[1]* between two random variables is a non-negative value, which measures the dependency between the variables. It is equal to zero if and only if two random variables are independent, and higher values mean higher dependency.

The function relies on nonparametric methods based on entropy estimation from k-nearest neighbors distances as described in *[2]* and *[3]*. Both methods are based on the idea originally proposed in *[4]*.

It can be used for univariate features selection, read more in the *User Guide*.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Feature matrix.

y [array_like, shape (n_samples,)] Target vector.

discrete_features [{'auto', bool, array_like}, default 'auto'] If bool, then determines whether to consider all features discrete or continuous. If array, then it should be either a boolean mask with shape (n_features,) or array with indices of discrete features. If 'auto', it is assigned to False for dense *X* and to True for sparse *X*.

n_neighbors [int, default 3] Number of neighbors to use for MI estimation for continuous variables, see *[2]* and *[3]*. Higher values reduce variance of the estimation, but could introduce a bias.

copy [bool, default True] Whether to make a copy of the given data. If set to False, the initial data will be overwritten.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator for adding small noise to continuous variables in order to remove repeated values. 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`.

Returns

mi [ndarray, shape (n_features,)] Estimated mutual information between each feature and the target.

Notes

1. The term "discrete features" is used instead of naming them "categorical", because it describes the essence more accurately. For example, pixel intensities of an image are discrete features (but hardly categorical) and you will get better results if mark them as such. Also note, that treating a continuous variable as discrete and vice versa will usually give incorrect results, so be attentive about that.
2. True mutual information can't be negative. If its estimate turns out to be negative, it is replaced by zero.

References

[1], *[2]*, *[3]*, *[4]*

6.16.15 sklearn.feature_selection.mutual_info_regression

sklearn.feature_selection.**mutual_info_regression**(*X*, *y*, *discrete_features='auto'*, *n_neighbors=3*, *copy=True*, *random_state=None*)

Estimate mutual information for a continuous target variable.

Mutual information (MI) *[1]* between two random variables is a non-negative value, which measures the dependency between the variables. It is equal to zero if and only if two random variables are independent, and higher values mean higher dependency.

The function relies on nonparametric methods based on entropy estimation from k-nearest neighbors distances as described in *[2]* and *[3]*. Both methods are based on the idea originally proposed in *[4]*.

It can be used for univariate features selection, read more in the *User Guide*.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Feature matrix.

y [array_like, shape (n_samples,)] Target vector.

discrete_features [{'auto', bool, array_like}, default 'auto'] If bool, then determines whether to consider all features discrete or continuous. If array, then it should be either a boolean mask with shape (n_features,) or array with indices of discrete features. If 'auto', it is assigned to False for dense *X* and to True for sparse *X*.

n_neighbors [int, default 3] Number of neighbors to use for MI estimation for continuous variables, see *[2]* and *[3]*. Higher values reduce variance of the estimation, but could introduce a bias.

copy [bool, default True] Whether to make a copy of the given data. If set to False, the initial data will be overwritten.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator for adding small noise to continuous variables in order to remove repeated values. 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`.

Returns

mi [ndarray, shape (n_features,)] Estimated mutual information between each feature and the target.

Notes

1. The term "discrete features" is used instead of naming them "categorical", because it describes the essence more accurately. For example, pixel intensities of an image are discrete features (but hardly categorical) and you will get better results if mark them as such. Also note, that treating a continuous variable as discrete and vice versa will usually give incorrect results, so be attentive about that.
2. True mutual information can't be negative. If its estimate turns out to be negative, it is replaced by zero.

References

[1], *[2]*, *[3]*, *[4]*

Examples using `sklearn.feature_selection.mutual_info_regression`

- *Comparison of F-test and mutual information*

6.17 `sklearn.gaussian_process`: Gaussian Processes

The *`sklearn.gaussian_process`* module implements Gaussian Process based regression and classification.

User guide: See the *Gaussian Processes* section for further details.

`gaussian_process.GaussianProcessClassifier`([kernel, ...])	Gaussian process classification (GPC) based on Laplace approximation.
`gaussian_process.GaussianProcessRegressor`([kernel, ...])	Gaussian process regression (GPR).

6.17.1 `sklearn.gaussian_process.GaussianProcessClassifier`

class `sklearn.gaussian_process.`**`GaussianProcessClassifier`**(*kernel=None*, *optimizer='fmin_l_bfgs_b'*, *n_restarts_optimizer=0*, *max_iter_predict=100*, *warm_start=False*, *copy_X_train=True*, *random_state=None*, *multi_class='one_vs_rest'*, *n_jobs=None*)

Gaussian process classification (GPC) based on Laplace approximation.

The implementation is based on Algorithm 3.1, 3.2, and 5.1 of Gaussian Processes for Machine Learning (GPML) by Rasmussen and Williams.

Internally, the Laplace approximation is used for approximating the non-Gaussian posterior by a Gaussian.

Currently, the implementation is restricted to using the logistic link function. For multi-class classification, several binary one-versus rest classifiers are fitted. Note that this class thus does not implement a true multi-class Laplace approximation.

Parameters

kernel [kernel object] The kernel specifying the covariance function of the GP. If None is passed, the kernel "1.0 * RBF(1.0)" is used as default. Note that the kernel's hyperparameters are optimized during fitting.

optimizer [string or callable, optional (default: "fmin_l_bfgs_b")] Can either be one of the internally supported optimizers for optimizing the kernel's parameters, specified by a string, or an externally defined optimizer passed as a callable. If a callable is passed, it must have the signature:

```
def optimizer(obj_func, initial_theta, bounds):
    # * 'obj_func' is the objective function to be maximized, which
    #   takes the hyperparameters theta as parameter and an
    #   optional flag eval_gradient, which determines if the
    #   gradient is returned additionally to the function value
    # * 'initial_theta': the initial value for theta, which can be
    #   used by local optimizers
    # * 'bounds': the bounds on the values of theta
    ....
    # Returned are the best found hyperparameters theta and
    # the corresponding value of the target function.
    return theta_opt, func_min
```

Per default, the 'fmin_l_bfgs_b' algorithm from scipy.optimize is used. If None is passed, the kernel's parameters are kept fixed. Available internal optimizers are:

```
'fmin_l_bfgs_b'
```

n_restarts_optimizer [int, optional (default: 0)] The number of restarts of the optimizer for finding the kernel's parameters which maximize the log-marginal likelihood. The first run of the optimizer is performed from the kernel's initial parameters, the remaining ones (if any) from thetas sampled log-uniform randomly from the space of allowed theta-values. If greater than 0, all bounds must be finite. Note that n_restarts_optimizer=0 implies that one run is performed.

max_iter_predict [int, optional (default: 100)] The maximum number of iterations in Newton's method for approximating the posterior during predict. Smaller values will reduce computation time at the cost of worse results.

warm_start [bool, optional (default: False)] If warm-starts are enabled, the solution of the last Newton iteration on the Laplace approximation of the posterior mode is used as initialization for the next call of _posterior_mode(). This can speed up convergence when _posterior_mode is called several times on similar problems as in hyperparameter optimization. See *the Glossary*.

copy_X_train [bool, optional (default: True)] If True, a persistent copy of the training data is stored in the object. Otherwise, just a reference to the training data is stored, which might cause predictions to change if the data is modified externally.

random_state [int, RandomState instance or None, optional (default: None)] The generator used to initialize the centers. 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`.

multi_class [string, default] Specifies how multi-class classification problems are handled. Supported are "one_vs_rest" and "one_vs_one". In "one_vs_rest", one binary Gaussian process classifier is fitted for each class, which is trained to separate this class from the rest. In "one_vs_one", one binary Gaussian process classifier is fitted for each pair of classes, which is trained to separate these two classes. The predictions of these binary predictors are combined into multi-class predictions. Note that "one_vs_one" does not support predicting probability estimates.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

kernel_ [kernel object] The kernel used for prediction. In case of binary classification, the structure of the kernel is the same as the one passed as parameter but with optimized hyperparameters. In case of multi-class classification, a CompoundKernel is returned which consists of the different kernels used in the one-versus-rest classifiers.

log_marginal_likelihood_value_ [float] The log-marginal-likelihood of `self.kernel_.theta`

classes_ [array-like, shape = (n_classes,)] Unique class labels.

n_classes_ [int] The number of classes in the training data

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.gaussian_process import GaussianProcessClassifier
>>> from sklearn.gaussian_process.kernels import RBF
>>> X, y = load_iris(return_X_y=True)
>>> kernel = 1.0 * RBF(1.0)
>>> gpc = GaussianProcessClassifier(kernel=kernel,
...         random_state=0).fit(X, y)
>>> gpc.score(X, y)
0.9866...
>>> gpc.predict_proba(X[:2,:])
array([[0.83548752, 0.03228706, 0.13222543],
       [0.79064206, 0.06525643, 0.14410151]])
```

New in version 0.18.

Methods

fit(self, X, y)	Fit Gaussian process classification model
get_params(self[, deep])	Get parameters for this estimator.
log_marginal_likelihood(self[, theta, ...])	Returns log-marginal likelihood of theta for training data.
predict(self, X)	Perform classification on an array of test vectors X.
predict_proba(self, X)	Return probability estimates for the test vector X.

Continued on next page

Table 6.117 – continued from previous page

score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *kernel=None*, *optimizer='fmin_l_bfgs_b'*, *n_restarts_optimizer=0*, *max_iter_predict=100*, *warm_start=False*, *copy_X_train=True*, *random_state=None*, *multi_class='one_vs_rest'*, *n_jobs=None*)

fit (*self*, *X*, *y*)
Fit Gaussian process classification model

Parameters

X [array-like, shape = (n_samples, n_features)] Training data

y [array-like, shape = (n_samples,)] Target values, must be binary

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

log_marginal_likelihood (*self*, *theta=None*, *eval_gradient=False*)
Returns log-marginal likelihood of theta for training data.

In the case of multi-class classification, the mean log-marginal likelihood of the one-versus-rest classifiers are returned.

Parameters

theta [array-like, shape = (n_kernel_params,) or none] Kernel hyperparameters for which the log-marginal likelihood is evaluated. In the case of multi-class classification, theta may be the hyperparameters of the compound kernel or of an individual kernel. In the latter case, all individual kernel get assigned the same theta values. If None, the precomputed log_marginal_likelihood of `self.kernel_.theta` is returned.

eval_gradient [bool, default: False] If True, the gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta is returned additionally. Note that gradient computation is not supported for non-binary classification. If True, theta must not be None.

Returns

log_likelihood [float] Log-marginal likelihood of theta for training data.

log_likelihood_gradient [array, shape = (n_kernel_params,), optional] Gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta. Only returned when eval_gradient is True.

predict (*self*, *X*)
Perform classification on an array of test vectors X.

Parameters

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

Returns

C [array, shape = (n_samples,)] Predicted target values for X, values are from `classes_`

predict_proba(*self*, *X*)

Return probability estimates for the test vector X.

Parameters

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

Returns

C [array-like, shape = (n_samples, n_classes)] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

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

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.gaussian_process.GaussianProcessClassifier`

- *Plot classification probability*
- *Classifier comparison*
- *Illustration of Gaussian process classification (GPC) on the XOR dataset*
- *Gaussian process classification (GPC) on iris dataset*
- *Iso-probability lines for Gaussian Processes classification (GPC)*
- *Probabilistic predictions with Gaussian process classification (GPC)*

6.17.2 sklearn.gaussian_process.GaussianProcessRegressor

class sklearn.gaussian_process.**GaussianProcessRegressor**(*kernel=None*, *alpha=1e-10*, *optimizer='fmin_l_bfgs_b'*, *n_restarts_optimizer=0*, *normalize_y=False*, *copy_X_train=True*, *random_state=None*)

Gaussian process regression (GPR).

The implementation is based on Algorithm 2.1 of Gaussian Processes for Machine Learning (GPML) by Rasmussen and Williams.

In addition to standard scikit-learn estimator API, GaussianProcessRegressor:

- allows prediction without prior fitting (based on the GP prior)
- provides an additional method sample_y(X), which evaluates samples drawn from the GPR (prior or posterior) at given inputs
- exposes a method log_marginal_likelihood(theta), which can be used externally for other ways of selecting hyperparameters, e.g., via Markov chain Monte Carlo.

Read more in the *User Guide*.

New in version 0.18.

Parameters

kernel [kernel object] The kernel specifying the covariance function of the GP. If None is passed, the kernel "1.0 * RBF(1.0)" is used as default. Note that the kernel's hyperparameters are optimized during fitting.

alpha [float or array-like, optional (default: 1e-10)] Value added to the diagonal of the kernel matrix during fitting. Larger values correspond to increased noise level in the observations. This can also prevent a potential numerical issue during fitting, by ensuring that the calculated values form a positive definite matrix. If an array is passed, it must have the same number of entries as the data used for fitting and is used as datapoint-dependent noise level. Note that this is equivalent to adding a WhiteKernel with c=alpha. Allowing to specify the noise level directly as a parameter is mainly for convenience and for consistency with Ridge.

optimizer [string or callable, optional (default: "fmin_l_bfgs_b")] Can either be one of the internally supported optimizers for optimizing the kernel's parameters, specified by a string, or an externally defined optimizer passed as a callable. If a callable is passed, it must have the signature:

```
def optimizer(obj_func, initial_theta, bounds):
    # * 'obj_func' is the objective function to be minimized, which
    #   takes the hyperparameters theta as parameter and an
    #   optional flag eval_gradient, which determines if the
    #   gradient is returned additionally to the function value
    # * 'initial_theta': the initial value for theta, which can be
    #   used by local optimizers
    # * 'bounds': the bounds on the values of theta
    ....
    # Returned are the best found hyperparameters theta and
    # the corresponding value of the target function.
    return theta_opt, func_min
```

Per default, the 'fmin_l_bfgs_b' algorithm from scipy.optimize is used. If None is passed, the kernel's parameters are kept fixed. Available internal optimizers are:

```
'fmin_l_bfgs_b'
```

n_restarts_optimizer [int, optional (default: 0)] The number of restarts of the optimizer for finding the kernel's parameters which maximize the log-marginal likelihood. The first run of the optimizer is performed from the kernel's initial parameters, the remaining ones (if any) from thetas sampled log-uniform randomly from the space of allowed theta-values. If greater than 0, all bounds must be finite. Note that n_restarts_optimizer == 0 implies that one run is performed.

normalize_y [boolean, optional (default: False)] Whether the target values y are normalized, i.e., the mean of the observed target values become zero. This parameter should be set to True if the target values' mean is expected to differ considerable from zero. When enabled, the normalization effectively modifies the GP's prior based on the data, which contradicts the likelihood principle; normalization is thus disabled per default.

copy_X_train [bool, optional (default: True)] If True, a persistent copy of the training data is stored in the object. Otherwise, just a reference to the training data is stored, which might cause predictions to change if the data is modified externally.

random_state [int, RandomState instance or None, optional (default: None)] The generator used to initialize the centers. 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`.

Attributes

X_train_ [array-like, shape = (n_samples, n_features)] Feature values in training data (also required for prediction)

y_train_ [array-like, shape = (n_samples, [n_output_dims])] Target values in training data (also required for prediction)

kernel_ [kernel object] The kernel used for prediction. The structure of the kernel is the same as the one passed as parameter but with optimized hyperparameters

L_ [array-like, shape = (n_samples, n_samples)] Lower-triangular Cholesky decomposition of the kernel in `X_train_`

alpha_ [array-like, shape = (n_samples,)] Dual coefficients of training data points in kernel space

log_marginal_likelihood_value_ [float] The log-marginal-likelihood of `self.kernel_.theta`

Examples

```
>>> from sklearn.datasets import make_friedman2
>>> from sklearn.gaussian_process import GaussianProcessRegressor
>>> from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel
>>> X, y = make_friedman2(n_samples=500, noise=0, random_state=0)
>>> kernel = DotProduct() + WhiteKernel()
>>> gpr = GaussianProcessRegressor(kernel=kernel,
...         random_state=0).fit(X, y)
>>> gpr.score(X, y)
0.3680...
>>> gpr.predict(X[:2,:], return_std=True)
(array([653.0..., 592.1...]), array([316.6..., 316.6...]))
```

Methods

fit(self, X, y)	Fit Gaussian process regression model.
get_params(self[, deep])	Get parameters for this estimator.
log_marginal_likelihood(self[, theta, ...])	Returns log-marginal likelihood of theta for training data.
predict(self, X[, return_std, return_cov])	Predict using the Gaussian process regression model
sample_y(self, X[, n_samples, random_state])	Draw samples from Gaussian process and evaluate at X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *kernel=None*, *alpha=1e-10*, *optimizer='fmin_l_bfgs_b'*, *n_restarts_optimizer=0*, *normalize_y=False*, *copy_X_train=True*, *random_state=None*)

fit (*self*, *X*, *y*)
Fit Gaussian process regression model.

Parameters

X [array-like, shape = (n_samples, n_features)] Training data

y [array-like, shape = (n_samples, [n_output_dims])] Target values

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

log_marginal_likelihood (*self*, *theta=None*, *eval_gradient=False*)
Returns log-marginal likelihood of theta for training data.

Parameters

theta [array-like, shape = (n_kernel_params,) or None] Kernel hyperparameters for which the log-marginal likelihood is evaluated. If None, the precomputed log_marginal_likelihood of `self.kernel_.theta` is returned.

eval_gradient [bool, default: False] If True, the gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta is returned additionally. If True, theta must not be None.

Returns

log_likelihood [float] Log-marginal likelihood of theta for training data.

log_likelihood_gradient [array, shape = (n_kernel_params,), optional] Gradient of the log-marginal likelihood with respect to the kernel hyperparameters at position theta. Only returned when eval_gradient is True.

predict(*self*, *X*, *return_std=False*, *return_cov=False*)
Predict using the Gaussian process regression model

We can also predict based on an unfitted model by using the GP prior. In addition to the mean of the predictive distribution, also its standard deviation (return_std=True) or covariance (return_cov=True). Note that at most one of the two can be requested.

Parameters

X [array-like, shape = (n_samples, n_features)] Query points where the GP is evaluated

return_std [bool, default: False] If True, the standard-deviation of the predictive distribution at the query points is returned along with the mean.

return_cov [bool, default: False] If True, the covariance of the joint predictive distribution at the query points is returned along with the mean

Returns

y_mean [array, shape = (n_samples, [n_output_dims])] Mean of predictive distribution a query points

y_std [array, shape = (n_samples,), optional] Standard deviation of predictive distribution at query points. Only returned when return_std is True.

y_cov [array, shape = (n_samples, n_samples), optional] Covariance of joint predictive distribution a query points. Only returned when return_cov is True.

sample_y(*self*, *X*, *n_samples=1*, *random_state=0*)
Draw samples from Gaussian process and evaluate at X.

Parameters

X [array-like, shape = (n_samples_X, n_features)] Query points where the GP samples are evaluated

n_samples [int, default: 1] The number of samples drawn from the Gaussian process

random_state [int, RandomState instance or None, optional (default=0)] 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`.

Returns

y_samples [array, shape = (n_samples_X, [n_output_dims], n_samples)] Values of n_samples samples drawn from Gaussian process and evaluated at query points.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.gaussian_process.GaussianProcessRegressor`

- *Comparison of kernel ridge and Gaussian process regression*
- *Illustration of prior and posterior Gaussian process for different kernels*
- *Gaussian process regression (GPR) with noise-level estimation*
- *Gaussian Processes regression: basic introductory example*
- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*

Kernels:

gaussian_process.kernels.CompoundKernel(kernels)	Kernel which is composed of a set of other kernels.
gaussian_process.kernels.ConstantKernel([...])	Constant kernel.
gaussian_process.kernels.DotProduct([...])	Dot-Product kernel.
gaussian_process.kernels.ExpSineSquared([...])	Exp-Sine-Squared kernel.
gaussian_process.kernels.Exponentiation(...)	Exponentiate kernel by given exponent.
gaussian_process.kernels.Hyperparameter	A kernel hyperparameter's specification in form of a namedtuple.
gaussian_process.kernels.Kernel	Base class for all kernels.
gaussian_process.kernels.Matern([...])	Matern kernel.
gaussian_process.kernels.PairwiseKernel([...])	Wrapper for kernels in sklearn.metrics.pairwise.
gaussian_process.kernels.Product(k1, k2)	Product-kernel k1 * k2 of two kernels k1 and k2.

Continued on next page

Table 6.119 – continued from previous page

gaussian_process.kernels.RBF([length_scale, ...])	Radial-basis function kernel (aka squared-exponential kernel).
gaussian_process.kernels.RationalQuadratic([...])	Rational Quadratic kernel.
gaussian_process.kernels.Sum(k1, k2)	Sum-kernel k1 + k2 of two kernels k1 and k2.
gaussian_process.kernels.WhiteKernel([...])	White kernel.

6.17.3 sklearn.gaussian_process.kernels.CompoundKernel

class sklearn.gaussian_process.kernels.**CompoundKernel**(*kernels*)

Kernel which is composed of a set of other kernels.

New in version 0.18.

Parameters

kernels [list of Kernel objects] The other kernels

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__(*self*, *kernels*)

__call__(*self*, *X*, *Y=None*, *eval_gradient=False*)

Return the kernel k(X, Y) and optionally its gradient.

Note that this compound kernel returns the results of all simple kernel stacked along an additional axis.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined.

Returns

K [array, shape (n_samples_X, n_samples_Y, n_kernels)] Kernel k(X, Y)

K_gradient [array, shape (n_samples_X, n_samples_X, n_dims, n_kernels)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X, n_kernels)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

6.17.4 sklearn.gaussian_process.kernels.ConstantKernel

class sklearn.gaussian_process.kernels.**ConstantKernel**(*constant_value=1.0*, *constant_value_bounds=(1e-05, 100000.0)*)

Constant kernel.

Can be used as part of a product-kernel where it scales the magnitude of the other factor (kernel) or as part of a sum-kernel, where it modifies the mean of the Gaussian process.

k(x_1, x_2) = constant_value for all x_1, x_2

New in version 0.18.

Parameters

constant_value [float, default: 1.0] The constant value which defines the covariance: k(x_1, x_2) = constant_value

constant_value_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on constant_value

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_constant_value

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__(*self*, *constant_value=1.0*, *constant_value_bounds=(1e-05, 100000.0)*)

__call__(*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.ConstantKernel`

- *Illustration of prior and posterior Gaussian process for different kernels*
- *Iso-probability lines for Gaussian Processes classification (GPC)*
- *Gaussian Processes regression: basic introductory example*

6.17.5 sklearn.gaussian_process.kernels.DotProduct

class sklearn.gaussian_process.kernels.**DotProduct**(*sigma_0=1.0*, *sigma_0_bounds=(1e-05, 100000.0)*)
Dot-Product kernel.

The DotProduct kernel is non-stationary and can be obtained from linear regression by putting N(0, 1) priors on the coefficients of x_d (d = 1, . . . , D) and a prior of N(0, sigma_0^2) on the bias. The DotProduct kernel is invariant to a rotation of the coordinates about the origin, but not translations. It is parameterized by a parameter sigma_0^2. For sigma_0^2 =0, the kernel is called the homogeneous linear kernel, otherwise it is inhomogeneous. The kernel is given by

k(x_i, x_j) = sigma_0 ^ 2 + x_i cdot x_j

The DotProduct kernel is commonly combined with exponentiation.

New in version 0.18.

Parameters

sigma_0 [float >= 0, default: 1.0] Parameter controlling the inhomogenity of the kernel. If sigma_0=0, the kernel is homogenous.

sigma_0_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on l

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_sigma_0

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *sigma_0=1.0*, *sigma_0_bounds=(1e-05*, *100000.0)*)

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag (*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.DotProduct`

- *Illustration of Gaussian process classification (GPC) on the XOR dataset*
- *Illustration of prior and posterior Gaussian process for different kernels*
- *Iso-probability lines for Gaussian Processes classification (GPC)*

6.17.6 sklearn.gaussian_process.kernels.ExpSineSquared

class sklearn.gaussian_process.kernels.**ExpSineSquared**(*length_scale=1.0*, *periodicity=1.0*, *length_scale_bounds=(1e-05, 100000.0)*, *periodicity_bounds=(1e-05, 100000.0)*)
Exp-Sine-Squared kernel.

The ExpSineSquared kernel allows modeling periodic functions. It is parameterized by a length-scale parameter length_scale>0 and a periodicity parameter periodicity>0. Only the isotropic variant where l is a scalar is supported at the moment. The kernel given by:

k(x_i, x_j) = exp(-2 (sin(pi / periodicity * d(x_i, x_j)) / length_scale) ^ 2)

New in version 0.18.

Parameters

length_scale [float > 0, default: 1.0] The length scale of the kernel.

periodicity [float > 0, default: 1.0] The periodicity of the kernel.

length_scale_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on length_scale

periodicity_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on periodicity

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_length_scale

hyperparameter_periodicity

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__(*self*, *length_scale=1.0*, *periodicity=1.0*, *length_scale_bounds=(1e-05, 100000.0)*, *periodicity_bounds=(1e-05, 100000.0)*)

__call__(*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.ExpSineSquared`

- *Comparison of kernel ridge and Gaussian process regression*
- *Illustration of prior and posterior Gaussian process for different kernels*
- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*

6.17.7 `sklearn.gaussian_process.kernels`.Exponentiation

class sklearn.gaussian_process.kernels.**Exponentiation**(*kernel*, *exponent*)
Exponentiate kernel by given exponent.

The resulting kernel is defined as k_exp(X, Y) = k(X, Y) ** exponent

New in version 0.18.

Parameters

kernel [Kernel object] The base kernel

exponent [float] The exponent for the base kernel

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameters Returns a list of all hyperparameter.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__(*self*, *kernel*, *exponent*)

__call__(*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

6.17.8 sklearn.gaussian_process.kernels.Hyperparameter

class sklearn.gaussian_process.kernels.**Hyperparameter**
A kernel hyperparameter's specification in form of a namedtuple.

New in version 0.18.

Attributes

name [string] Alias for field number 0

value_type [string] Alias for field number 1

bounds [pair of floats >= 0 or "fixed"] Alias for field number 2

n_elements [int, default=1] Alias for field number 3

fixed [bool, default: None] Alias for field number 4

Methods

count()	
index()	Raises ValueError if the value is not present.

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

__call__(**args*, ***kwargs*)
Call self as a function.

bounds
Alias for field number 2

count()

fixed
Alias for field number 4

index()

Raises ValueError if the value is not present.

n_elements
Alias for field number 3

name
Alias for field number 0

value_type
Alias for field number 1

6.17.9 sklearn.gaussian_process.kernels.Kernel

class sklearn.gaussian_process.kernels.**Kernel**
Base class for all kernels.

New in version 0.18.

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Evaluate the kernel.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

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

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Evaluate the kernel.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

6.17.10 sklearn.gaussian_process.kernels.Matern

class sklearn.gaussian_process.kernels.**Matern**(*length_scale=1.0*, *length_scale_bounds=(1e-05*, *100000.0)*, *nu=1.5*)
Matern kernel.

The class of Matern kernels is a generalization of the RBF and the absolute exponential kernel parameterized by an additional parameter nu. The smaller nu, the less smooth the approximated function is. For nu=inf, the kernel

becomes equivalent to the RBF kernel and for nu=0.5 to the absolute exponential kernel. Important intermediate values are nu=1.5 (once differentiable functions) and nu=2.5 (twice differentiable functions).

See Rasmussen and Williams 2006, pp84 for details regarding the different variants of the Matern kernel.

New in version 0.18.

Parameters

length_scale [float or array with shape (n_features,), default: 1.0] The length scale of the kernel. If a float, an isotropic kernel is used. If an array, an anisotropic kernel is used where each dimension of l defines the length-scale of the respective feature dimension.

length_scale_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on length_scale

nu [float, default: 1.5] The parameter nu controlling the smoothness of the learned function. The smaller nu, the less smooth the approximated function is. For nu=inf, the kernel becomes equivalent to the RBF kernel and for nu=0.5 to the absolute exponential kernel. Important intermediate values are nu=1.5 (once differentiable functions) and nu=2.5 (twice differentiable functions). Note that values of nu not in [0.5, 1.5, 2.5, inf] incur a considerably higher computational cost (appr. 10 times higher) since they require to evaluate the modified Bessel function. Furthermore, in contrast to l, nu is kept fixed to its initial value and not optimized.

Attributes

anisotropic

bounds Returns the log-transformed bounds on the theta.

hyperparameter_length_scale

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *length_scale=1.0*, *length_scale_bounds=(1e-05*, *100000.0)*, *nu=1.5*)

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta

Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.Matern`

- *Illustration of prior and posterior Gaussian process for different kernels*

6.17.11 `sklearn.gaussian_process.kernels`.PairwiseKernel

class sklearn.gaussian_process.kernels.**PairwiseKernel**(*gamma=1.0*, *gamma_bounds=(1e-05, 100000.0)*, *metric='linear'*, *pairwise_kernels_kwargs=None*)

Wrapper for kernels in sklearn.metrics.pairwise.

A thin wrapper around the functionality of the kernels in sklearn.metrics.pairwise.

Note: Evaluation of eval_gradient is not analytic but numeric and all kernels support only isotropic distances. The parameter gamma is considered to be a hyperparameter and may be optimized. The other kernel parameters are set directly at initialization and are kept fixed.

New in version 0.18.

Parameters

gamma [float >= 0, default: 1.0] Parameter gamma of the pairwise kernel specified by metric

gamma_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on gamma

metric [string, or callable, default: "linear"] The metric to use when calculating kernel between instances in a feature array. If metric is a string, it must be one of the metrics in pairwise.PAIRWISE_KERNEL_FUNCTIONS. If metric is "precomputed", X is assumed to be a kernel matrix. Alternatively, 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 from X as input and return a value indicating the distance between them.

pairwise_kernels_kwargs [dict, default: None] All entries of this dict (if any) are passed as keyword arguments to the pairwise kernel function.

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_gamma

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *gamma=1.0*, *gamma_bounds=(1e-05*, *100000.0)*, *metric='linear'*, *pairwise_kernels_kwargs=None*)

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)

Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds

Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)

Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag (*self*, *X*)

Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

6.17.12 sklearn.gaussian_process.kernels.Product

class sklearn.gaussian_process.kernels.**Product**(*k1*, *k2*)
Product-kernel k1 * k2 of two kernels k1 and k2.

The resulting kernel is defined as k_prod(X, Y) = k1(X, Y) * k2(X, Y)

New in version 0.18.

Parameters

k1 [Kernel object] The first base-kernel of the product-kernel

k2 [Kernel object] The second base-kernel of the product-kernel

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameters Returns a list of all hyperparameter.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *k1*, *k2*)

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag (*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

6.17.13 sklearn.gaussian_process.kernels.RBF

class sklearn.gaussian_process.kernels.**RBF**(*length_scale=1.0*, *length_scale_bounds=(1e-05, 100000.0)*)
Radial-basis function kernel (aka squared-exponential kernel).

The RBF kernel is a stationary kernel. It is also known as the "squared exponential" kernel. It is parameterized by a length-scale parameter length_scale>0, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs X (anisotropic variant of the kernel). The kernel is given by:

k(x_i, x_j) = exp(-1 / 2 d(x_i / length_scale, x_j / length_scale)^2)

This kernel is infinitely differentiable, which implies that GPs with this kernel as covariance function have mean square derivatives of all orders, and are thus very smooth.

New in version 0.18.

Parameters

length_scale [float or array with shape (n_features,), default: 1.0] The length scale of the kernel. If a float, an isotropic kernel is used. If an array, an anisotropic kernel is used where each dimension of l defines the length-scale of the respective feature dimension.

length_scale_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on length_scale

Attributes

anisotropic

bounds Returns the log-transformed bounds on the theta.

hyperparameter_length_scale

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *length_scale=1.0*, *length_scale_bounds=(1e-05*, *100000.0)*)

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.RBF`

- *Plot classification probability*
- *Classifier comparison*
- *Illustration of Gaussian process classification (GPC) on the XOR dataset*
- *Gaussian process classification (GPC) on iris dataset*
- *Illustration of prior and posterior Gaussian process for different kernels*
- *Probabilistic predictions with Gaussian process classification (GPC)*
- *Gaussian process regression (GPR) with noise-level estimation*
- *Gaussian Processes regression: basic introductory example*
- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*

6.17.14 `sklearn.gaussian_process.kernels.RationalQuadratic`

class `sklearn.gaussian_process.kernels.`**`RationalQuadratic`**(*length_scale=1.0*, *alpha=1.0*, *length_scale_bounds=(1e-05, 100000.0)*, *alpha_bounds=(1e-05, 100000.0)*)

Rational Quadratic kernel.

The RationalQuadratic kernel can be seen as a scale mixture (an infinite sum) of RBF kernels with different characteristic length-scales. It is parameterized by a length-scale parameter length_scale>0 and a scale mixture parameter alpha>0. Only the isotropic variant where length_scale is a scalar is supported at the moment. The kernel given by:

k(x_i, x_j) = (1 + d(x_i, x_j)^2 / (2*alpha * length_scale^2))^-alpha

New in version 0.18.

Parameters

length_scale [float > 0, default: 1.0] The length scale of the kernel.

alpha [float > 0, default: 1.0] Scale mixture parameter

length_scale_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on length_scale

alpha_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on alpha

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_alpha

hyperparameter_length_scale

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *length_scale=1.0*, *alpha=1.0*, *length_scale_bounds=(1e-05*, *100000.0)*, *alpha_bounds=(1e-05*, *100000.0)*)

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)

Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds

Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)

Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag (*self*, *X*)

Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.RationalQuadratic`

- *Illustration of prior and posterior Gaussian process for different kernels*
- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*

6.17.15 sklearn.gaussian_process.kernels.Sum

class sklearn.gaussian_process.kernels.**Sum**(*k1*, *k2*)
Sum-kernel k1 + k2 of two kernels k1 and k2.

The resulting kernel is defined as k_sum(X, Y) = k1(X, Y) + k2(X, Y)

New in version 0.18.

Parameters

k1 [Kernel object] The first base-kernel of the sum-kernel

k2 [Kernel object] The second base-kernel of the sum-kernel

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameters Returns a list of all hyperparameter.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *k1*, *k2*)

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta (*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag (*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

6.17.16 sklearn.gaussian_process.kernels.WhiteKernel

class sklearn.gaussian_process.kernels.**WhiteKernel**(*noise_level=1.0*, *noise_level_bounds=(1e-05*, *100000.0)*)

White kernel.

The main use-case of this kernel is as part of a sum-kernel where it explains the noise-component of the signal. Tuning its parameter corresponds to estimating the noise-level.

k(x_1, x_2) = noise_level if x_1 == x_2 else 0

New in version 0.18.

Parameters

noise_level [float, default: 1.0] Parameter controlling the noise level

noise_level_bounds [pair of floats >= 0, default: (1e-5, 1e5)] The lower and upper bound on noise_level

Attributes

bounds Returns the log-transformed bounds on the theta.

hyperparameter_noise_level

hyperparameters Returns a list of all hyperparameter specifications.

n_dims Returns the number of non-fixed hyperparameters of the kernel.

theta Returns the (flattened, log-transformed) non-fixed hyperparameters.

Methods

__call__(self, X[, Y, eval_gradient])	Return the kernel k(X, Y) and optionally its gradient.
clone_with_theta(self, theta)	Returns a clone of self with given hyperparameters theta.
diag(self, X)	Returns the diagonal of the kernel k(X, X).
get_params(self[, deep])	Get parameters of this kernel.
is_stationary(self)	Returns whether the kernel is stationary.
set_params(self, **params)	Set the parameters of this kernel.

__init__ (*self*, *noise_level=1.0*, *noise_level_bounds=(1e-05*, *100000.0)*)

__call__ (*self*, *X*, *Y=None*, *eval_gradient=False*)
Return the kernel k(X, Y) and optionally its gradient.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Y [array, shape (n_samples_Y, n_features), (optional, default=None)] Right argument of the returned kernel k(X, Y). If None, k(X, X) if evaluated instead.

eval_gradient [bool (optional, default=False)] Determines whether the gradient with respect to the kernel hyperparameter is determined. Only supported when Y is None.

Returns

K [array, shape (n_samples_X, n_samples_Y)] Kernel k(X, Y)

K_gradient [array (opt.), shape (n_samples_X, n_samples_X, n_dims)] The gradient of the kernel k(X, X) with respect to the hyperparameter of the kernel. Only returned when eval_gradient is True.

bounds
Returns the log-transformed bounds on the theta.

Returns

bounds [array, shape (n_dims, 2)] The log-transformed bounds on the kernel's hyperparameters theta

clone_with_theta(*self*, *theta*)
Returns a clone of self with given hyperparameters theta.

Parameters

theta [array, shape (n_dims,)] The hyperparameters

diag(*self*, *X*)
Returns the diagonal of the kernel k(X, X).

The result of this method is identical to np.diag(self(X)); however, it can be evaluated more efficiently since only the diagonal is evaluated.

Parameters

X [array, shape (n_samples_X, n_features)] Left argument of the returned kernel k(X, Y)

Returns

K_diag [array, shape (n_samples_X,)] Diagonal of kernel k(X, X)

get_params(*self*, *deep=True*)
Get parameters of this kernel.

Parameters

deep [boolean, optional] 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.

hyperparameters
Returns a list of all hyperparameter specifications.

is_stationary(*self*)
Returns whether the kernel is stationary.

n_dims
Returns the number of non-fixed hyperparameters of the kernel.

set_params(*self*, ***params*)
Set the parameters of this kernel.

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

Returns

self

theta
Returns the (flattened, log-transformed) non-fixed hyperparameters.

Note that theta are typically the log-transformed values of the kernel's hyperparameters as this representation of the search space is more amenable for hyperparameter search, as hyperparameters like length-scales naturally live on a log-scale.

Returns

theta [array, shape (n_dims,)] The non-fixed, log-transformed hyperparameters of the kernel

Examples using `sklearn.gaussian_process.kernels.WhiteKernel`

- *Comparison of kernel ridge and Gaussian process regression*
- *Gaussian process regression (GPR) with noise-level estimation*
- *Gaussian process regression (GPR) on Mauna Loa CO2 data.*

6.18 `sklearn.isotonic`: Isotonic regression

User guide: See the *Isotonic regression* section for further details.

isotonic.IsotonicRegression([y_min, y_max, ...])	Isotonic regression model.

6.18.1 `sklearn.isotonic`.IsotonicRegression

class `sklearn.isotonic.`**`IsotonicRegression`**(*y_min=None*, *y_max=None*, *increasing=True*, *out_of_bounds='nan'*)

Isotonic regression model.

The isotonic regression optimization problem is defined by:

```
min sum w_i (y[i] - y_[i]) ** 2

subject to y_[i] <= y_[j] whenever X[i] <= X[j]
and min(y_) = y_min, max(y_) = y_max
```

where:

- `y[i]` are inputs (real numbers)
- `y_[i]` are fitted
- `X` specifies the order. If `X` is non-decreasing then `y_` is non-decreasing.
- `w[i]` are optional strictly positive weights (default to 1.0)

Read more in the *User Guide*.

Parameters

y_min [optional, default: None] If not None, set the lowest value of the fit to y_min.

y_max [optional, default: None] If not None, set the highest value of the fit to y_max.

increasing [boolean or string, optional, default: True] If boolean, whether or not to fit the isotonic regression with y increasing or decreasing.

The string value "auto" determines whether y should increase or decrease based on the Spearman correlation estimate's sign.

out_of_bounds [string, optional, default: "nan"] The `out_of_bounds` parameter handles how x-values outside of the training domain are handled. When set to "nan", predicted y-values will be NaN. When set to "clip", predicted y-values will be set to the value corresponding to the nearest train interval endpoint. When set to "raise", allow `interp1d` to throw ValueError.

Attributes

X_min_ [float] Minimum value of input array `X_` for left bound.

X_max_ [float] Maximum value of input array `X_` for right bound.

f_ [function] The stepwise interpolating function that covers the input domain `X`.

Notes

Ties are broken using the secondary method from Leeuw, 1977.

References

Isotonic Median Regression: A Linear Programming Approach Nilotpal Chakravarti Mathematics of Operations Research Vol. 14, No. 2 (May, 1989), pp. 303-308

Isotone Optimization in R : Pool-Adjacent-Violators Algorithm (PAVA) and Active Set Methods Leeuw, Hornik, Mair Journal of Statistical Software 2009

Correctness of Kruskal's algorithms for monotone regression with ties Leeuw, Psychometrica, 1977

Examples

```
>>> from sklearn.datasets import make_regression
>>> from sklearn.isotonic import IsotonicRegression
>>> X, y = make_regression(n_samples=10, n_features=1, random_state=41)
>>> iso_reg = IsotonicRegression().fit(X.flatten(), y)
>>> iso_reg.predict([.1, .2])
array([1.8628..., 3.7256...])
```

Methods

fit(self, X, y[, sample_weight])	Fit the model using X, y as training data.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, T)	Predict new data by linear interpolation.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, T)	Transform new data by linear interpolation

__init__ (*self*, *y_min=None*, *y_max=None*, *increasing=True*, *out_of_bounds='nan'*)

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the model using X, y as training data.

Parameters

X [array-like, shape=(n_samples,)] Training data.

y [array-like, shape=(n_samples,)] Training target.

sample_weight [array-like, shape=(n_samples,), optional, default: None] Weights. If set to None, all weights will be set to 1 (equal weights).

Returns

self [object] Returns an instance of self.

Notes

X is stored for future use, as `transform` needs X to interpolate new input data.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *T*)
Predict new data by linear interpolation.

Parameters

T [array-like, shape=(n_samples,)] Data to transform.

Returns

T_ [array, shape=(n_samples,)] Transformed data.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

transform(*self*, *T*)
Transform new data by linear interpolation

Parameters

T [array-like, shape=(n_samples,)] Data to transform.

Returns

T_ [array, shape=(n_samples,)] The transformed data

Examples using `sklearn.isotonic.IsotonicRegression`

- *Isotonic Regression*

isotonic.check_increasing(x, y)	Determine whether y is monotonically correlated with x.
isotonic.isotonic_regression(y[, ...])	Solve the isotonic regression model:

6.18.2 sklearn.isotonic.check_increasing

sklearn.isotonic.**check_increasing**(*x*, *y*)
Determine whether y is monotonically correlated with x.

y is found increasing or decreasing with respect to x based on a Spearman correlation test.

Parameters

x [array-like, shape=(n_samples,)] Training data.

y [array-like, shape=(n_samples,)] Training target.

Returns

increasing_bool [boolean] Whether the relationship is increasing or decreasing.

Notes

The Spearman correlation coefficient is estimated from the data, and the sign of the resulting estimate is used as the result.

In the event that the 95% confidence interval based on Fisher transform spans zero, a warning is raised.

References

Fisher transformation. Wikipedia. https://en.wikipedia.org/wiki/Fisher_transformation

6.18.3 sklearn.isotonic.isotonic_regression

sklearn.isotonic.**isotonic_regression**(*y*, *sample_weight=None*, *y_min=None*, *y_max=None*, *increasing=True*)

Solve the isotonic regression model:

```
min sum w[i] (y[i] - y_[i]) ** 2

subject to y_min = y_[1] <= y_[2] ... <= y_[n] = y_max
```

where:

- y[i] are inputs (real numbers)
- y_[i] are fitted
- w[i] are optional strictly positive weights (default to 1.0)

Read more in the *User Guide*.

Parameters

y [iterable of floats] The data.

sample_weight [iterable of floats, optional, default: None] Weights on each point of the regression. If None, weight is set to 1 (equal weights).

y_min [optional, default: None] If not None, set the lowest value of the fit to y_min.

y_max [optional, default: None] If not None, set the highest value of the fit to y_max.

increasing [boolean, optional, default: True] Whether to compute `y_` is increasing (if set to True) or decreasing (if set to False)

Returns

y_ [list of floats] Isotonic fit of y.

References

"Active set algorithms for isotonic regression; A unifying framework" by Michael J. Best and Nilotpal Chakravarti, section 3.

6.19 sklearn.impute: Impute

Transformers for missing value imputation

User guide: See the *Imputation of missing values* section for further details.

impute.SimpleImputer([missing_values, ...])	Imputation transformer for completing missing values.
impute.IterativeImputer([estimator, ...])	Multivariate imputer that estimates each feature from all the others.
impute.MissingIndicator([missing_values, ...])	Binary indicators for missing values.

6.19.1 sklearn.impute.SimpleImputer

class sklearn.impute.**SimpleImputer**(*missing_values=nan*, *strategy='mean'*, *fill_value=None*, *verbose=0*, *copy=True*, *add_indicator=False*)

Imputation transformer for completing missing values.

Read more in the *User Guide*.

Parameters

missing_values [number, string, np.nan (default) or None] The placeholder for the missing values. All occurrences of `missing_values` will be imputed.

strategy [string, optional (default="mean")] The imputation strategy.

- If "mean", then replace missing values using the mean along each column. Can only be used with numeric data.
- If "median", then replace missing values using the median along each column. Can only be used with numeric data.
- If "most_frequent", then replace missing using the most frequent value along each column. Can be used with strings or numeric data.
- If "constant", then replace missing values with fill_value. Can be used with strings or numeric data.

New in version 0.20: strategy="constant" for fixed value imputation.

fill_value [string or numerical value, optional (default=None)] When strategy == "constant", fill_value is used to replace all occurrences of missing_values. If left to the default, fill_value will be 0 when imputing numerical data and "missing_value" for strings or object data types.

verbose [integer, optional (default=0)] Controls the verbosity of the imputer.

copy [boolean, optional (default=True)] If True, a copy of X will be created. If False, imputation will be done in-place whenever possible. Note that, in the following cases, a new copy will always be made, even if `copy=False`:

- If X is not an array of floating values;
- If X is encoded as a CSR matrix;
- If add_indicator=True.

add_indicator [boolean, optional (default=False)] If True, a *MissingIndicator* transform will stack onto output of the imputer's transform. This allows a predictive estimator to account for missingness despite imputation. If a feature has no missing values at fit/train

time, the feature won't appear on the missing indicator even if there are missing values at transform/test time.

Attributes

statistics_ [array of shape (n_features,)] The imputation fill value for each feature.

indicator_ [*sklearn.impute.MissingIndicator*] Indicator used to add binary indicators for missing values. `None` if add_indicator is False.

See also:

IterativeImputer Multivariate imputation of missing values.

Notes

Columns which only contained missing values at *fit* are discarded upon *transform* if strategy is not "constant".

Examples

```
>>> import numpy as np
>>> from sklearn.impute import SimpleImputer
>>> imp_mean = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imp_mean.fit([[7, 2, 3], [4, np.nan, 6], [10, 5, 9]])
...
SimpleImputer(add_indicator=False, copy=True, fill_value=None,
        missing_values=nan, strategy='mean', verbose=0)
>>> X = [[np.nan, 2, 3], [4, np.nan, 6], [10, np.nan, 9]]
>>> print(imp_mean.transform(X))
...
[[ 7.   2.   3. ]
 [ 4.   3.5  6. ]
 [10.   3.5  9. ]]
```

Methods

fit(self, X[, y])	Fit the imputer on X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Impute all missing values in X.

__init__(*self*, *missing_values=nan*, *strategy='mean'*, *fill_value=None*, *verbose=0*, *copy=True*, *add_indicator=False*)

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

Fit the imputer on X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

self [SimpleImputer]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Impute all missing values in X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data to complete.

Examples using `sklearn.impute.SimpleImputer`

- *Column Transformer with Mixed Types*
- *Imputing missing values with variants of IterativeImputer*
- *Imputing missing values before building an estimator*

6.19.2 sklearn.impute.IterativeImputer

class sklearn.impute.**IterativeImputer**(*estimator=None*, *missing_values=nan*, *sample_posterior=False*, *max_iter=10*, *tol=0.001*, *n_nearest_features=None*, *initial_strategy='mean'*, *imputation_order='ascending'*, *min_value=None*, *max_value=None*, *verbose=0*, *random_state=None*, *add_indicator=False*)

Multivariate imputer that estimates each feature from all the others.

A strategy for imputing missing values by modeling each feature with missing values as a function of other features in a round-robin fashion.

Read more in the *User Guide*.

Note: This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import `enable_iterative_imputer`:

```
>>> # explicitly require this experimental feature
>>> from sklearn.experimental import enable_iterative_imputer  # noqa
>>> # now you can import normally from sklearn.impute
>>> from sklearn.impute import IterativeImputer
```

Parameters

estimator [estimator object, default=BayesianRidge()] The estimator to use at each step of the round-robin imputation. If `sample_posterior` is True, the estimator must support `return_std` in its `predict` method.

missing_values [int, np.nan, optional (default=np.nan)] The placeholder for the missing values. All occurrences of `missing_values` will be imputed.

sample_posterior [boolean, default=False] Whether to sample from the (Gaussian) predictive posterior of the fitted estimator for each imputation. Estimator must support `return_std` in its `predict` method if set to `True`. Set to `True` if using `IterativeImputer` for multiple imputations.

max_iter [int, optional (default=10)] Maximum number of imputation rounds to perform before returning the imputations computed during the final round. A round is a single imputation of each feature with missing values. The stopping criterion is met once `abs(max(X_t - X_{t-1}))/abs(max(X[known_vals]))` < tol, where `X_t` is *X* at iteration `t. Note that early stopping is only applied if ``sample_posterior=False``.`

tol [float, optional (default=1e-3)] Tolerance of the stopping condition.

n_nearest_features [int, optional (default=None)] Number of other features to use to estimate the missing values of each feature column. Nearness between features is measured using the absolute correlation coefficient between each feature pair (after initial imputation). To ensure coverage of features throughout the imputation process, the neighbor features are not necessarily nearest, but are drawn with probability proportional to correlation for each imputed target feature. Can provide significant speed-up when the number of features is huge. If `None`, all features will be used.

initial_strategy [str, optional (default="mean")] Which strategy to use to initialize the missing values. Same as the `strategy` parameter in *sklearn.impute.SimpleImputer* Valid values: {"mean", "median", "most_frequent", or "constant"}.

imputation_order [str, optional (default="ascending")] The order in which the features will be imputed. Possible values:

"ascending" From features with fewest missing values to most.

"descending" From features with most missing values to fewest.

"roman" Left to right.

"arabic" Right to left.

"random" A random order for each round.

min_value [float, optional (default=None)] Minimum possible imputed value. Default of `None` will set minimum to negative infinity.

max_value [float, optional (default=None)] Maximum possible imputed value. Default of `None` will set maximum to positive infinity.

verbose [int, optional (default=0)] Verbosity flag, controls the debug messages that are issued as functions are evaluated. The higher, the more verbose. Can be 0, 1, or 2.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use. Randomizes selection of estimator features if n_nearest_features is not None, the `imputation_order` if `random`, and the sampling from posterior if `sample_posterior` is True. Use an integer for determinism. See *the Glossary*.

add_indicator [boolean, optional (default=False)] If True, a *`MissingIndicator`* transform will stack onto output of the imputer's transform. This allows a predictive estimator to account for missingness despite imputation. If a feature has no missing values at fit/train time, the feature won't appear on the missing indicator even if there are missing values at transform/test time.

Attributes

initial_imputer_ [object of type *`sklearn.impute.SimpleImputer`*] Imputer used to initialize the missing values.

imputation_sequence_ [list of tuples] Each tuple has `(feat_idx, neighbor_feat_idx, estimator)`, where `feat_idx` is the current feature to be imputed, `neighbor_feat_idx` is the array of other features used to impute the current feature, and `estimator` is the trained estimator used for the imputation. Length is `self.n_features_with_missing_ * self.n_iter_`.

n_iter_ [int] Number of iteration rounds that occurred. Will be less than `self.max_iter` if early stopping criterion was reached.

n_features_with_missing_ [int] Number of features with missing values.

indicator_ [*`sklearn.impute.MissingIndicator`*] Indicator used to add binary indicators for missing values. `None` if add_indicator is False.

See also:

`SimpleImputer` Univariate imputation of missing values.

Notes

To support imputation in inductive mode we store each feature's estimator during the `fit` phase, and predict without refitting (in order) during the `transform` phase.

Features which contain all missing values at `fit` are discarded upon `transform`.

Features with missing values during `transform` which did not have any missing values during `fit` will be imputed with the initial imputation method only.

References

[Rcd31b817a31e-1], *[Rcd31b817a31e-2]*

Methods

fit(self, X[, y])	Fits the imputer on X and return self.
fit_transform(self, X[, y])	Fits the imputer on X and return the transformed X.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Imputes all missing values in X.

__init__(*self*, *estimator=None*, *missing_values=nan*, *sample_posterior=False*, *max_iter=10*, *tol=0.001*, *n_nearest_features=None*, *initial_strategy='mean'*, *imputation_order='ascending'*, *min_value=None*, *max_value=None*, *verbose=0*, *random_state=None*, *add_indicator=False*)

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

Fits the imputer on X and return self.

Parameters

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

y [ignored]

Returns

self [object] Returns self.

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

Fits the imputer on X and return the transformed X.

Parameters

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

y [ignored.]

Returns

Xt [array-like, shape (n_samples, n_features)] The imputed input data.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Imputes all missing values in X.

Note that this is stochastic, and that if random_state is not fixed, repeated calls, or permuted input, will yield different results.

Parameters

X [array-like, shape = [n_samples, n_features]] The input data to complete.

Returns

Xt [array-like, shape (n_samples, n_features)] The imputed input data.

Examples using `sklearn.impute.IterativeImputer`

- *Imputing missing values with variants of IterativeImputer*
- *Imputing missing values before building an estimator*

6.19.3 `sklearn.impute`.MissingIndicator

class `sklearn.impute.`**MissingIndicator**(*missing_values=nan*, *features='missing-only'*, *sparse='auto'*, *error_on_new=True*)
Binary indicators for missing values.

Note that this component typically should not be used in a vanilla `Pipeline` consisting of transformers and a classifier, but rather could be added using a `FeatureUnion` or `ColumnTransformer`.

Read more in the *User Guide*.

Parameters

missing_values [number, string, np.nan (default) or None] The placeholder for the missing values. All occurrences of `missing_values` will be indicated (True in the output array), the other values will be marked as False.

features [str, optional] Whether the imputer mask should represent all or a subset of features.

- If "missing-only" (default), the imputer mask will only represent features containing missing values during fit time.
- If "all", the imputer mask will represent all features.

sparse [boolean or "auto", optional] Whether the imputer mask format should be sparse or dense.

- If "auto" (default), the imputer mask will be of same type as input.
- If True, the imputer mask will be a sparse matrix.

- If False, the imputer mask will be a numpy array.

error_on_new [boolean, optional] If True (default), transform will raise an error when there are features with missing values in transform that have no missing values in fit. This is applicable only when `features="missing-only"`.

Attributes

features_ [ndarray, shape (n_missing_features,) or (n_features,)] The features indices which will be returned when calling `transform`. They are computed during `fit`. For `features='all'`, it is to `range(n_features)`.

Examples

```
>>> import numpy as np
>>> from sklearn.impute import MissingIndicator
>>> X1 = np.array([[np.nan, 1, 3],
...                [4, 0, np.nan],
...                [8, 1, 0]])
>>> X2 = np.array([[5, 1, np.nan],
...                [np.nan, 2, 3],
...                [2, 4, 0]])
>>> indicator = MissingIndicator()
>>> indicator.fit(X1)
MissingIndicator(error_on_new=True, features='missing-only',
         missing_values=nan, sparse='auto')
>>> X2_tr = indicator.transform(X2)
>>> X2_tr
array([[False,  True],
       [ True, False],
       [False, False]])
```

Methods

fit(self, X[, y])	Fit the transformer on X.
fit_transform(self, X[, y])	Generate missing values indicator for X.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Generate missing values indicator for X.

__init__ (*self*, *missing_values=nan*, *features='missing-only'*, *sparse='auto'*, *error_on_new=True*)

fit (*self*, *X*, *y=None*)
Fit the transformer on X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

self [object] Returns self.

fit_transform (*self*, *X*, *y=None*)
Generate missing values indicator for X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data to complete.

Returns

Xt [{ndarray or sparse matrix}, shape (n_samples, n_features)] The missing indicator for input data. The data type of `Xt` will be boolean.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Generate missing values indicator for X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data to complete.

Returns

Xt [{ndarray or sparse matrix}, shape (n_samples, n_features)] The missing indicator for input data. The data type of `Xt` will be boolean.

Examples using `sklearn.impute.MissingIndicator`

- *Imputing missing values before building an estimator*

6.20 `sklearn.kernel_approximation` Kernel Approximation

The *`sklearn.kernel_approximation`* module implements several approximate kernel feature maps base on Fourier transforms.

User guide: See the *Kernel Approximation* section for further details.

`kernel_approximation.AdditiveChi2Sampler`([...])	Approximate feature map for additive chi2 kernel.
`kernel_approximation.Nystroem`([kernel, ...])	Approximate a kernel map using a subset of the training data.

Continued on next page

Table 6.141 – continued from previous page

kernel_approximation.RBFSampler([gamma, ...])	Approximates feature map of an RBF kernel by Monte Carlo approximation of its Fourier transform.
kernel_approximation.SkewedChi2Sampler([...])	Approximates feature map of the "skewed chi-squared" kernel by Monte Carlo approximation of its Fourier transform.

6.20.1 sklearn.kernel_approximation.AdditiveChi2Sampler

class sklearn.kernel_approximation.**AdditiveChi2Sampler**(*sample_steps=2*, *sample_interval=None*)

Approximate feature map for additive chi2 kernel.

Uses sampling the fourier transform of the kernel characteristic at regular intervals.

Since the kernel that is to be approximated is additive, the components of the input vectors can be treated separately. Each entry in the original space is transformed into 2*sample_steps+1 features, where sample_steps is a parameter of the method. Typical values of sample_steps include 1, 2 and 3.

Optimal choices for the sampling interval for certain data ranges can be computed (see the reference). The default values should be reasonable.

Read more in the *User Guide*.

Parameters

sample_steps [int, optional] Gives the number of (complex) sampling points.

sample_interval [float, optional] Sampling interval. Must be specified when sample_steps not in {1,2,3}.

See also:

SkewedChi2Sampler A Fourier-approximation to a non-additive variant of the chi squared kernel.

sklearn.metrics.pairwise.chi2_kernel The exact chi squared kernel.

sklearn.metrics.pairwise.additive_chi2_kernel The exact additive chi squared kernel.

Notes

This estimator approximates a slightly different version of the additive chi squared kernel then `metric.additive_chi2` computes.

References

See "Efficient additive kernels via explicit feature maps" A. Vedaldi and A. Zisserman, Pattern Analysis and Machine Intelligence, 2011

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.linear_model import SGDClassifier
>>> from sklearn.kernel_approximation import AdditiveChi2Sampler
>>> X, y = load_digits(return_X_y=True)
```

```
>>> chi2sampler = AdditiveChi2Sampler(sample_steps=2)
>>> X_transformed = chi2sampler.fit_transform(X, y)
>>> clf = SGDClassifier(max_iter=5, random_state=0, tol=1e-3)
>>> clf.fit(X_transformed, y)
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=5,
       n_iter_no_change=5, n_jobs=None, penalty='l2', power_t=0.5,
       random_state=0, shuffle=True, tol=0.001, validation_fraction=0.1,
       verbose=0, warm_start=False)
>>> clf.score(X_transformed, y)
0.9499...
```

Methods

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

__init__ (*self*, *sample_steps=2*, *sample_interval=None*)

fit (*self*, *X*, *y=None*)
Set the parameters

Parameters

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

Returns

self [object] Returns the transformer.

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Apply approximate feature map to X.

Parameters

X [{array-like, sparse matrix}, shape = (n_samples, n_features)]

Returns

X_new [{array, sparse matrix}, shape = (n_samples, n_features * (2*sample_steps + 1))] Whether the return value is an array of sparse matrix depends on the type of the input X.

6.20.2 sklearn.kernel_approximation.Nystroem

class sklearn.kernel_approximation.**Nystroem**(*kernel='rbf'*, *gamma=None*, *coef0=None*, *degree=None*, *kernel_params=None*, *n_components=100*, *random_state=None*)
Approximate a kernel map using a subset of the training data.

Constructs an approximate feature map for an arbitrary kernel using a subset of the data as basis.

Read more in the *User Guide*.

Parameters

kernel [string or callable, default="rbf"] Kernel map to be approximated. A callable should accept two arguments and the keyword arguments passed to this object as kernel_params, and should return a floating point number.

gamma [float, default=None] Gamma parameter for the RBF, laplacian, polynomial, exponential chi2 and sigmoid kernels. Interpretation of the default value is left to the kernel; see the documentation for sklearn.metrics.pairwise. Ignored by other kernels.

coef0 [float, default=None] Zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels.

degree [float, default=None] Degree of the polynomial kernel. Ignored by other kernels.

kernel_params [mapping of string to any, optional] Additional parameters (keyword arguments) for kernel function passed as callable object.

n_components [int] Number of features to construct. How many data points will be used to construct the mapping.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Attributes

components_ [array, shape (n_components, n_features)] Subset of training points used to construct the feature map.

component_indices_ [array, shape (n_components)] Indices of `components_` in the training set.

normalization_ [array, shape (n_components, n_components)] Normalization matrix needed for embedding. Square root of the kernel matrix on `components_`.

See also:

RBFSampler An approximation to the RBF kernel using random Fourier features.

sklearn.metrics.pairwise.kernel_metrics List of built-in kernels.

References

- Williams, C.K.I. and Seeger, M. "Using the Nystroem method to speed up kernel machines", Advances in neural information processing systems 2001
- T. Yang, Y. Li, M. Mahdavi, R. Jin and Z. Zhou "Nystroem Method vs Random Fourier Features: A Theoretical and Empirical Comparison", Advances in Neural Information Processing Systems 2012

Examples

```
>>> from sklearn import datasets, svm
>>> from sklearn.kernel_approximation import Nystroem
>>> digits = datasets.load_digits(n_class=9)
>>> data = digits.data / 16.
>>> clf = svm.LinearSVC()
>>> feature_map_nystroem = Nystroem(gamma=.2,
...                                 random_state=1,
...                                 n_components=300)
>>> data_transformed = feature_map_nystroem.fit_transform(data)
>>> clf.fit(data_transformed, digits.target)
...
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
>>> clf.score(data_transformed, digits.target)
0.9987...
```

Methods

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

__init__(*self*, *kernel='rbf'*, *gamma=None*, *coef0=None*, *degree=None*, *kernel_params=None*, *n_components=100*, *random_state=None*)

fit(*self*, *X*, *y=None*)
Fit estimator to data.

Samples a subset of training points, computes kernel on these and computes normalization matrix.

Parameters

X [array-like, shape=(n_samples, n_feature)] Training data.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Apply feature map to X.

Computes an approximate feature map using the kernel between some training points and X.

Parameters

X [array-like, shape=(n_samples, n_features)] Data to transform.

Returns

X_transformed [array, shape=(n_samples, n_components)] Transformed data.

Examples using `sklearn.kernel_approximation.Nystroem`

- *Explicit feature map approximation for RBF kernels*

6.20.3 sklearn.kernel_approximation.RBFSampler

class sklearn.kernel_approximation.**RBFSampler**(*gamma=1.0*, *n_components=100*, *random_state=None*)

Approximates feature map of an RBF kernel by Monte Carlo approximation of its Fourier transform.

It implements a variant of Random Kitchen Sinks.[1]

Read more in the *User Guide*.

Parameters

gamma [float] Parameter of RBF kernel: exp(-gamma * x^2)

n_components [int] Number of Monte Carlo samples per original feature. Equals the dimensionality of the computed feature space.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Notes

See "Random Features for Large-Scale Kernel Machines" by A. Rahimi and Benjamin Recht.

[1] "Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning" by A. Rahimi and Benjamin Recht. (https://people.eecs.berkeley.edu/~brecht/papers/08.rah.rec.nips.pdf)

Examples

```
>>> from sklearn.kernel_approximation import RBFSampler
>>> from sklearn.linear_model import SGDClassifier
>>> X = [[0, 0], [1, 1], [1, 0], [0, 1]]
>>> y = [0, 0, 1, 1]
>>> rbf_feature = RBFSampler(gamma=1, random_state=1)
>>> X_features = rbf_feature.fit_transform(X)
>>> clf = SGDClassifier(max_iter=5, tol=1e-3)
>>> clf.fit(X_features, y)
...
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=5,
       n_iter_no_change=5, n_jobs=None, penalty='l2', power_t=0.5,
       random_state=None, shuffle=True, tol=0.001, validation_fraction=0.1,
       verbose=0, warm_start=False)
>>> clf.score(X_features, y)
1.0
```

Methods

fit(self, X[, y])	Fit the model with X.
fit_transform(self, X[, y])	Fit to data, then transform it.

Continued on next page

Table 6.144 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply the approximate feature map to X.

__init__ (*self*, *gamma=1.0*, *n_components=100*, *random_state=None*)

fit (*self*, *X*, *y=None*)
Fit the model with X.

Samples random projection according to n_features.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data, where n_samples in the number of samples and n_features is the number of features.

Returns

self [object] Returns the transformer.

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, ***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.

Returns

self

transform (*self*, *X*)
Apply the approximate feature map to X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] New data, where n_samples in the number of samples and n_features is the number of features.

Returns

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

Examples using `sklearn.kernel_approximation.RBFSampler`

- *Explicit feature map approximation for RBF kernels*

6.20.4 `sklearn.kernel_approximation`.SkewedChi2Sampler

class `sklearn.kernel_approximation.`**`SkewedChi2Sampler`**(*skewedness=1.0*, *n_components=100*, *random_state=None*)

Approximates feature map of the "skewed chi-squared" kernel by Monte Carlo approximation of its Fourier transform.

Read more in the *User Guide*.

Parameters

skewedness [float] "skewedness" parameter of the kernel. Needs to be cross-validated.

n_components [int] number of Monte Carlo samples per original feature. Equals the dimensionality of the computed feature space.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

See also:

AdditiveChi2Sampler A different approach for approximating an additive variant of the chi squared kernel.

sklearn.metrics.pairwise.chi2_kernel The exact chi squared kernel.

References

See "Random Fourier Approximations for Skewed Multiplicative Histogram Kernels" by Fuxin Li, Catalin Ionescu and Cristian Sminchisescu.

Examples

```
>>> from sklearn.kernel_approximation import SkewedChi2Sampler
>>> from sklearn.linear_model import SGDClassifier
>>> X = [[0, 0], [1, 1], [1, 0], [0, 1]]
>>> y = [0, 0, 1, 1]
>>> chi2_feature = SkewedChi2Sampler(skewedness=.01,
...                                  n_components=10,
...                                  random_state=0)
>>> X_features = chi2_feature.fit_transform(X, y)
>>> clf = SGDClassifier(max_iter=10, tol=1e-3)
>>> clf.fit(X_features, y)
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
```

```
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=10,
       n_iter_no_change=5, n_jobs=None, penalty='l2', power_t=0.5,
       random_state=None, shuffle=True, tol=0.001, validation_fraction=0.1,
       verbose=0, warm_start=False)
>>> clf.score(X_features, y)
1.0
```

Methods

fit(self, X[, y])	Fit the model with X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply the approximate feature map to X.

__init__ (*self*, *skewedness=1.0*, *n_components=100*, *random_state=None*)

fit (*self*, *X*, *y=None*)
Fit the model with X.

Samples random projection according to n_features.

Parameters

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

Returns

self [object] Returns the transformer.

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Apply the approximate feature map to X.

Parameters

X [array-like, shape (n_samples, n_features)] New data, where n_samples in the number of samples and n_features is the number of features. All values of X must be strictly greater than "-skewedness".

Returns

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

6.21 sklearn.kernel_ridge Kernel Ridge Regression

Module *sklearn.kernel_ridge* implements kernel ridge regression.

User guide: See the *Kernel ridge regression* section for further details.

kernel_ridge.KernelRidge([alpha, kernel, ...])	Kernel ridge regression.

6.21.1 sklearn.kernel_ridge.KernelRidge

class sklearn.kernel_ridge.**KernelRidge**(*alpha=1*, *kernel='linear'*, *gamma=None*, *degree=3*, *coef0=1*, *kernel_params=None*)
Kernel ridge regression.

Kernel ridge regression (KRR) combines ridge regression (linear least squares with l2-norm regularization) with the kernel trick. It thus learns a linear function in the space induced by the respective kernel and the data. For non-linear kernels, this corresponds to a non-linear function in the original space.

The form of the model learned by KRR is identical to support vector regression (SVR). However, different loss functions are used: KRR uses squared error loss while support vector regression uses epsilon-insensitive loss, both combined with l2 regularization. In contrast to SVR, fitting a KRR model can be done in closed-form and is typically faster for medium-sized datasets. On the other hand, the learned model is non-sparse and thus slower than SVR, which learns a sparse model for epsilon > 0, at prediction-time.

This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]).

Read more in the *User Guide*.

Parameters

alpha [{float, array-like}, shape = [n_targets]] Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to `(2*C)^-1` in other linear models such as LogisticRegression or LinearSVC. If an array is

passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

kernel [string or callable, default="linear"] Kernel mapping used internally. A callable should accept two arguments and the keyword arguments passed to this object as kernel_params, and should return a floating point number. Set to "precomputed" in order to pass a precomputed kernel matrix to the estimator methods instead of samples.

gamma [float, default=None] Gamma parameter for the RBF, laplacian, polynomial, exponential chi2 and sigmoid kernels. Interpretation of the default value is left to the kernel; see the documentation for sklearn.metrics.pairwise. Ignored by other kernels.

degree [float, default=3] Degree of the polynomial kernel. Ignored by other kernels.

coef0 [float, default=1] Zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels.

kernel_params [mapping of string to any, optional] Additional parameters (keyword arguments) for kernel function passed as callable object.

Attributes

dual_coef_ [array, shape = [n_samples] or [n_samples, n_targets]] Representation of weight vector(s) in kernel space

X_fit_ [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training data, which is also required for prediction. If kernel == "precomputed" this is instead the precomputed training matrix, shape = [n_samples, n_samples].

See also:

sklearn.linear_model.Ridge Linear ridge regression.

sklearn.svm.SVR Support Vector Regression implemented using libsvm.

References

- Kevin P. Murphy "Machine Learning: A Probabilistic Perspective", The MIT Press chapter 14.4.3, pp. 492-493

Examples

```
>>> from sklearn.kernel_ridge import KernelRidge
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> clf = KernelRidge(alpha=1.0)
>>> clf.fit(X, y)
KernelRidge(alpha=1.0, coef0=1, degree=3, gamma=None, kernel='linear',
            kernel_params=None)
```

Methods

fit(self, X[, y, sample_weight])	Fit Kernel Ridge regression model
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the kernel ridge model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *alpha=1*, *kernel='linear'*, *gamma=None*, *degree=3*, *coef0=1*, *kernel_params=None*)

fit (*self*, *X*, *y=None*, *sample_weight=None*)
Fit Kernel Ridge regression model

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training data. If kernel == "precomputed" this is instead a precomputed kernel matrix, shape = [n_samples, n_samples].

y [array-like, shape = [n_samples] or [n_samples, n_targets]] Target values

sample_weight [float or array-like of shape [n_samples]] Individual weights for each sample, ignored if None is passed.

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict using the kernel ridge model

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Samples. If kernel == "precomputed" this is instead a precomputed kernel matrix, shape = [n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for this estimator.

Returns

C [array, shape = [n_samples] or [n_samples, n_targets]] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.kernel_ridge.KernelRidge`

- *Comparison of kernel ridge regression and SVR*
- *Comparison of kernel ridge and Gaussian process regression*

6.22 `sklearn.linear_model`: Generalized Linear Models

The *sklearn.linear_model* module implements generalized linear models. It includes Ridge regression, Bayesian Regression, Lasso and Elastic Net estimators computed with Least Angle Regression and coordinate descent. It also implements Stochastic Gradient Descent related algorithms.

User guide: See the *Generalized Linear Models* section for further details.

linear_model.ARDRegression([n_iter, tol, ...])	Bayesian ARD regression.
linear_model.BayesianRidge([n_iter, tol, ...])	Bayesian ridge regression.
linear_model.ElasticNet([alpha, l1_ratio, ...])	Linear regression with combined L1 and L2 priors as regularizer.
linear_model.ElasticNetCV([l1_ratio, eps, ...])	Elastic Net model with iterative fitting along a regularization path.
linear_model.HuberRegressor([epsilon, ...])	Linear regression model that is robust to outliers.
linear_model.Lars([fit_intercept, verbose, ...])	Least Angle Regression model a.k.a.

Continued on next page

Table 6.148 – continued from previous page

linear_model.LarsCV([fit_intercept, ...])	Cross-validated Least Angle Regression model.
linear_model.Lasso([alpha, fit_intercept, ...])	Linear Model trained with L1 prior as regularizer (aka the Lasso)
linear_model.LassoCV([eps, n_alphas, ...])	Lasso linear model with iterative fitting along a regularization path.
linear_model.LassoLars([alpha, ...])	Lasso model fit with Least Angle Regression a.k.a.
linear_model.LassoLarsCV([fit_intercept, ...])	Cross-validated Lasso, using the LARS algorithm.
linear_model.LassoLarsIC([criterion, ...])	Lasso model fit with Lars using BIC or AIC for model selection
linear_model.LinearRegression([...])	Ordinary least squares Linear Regression.
linear_model.LogisticRegression([penalty, ...])	Logistic Regression (aka logit, MaxEnt) classifier.
linear_model.LogisticRegressionCV([Cs, ...])	Logistic Regression CV (aka logit, MaxEnt) classifier.
linear_model.MultiTaskLasso([alpha, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.
linear_model.MultiTaskElasticNet([alpha, ...])	Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer
linear_model.MultiTaskLassoCV([eps, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.
linear_model.MultiTaskElasticNetCV([...])	Multi-task L1/L2 ElasticNet with built-in cross-validation.
linear_model.OrthogonalMatchingPursuit([...])	Orthogonal Matching Pursuit model (OMP)
linear_model.OrthogonalMatchingPursuitCV([...])	Cross-validated Orthogonal Matching Pursuit model (OMP).
linear_model.PassiveAggressiveClassifier([...])	Passive Aggressive Classifier
linear_model.PassiveAggressiveRegressor([C, ...])	Passive Aggressive Regressor
linear_model.Perceptron([penalty, alpha, ...])	Read more in the *User Guide*.
linear_model.RANSACRegressor([...])	RANSAC (RANdom SAmple Consensus) algorithm.
linear_model.Ridge([alpha, fit_intercept, ...])	Linear least squares with l2 regularization.
linear_model.RidgeClassifier([alpha, ...])	Classifier using Ridge regression.
linear_model.RidgeClassifierCV([alphas, ...])	Ridge classifier with built-in cross-validation.
linear_model.RidgeCV([alphas, ...])	Ridge regression with built-in cross-validation.
linear_model.SGDClassifier([loss, penalty, ...])	Linear classifiers (SVM, logistic regression, a.o.) with SGD training.
linear_model.SGDRegressor([loss, penalty, ...])	Linear model fitted by minimizing a regularized empirical loss with SGD
linear_model.TheilSenRegressor([...])	Theil-Sen Estimator: robust multivariate regression model.

6.22.1 sklearn.linear_model.ARDRegression

class sklearn.linear_model.**ARDRegression**(*n_iter=300*, *tol=0.001*, *alpha_1=1e-06*, *alpha_2=1e-06*, *lambda_1=1e-06*, *lambda_2=1e-06*, *compute_score=False*, *threshold_lambda=10000.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *verbose=False*)

Bayesian ARD regression.

Fit the weights of a regression model, using an ARD prior. The weights of the regression model are assumed to be in Gaussian distributions. Also estimate the parameters lambda (precisions of the distributions of the weights) and alpha (precision of the distribution of the noise). The estimation is done by an iterative procedures

(Evidence Maximization)

Read more in the *User Guide*.

Parameters

n_iter [int, optional] Maximum number of iterations. Default is 300

tol [float, optional] Stop the algorithm if w has converged. Default is 1.e-3.

alpha_1 [float, optional] Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

alpha_2 [float, optional] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

lambda_1 [float, optional] Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

lambda_2 [float, optional] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

compute_score [boolean, optional] If True, compute the objective function at each step of the model. Default is False.

threshold_lambda [float, optional] threshold for removing (pruning) weights with high precision from the computation. Default is 1.e+4.

fit_intercept [boolean, optional] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered). Default is True.

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True.] If True, X will be copied; else, it may be overwritten.

verbose [boolean, optional, default False] Verbose mode when fitting the model.

Attributes

coef_ [array, shape = (n_features)] Coefficients of the regression model (mean of distribution)

alpha_ [float] estimated precision of the noise.

lambda_ [array, shape = (n_features)] estimated precisions of the weights.

sigma_ [array, shape = (n_features, n_features)] estimated variance-covariance matrix of the weights

scores_ [float] if computed, value of the objective function (to be maximized)

Notes

For an example, see *examples/linear_model/plot_ard.py*.

References

D. J. C. MacKay, Bayesian nonlinear modeling for the prediction competition, ASHRAE Transactions, 1994.

R. Salakhutdinov, Lecture notes on Statistical Machine Learning, http://www.utstat.toronto.edu/~rsalakhu/sta4273/notes/Lecture2.pdf#page=15 Their beta is our `self.alpha_` Their alpha is our `self.lambda_` ARD is a little different than the slide: only dimensions/features for which `self.lambda_ < self.threshold_lambda` are kept and the rest are discarded.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.ARDRegression()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
...
ARDRegression(alpha_1=1e-06, alpha_2=1e-06, compute_score=False,
        copy_X=True, fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06,
        n_iter=300, normalize=False, threshold_lambda=10000.0, tol=0.001,
        verbose=False)
>>> clf.predict([[1, 1]])
array([1.])
```

Methods

fit(self, X, y)	Fit the ARDRegression model according to the given training data and parameters.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, return_std])	Predict using the linear model.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *n_iter=300*, *tol=0.001*, *alpha_1=1e-06*, *alpha_2=1e-06*, *lambda_1=1e-06*, *lambda_2=1e-06*, *compute_score=False*, *threshold_lambda=10000.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *verbose=False*)

fit (*self*, *X*, *y*)

Fit the ARDRegression model according to the given training data and parameters.

Iterative procedure to maximize the evidence

Parameters

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

y [array, shape = [n_samples]] Target values (integers). Will be cast to X's dtype if necessary

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*, *return_std=False*)
Predict using the linear model.

In addition to the mean of the predictive distribution, also its standard deviation can be returned.

Parameters

X [{array-like, sparse matrix}, shape = (n_samples, n_features)] Samples.

return_std [boolean, optional] Whether to return the standard deviation of posterior prediction.

Returns

y_mean [array, shape = (n_samples,)] Mean of predictive distribution of query points.

y_std [array, shape = (n_samples,)] Standard deviation of predictive distribution of query points.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.linear_model.ARDRegression`

- *Automatic Relevance Determination Regression (ARD)*

6.22.2 `sklearn.linear_model`.BayesianRidge

class `sklearn.linear_model.`**`BayesianRidge`**(*n_iter=300*, *tol=0.001*, *alpha_1=1e-06*, *alpha_2=1e-06*, *lambda_1=1e-06*, *lambda_2=1e-06*, *compute_score=False*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *verbose=False*)

Bayesian ridge regression.

Fit a Bayesian ridge model. See the Notes section for details on this implementation and the optimization of the regularization parameters lambda (precision of the weights) and alpha (precision of the noise).

Read more in the *User Guide*.

Parameters

n_iter [int, optional] Maximum number of iterations. Default is 300. Should be greater than or equal to 1.

tol [float, optional] Stop the algorithm if w has converged. Default is 1.e-3.

alpha_1 [float, optional] Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter. Default is 1.e-6

alpha_2 [float, optional] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter. Default is 1.e-6.

lambda_1 [float, optional] Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter. Default is 1.e-6.

lambda_2 [float, optional] Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter. Default is 1.e-6

compute_score [boolean, optional] If True, compute the log marginal likelihood at each iteration of the optimization. Default is False.

fit_intercept [boolean, optional, default True] Whether to calculate the intercept for this model. The intercept is not treated as a probabilistic parameter and thus has no associated variance. If set to False, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

verbose [boolean, optional, default False] Verbose mode when fitting the model.

Attributes

coef_ [array, shape = (n_features,)] Coefficients of the regression model (mean of distribution).

intercept_ [float] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

alpha_ [float] Estimated precision of the noise.

lambda_ [float] Estimated precision of the weights.

sigma_ [array, shape = (n_features, n_features)] Estimated variance-covariance matrix of the weights.

scores_ [array, shape = (**n_iter_** + 1,)] If computed_score is True, value of the log marginal likelihood (to be maximized) at each iteration of the optimization. The array starts with the value of the log marginal likelihood obtained for the initial values of alpha and lambda and ends with the value obtained for the estimated alpha and lambda.

n_iter_ [int] The actual number of iterations to reach the stopping criterion.

Notes

There exist several strategies to perform Bayesian ridge regression. This implementation is based on the algorithm described in Appendix A of (Tipping, 2001) where updates of the regularization parameters are done as suggested in (MacKay, 1992). Note that according to A New View of Automatic Relevance Determination (Wipf and Nagarajan, 2008) these update rules do not guarantee that the marginal likelihood is increasing between two consecutive iterations of the optimization.

References

D. J. C. MacKay, Bayesian Interpolation, Computation and Neural Systems, Vol. 4, No. 3, 1992.

M. E. Tipping, Sparse Bayesian Learning and the Relevance Vector Machine, Journal of Machine Learning Research, Vol. 1, 2001.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.BayesianRidge()
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
...
BayesianRidge(alpha_1=1e-06, alpha_2=1e-06, compute_score=False,
        copy_X=True, fit_intercept=True, lambda_1=1e-06, lambda_2=1e-06,
        n_iter=300, normalize=False, tol=0.001, verbose=False)
>>> clf.predict([[1, 1]])
array([1.])
```

Methods

fit(self, X, y[, sample_weight])	Fit the model
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, return_std])	Predict using the linear model.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.

Continued on next page

Table 6.150 – continued from previous page

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

__init__ (*self*, *n_iter=300*, *tol=0.001*, *alpha_1=1e-06*, *alpha_2=1e-06*, *lambda_1=1e-06*, *lambda_2=1e-06*, *compute_score=False*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *verbose=False*)

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit the model

Parameters

X [numpy array of shape [n_samples,n_features]] Training data

y [numpy array of shape [n_samples]] Target values. Will be cast to X's dtype if necessary

sample_weight [numpy array of shape [n_samples]] Individual weights for each sample

New in version 0.20: parameter *sample_weight* support to BayesianRidge.

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*, *return_std=False*)
Predict using the linear model.

In addition to the mean of the predictive distribution, also its standard deviation can be returned.

Parameters

X [{array-like, sparse matrix}, shape = (n_samples, n_features)] Samples.

return_std [boolean, optional] Whether to return the standard deviation of posterior prediction.

Returns

y_mean [array, shape = (n_samples,)] Mean of predictive distribution of query points.

y_std [array, shape = (n_samples,)] Standard deviation of predictive distribution of query points.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.linear_model.BayesianRidge`

- *Feature agglomeration vs. univariate selection*
- *Imputing missing values with variants of IterativeImputer*
- *Bayesian Ridge Regression*

6.22.3 `sklearn.linear_model`.ElasticNet

class `sklearn.linear_model.`**ElasticNet**(*alpha=1.0*, *l1_ratio=0.5*, *fit_intercept=True*, *normalize=False*, *precompute=False*, *max_iter=1000*, *copy_X=True*, *tol=0.0001*, *warm_start=False*, *positive=False*, *random_state=None*, *selection='cyclic'*)
Linear regression with combined L1 and L2 priors as regularizer.

Minimizes the objective function:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

If you are interested in controlling the L1 and L2 penalty separately, keep in mind that this is equivalent to:

```
a * L1 + b * L2
```

where:

```
alpha = a + b and l1_ratio = a / (a + b)
```

The parameter l1_ratio corresponds to alpha in the glmnet R package while alpha corresponds to the lambda parameter in glmnet. Specifically, l1_ratio = 1 is the lasso penalty. Currently, l1_ratio <= 0.01 is not reliable, unless you supply your own sequence of alpha.

Read more in the *User Guide*.

Parameters

alpha [float, optional] Constant that multiplies the penalty terms. Defaults to 1.0. See the notes for the exact mathematical meaning of this parameter.‘‘alpha = 0‘‘ is equivalent to an ordinary least square, solved by the *LinearRegression* object. For numerical reasons, using `alpha = 0` with the `Lasso` object is not advised. Given this, you should use the *LinearRegression* object.

l1_ratio [float] The ElasticNet mixing parameter, with `0 <= l1_ratio <= 1`. For `l1_ratio = 0` the penalty is an L2 penalty. `For l1_ratio = 1` it is an L1 penalty. For `0 < l1_ratio < 1`, the penalty is a combination of L1 and L2.

fit_intercept [bool] Whether the intercept should be estimated or not. If `False`, the data is assumed to be already centered.

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *sklearn.preprocessing.StandardScaler* before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | array-like] Whether to use a precomputed Gram matrix to speed up calculations. The Gram matrix can also be passed as argument. For sparse input this option is always `True` to preserve sparsity.

max_iter [int, optional] The maximum number of iterations

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

warm_start [bool, optional] When set to `True`, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

positive [bool, optional] When set to `True`, forces the coefficients to be positive.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == ‘random’.

selection [str, default ‘cyclic’] If set to ‘random’, a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to ‘random’) often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

coef_ [array, shape (n_features,) | (n_targets, n_features)] parameter vector (w in the cost function formula)

sparse_coef_ [scipy.sparse matrix, shape (n_features, 1) | (n_targets, n_features)] sparse representation of the fitted `coef_`

intercept_ [float | array, shape (n_targets,)] independent term in decision function.

n_iter_ [array-like, shape (n_targets,)] number of iterations run by the coordinate descent solver to reach the specified tolerance.

See also:

ElasticNetCV Elastic net model with best model selection by cross-validation.

SGDRegressor implements elastic net regression with incremental training.

SGDClassifier implements logistic regression with elastic net penalty (`SGDClassifier(loss="log", penalty="elasticnet")`).

Notes

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn.linear_model import ElasticNet
>>> from sklearn.datasets import make_regression
```

```
>>> X, y = make_regression(n_features=2, random_state=0)
>>> regr = ElasticNet(random_state=0)
>>> regr.fit(X, y)
ElasticNet(alpha=1.0, copy_X=True, fit_intercept=True, l1_ratio=0.5,
      max_iter=1000, normalize=False, positive=False, precompute=False,
      random_state=0, selection='cyclic', tol=0.0001, warm_start=False)
>>> print(regr.coef_)
[18.83816048 64.55968825]
>>> print(regr.intercept_)
1.451...
>>> print(regr.predict([[0, 0]]))
[1.451...]
```

Methods

fit(self, X, y[, check_input])	Fit model with coordinate descent.
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *l1_ratio=0.5*, *fit_intercept=True*, *normalize=False*, *precompute=False*, *max_iter=1000*, *copy_X=True*, *tol=0.0001*, *warm_start=False*, *positive=False*, *random_state=None*, *selection='cyclic'*)

fit(*self*, *X*, *y*, *check_input=True*)

Fit model with coordinate descent.

Parameters

X [ndarray or scipy.sparse matrix, (n_samples, n_features)] Data

y [ndarray, shape (n_samples,) or (n_samples, n_targets)] Target. Will be cast to X's dtype if necessary

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a Fortran-contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

static **path**(*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)

Compute elastic net path with coordinate descent

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values

l1_ratio [float, optional] float between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

predict (*self*, *X*)
Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

sparse_coef_
sparse representation of the fitted `coef_`

Examples using `sklearn.linear_model.ElasticNet`

- *Lasso and Elastic Net for Sparse Signals*
- *Train error vs Test error*

6.22.4 `sklearn.linear_model`.HuberRegressor

class `sklearn.linear_model.`**`HuberRegressor`**(*epsilon=1.35*, *max_iter=100*, *alpha=0.0001*, *warm_start=False*, *fit_intercept=True*, *tol=1e-05*)

Linear regression model that is robust to outliers.

The Huber Regressor optimizes the squared loss for the samples where `|(y - X'w) / sigma| < epsilon` and the absolute loss for the samples where `|(y - X'w) / sigma| > epsilon`, where w and sigma are parameters to be optimized. The parameter sigma makes sure that if y is scaled up or down by a certain factor, one does not need to rescale epsilon to achieve the same robustness. Note that this does not take into account the fact that the different features of X may be of different scales.

This makes sure that the loss function is not heavily influenced by the outliers while not completely ignoring their effect.

Read more in the *User Guide*

New in version 0.18.

Parameters

epsilon [float, greater than 1.0, default 1.35] The parameter epsilon controls the number of samples that should be classified as outliers. The smaller the epsilon, the more robust it is to outliers.

max_iter [int, default 100] Maximum number of iterations that scipy.optimize.fmin_l_bfgs_b should run for.

alpha [float, default 0.0001] Regularization parameter.

warm_start [bool, default False] This is useful if the stored attributes of a previously used model has to be reused. If set to False, then the coefficients will be rewritten for every call to fit. See *the Glossary*.

fit_intercept [bool, default True] Whether or not to fit the intercept. This can be set to False if the data is already centered around the origin.

tol [float, default 1e-5] The iteration will stop when `max{|proj g_i | i = 1, ..., n} <= tol` where pg_i is the i-th component of the projected gradient.

Attributes

coef_ [array, shape (n_features,)] Features got by optimizing the Huber loss.

intercept_ [float] Bias.

scale_ [float] The value by which `|y - X'w - c|` is scaled down.

n_iter_ [int] Number of iterations that fmin_l_bfgs_b has run for.

Changed in version 0.20: In SciPy <= 1.0.0 the number of lbfgs iterations may exceed `max_iter`. `n_iter_` will now report at most `max_iter`.

outliers_ [array, shape (n_samples,)] A boolean mask which is set to True where the samples are identified as outliers.

References

[Re4616ef910fb-1], *[Re4616ef910fb-2]*

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import HuberRegressor, LinearRegression
>>> from sklearn.datasets import make_regression
>>> rng = np.random.RandomState(0)
>>> X, y, coef = make_regression(
...     n_samples=200, n_features=2, noise=4.0, coef=True, random_state=0)
>>> X[:4] = rng.uniform(10, 20, (4, 2))
>>> y[:4] = rng.uniform(10, 20, 4)
>>> huber = HuberRegressor().fit(X, y)
>>> huber.score(X, y)
-7.284608623514573
>>> huber.predict(X[:1,])
array([806.7200...])
>>> linear = LinearRegression().fit(X, y)
>>> print("True coefficients:", coef)
True coefficients: [20.4923...  34.1698...]
>>> print("Huber coefficients:", huber.coef_)
Huber coefficients: [17.7906... 31.0106...]
>>> print("Linear Regression coefficients:", linear.coef_)
Linear Regression coefficients: [-1.9221...  7.0226...]
```

Methods

fit(self, X, y[, sample_weight])	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *epsilon=1.35*, *max_iter=100*, *alpha=0.0001*, *warm_start=False*, *fit_intercept=True*, *tol=1e-05*)

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

Fit the model according to the given training data.

Parameters

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

y [array-like, shape (n_samples,)] Target vector relative to X.

sample_weight [array-like, shape (n_samples,)] Weight given to each sample.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)
Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.linear_model.HuberRegressor`

- *HuberRegressor vs Ridge on dataset with strong outliers*
- *Robust linear estimator fitting*

6.22.5 `sklearn.linear_model`.Lars

class `sklearn.linear_model.`**Lars**(*fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *n_nonzero_coefs=500*, *eps=2.220446049250313e-16*, *copy_X=True*, *fit_path=True*, *positive=False*)

Least Angle Regression model a.k.a. LAR

Read more in the *User Guide*.

Parameters

fit_intercept [boolean] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

verbose [boolean or integer, optional] Sets the verbosity amount

normalize [boolean, optional, default True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

n_nonzero_coefs [int, optional] Target number of non-zero coefficients. Use `np.inf` for no limit.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the `tol` parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

fit_path [boolean] If True the full path is stored in the `coef_path_` attribute. If you compute the solution for a large problem or many targets, setting `fit_path` to `False` will lead to a speedup, especially with a small alpha.

positive [boolean (default=False)] Restrict coefficients to be >= 0. Be aware that you might want to remove fit_intercept which is set True by default.

Deprecated since version 0.20: The option is broken and deprecated. It will be removed in v0.22.

Attributes

alphas_ [array, shape (n_alphas + 1,) | list of n_targets such arrays] Maximum of covariances (in absolute value) at each iteration. `n_alphas` is either `n_nonzero_coefs` or `n_features`, whichever is smaller.

active_ [list, length = n_alphas | list of n_targets such lists] Indices of active variables at the end of the path.

coef_path_ [array, shape (n_features, n_alphas + 1) | list of n_targets such arrays] The varying values of the coefficients along the path. It is not present if the `fit_path` parameter is `False`.

coef_ [array, shape (n_features,) or (n_targets, n_features)] Parameter vector (w in the formulation formula).

intercept_ [float | array, shape (n_targets,)] Independent term in decision function.

n_iter_ [array-like or int] The number of iterations taken by lars_path to find the grid of alphas for each target.

See also:

lars_path, *LarsCV*

sklearn.decomposition.sparse_encode

Examples

```
>>> from sklearn import linear_model
>>> reg = linear_model.Lars(n_nonzero_coefs=1)
>>> reg.fit([[-1, 1], [0, 0], [1, 1]], [-1.1111, 0, -1.1111])
...
Lars(copy_X=True, eps=..., fit_intercept=True, fit_path=True,
   n_nonzero_coefs=1, normalize=True, positive=False, precompute='auto',
   verbose=False)
>>> print(reg.coef_)
[ 0. -1.11...]
```

Methods

fit(self, X, y[, Xy])	Fit the model using X, y as training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *n_nonzero_coefs=500*, *eps=2.220446049250313e-16*, *copy_X=True*, *fit_path=True*, *positive=False*)

fit(*self*, *X*, *y*, *Xy=None*)
Fit the model using X, y as training data.

Parameters

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

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values.

Xy [array-like, shape (n_samples,) or (n_samples, n_targets), optional] Xy = np.dot(X.T, y)

that can be precomputed. It is useful only when the Gram matrix is precomputed.

Returns

self [object] returns an instance of self.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

6.22.6 sklearn.linear_model.Lasso

class sklearn.linear_model.**Lasso**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *precompute=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Linear Model trained with L1 prior as regularizer (aka the Lasso)

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Technically the Lasso model is optimizing the same objective function as the Elastic Net with `l1_ratio=1.0` (no L2 penalty).

Read more in the *User Guide*.

Parameters

alpha [float, optional] Constant that multiplies the L1 term. Defaults to 1.0. `alpha = 0` is equivalent to an ordinary least square, solved by the *LinearRegression* object. For numerical reasons, using `alpha = 0` with the `Lasso` object is not advised. Given this, you should use the *LinearRegression* object.

fit_intercept [boolean, optional, default True] Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *sklearn.preprocessing.StandardScaler* before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | array-like, default=False] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument. For sparse input this option is always `True` to preserve sparsity.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

positive [bool, optional] When set to `True`, forces the coefficients to be positive.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

coef_ [array, shape (n_features,) | (n_targets, n_features)] parameter vector (w in the cost function formula)

sparse_coef_ [scipy.sparse matrix, shape (n_features, 1) | (n_targets, n_features)] sparse representation of the fitted `coef_`

intercept_ [float | array, shape (n_targets,)] independent term in decision function.

n_iter_ [int | array-like, shape (n_targets,)] number of iterations run by the coordinate descent solver to reach the specified tolerance.

See also:

lars_path

lasso_path

LassoLars

LassoCV

LassoLarsCV

sklearn.decomposition.sparse_encode

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.Lasso(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
...
Lasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
>>> print(clf.coef_)
[0.85 0.  ]
>>> print(clf.intercept_)
0.15...
```

Methods

fit(self, X, y[, check_input])	Fit model with coordinate descent.
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *alpha=1.0*, *fit_intercept=True*, *normalize=False*, *precompute=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *positive=False*, *random_state=None*, *selection='cyclic'*)

fit (*self*, *X*, *y*, *check_input=True*)
Fit model with coordinate descent.

Parameters

X [ndarray or scipy.sparse matrix, (n_samples, n_features)] Data

y [ndarray, shape (n_samples,) or (n_samples, n_targets)] Target. Will be cast to X's dtype if necessary

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a Fortran-contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

static path (*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)
Compute elastic net path with coordinate descent

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values

l1_ratio [float, optional] float between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

predict (*self*, *X*)

Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score*

directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer 'r2' uses multioutput='uniform_average').

set_params(*self*, ***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.

Returns

self

sparse_coef_
sparse representation of the fitted coef_

Examples using sklearn.linear_model.Lasso

- *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*
- *Cross-validation on diabetes Dataset Exercise*
- *Lasso on dense and sparse data*
- *Joint feature selection with multi-task Lasso*
- *Lasso and Elastic Net for Sparse Signals*

6.22.7 sklearn.linear_model.LassoLars

class sklearn.linear_model.**LassoLars**(*alpha=1.0*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *max_iter=500*, *eps=2.220446049250313e-16*, *copy_X=True*, *fit_path=True*, *positive=False*)
Lasso model fit with Least Angle Regression a.k.a. Lars

It is a Linear Model trained with an L1 prior as regularizer.

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Read more in the *User Guide*.

Parameters

alpha [float] Constant that multiplies the penalty term. Defaults to 1.0. alpha = 0 is equivalent to an ordinary least square, solved by *LinearRegression*. For numerical reasons, using alpha = 0 with the LassoLars object is not advised and you should prefer the LinearRegression object.

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

verbose [boolean or integer, optional] Sets the verbosity amount

normalize [boolean, optional, default True] This parameter is ignored when fit_intercept is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use

sklearn.preprocessing.StandardScaler before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

max_iter [integer, optional] Maximum number of iterations to perform.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the `tol` parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

fit_path [boolean] If `True` the full path is stored in the `coef_path_` attribute. If you compute the solution for a large problem or many targets, setting `fit_path` to `False` will lead to a speedup, especially with a small alpha.

positive [boolean (default=False)] Restrict coefficients to be >= 0. Be aware that you might want to remove fit_intercept which is set True by default. Under the positive restriction the model coefficients will not converge to the ordinary-least-squares solution for small values of alpha. Only coefficients up to the smallest alpha value (`alphas_[alphas_ > 0.].min()` when fit_path=True) reached by the stepwise Lars-Lasso algorithm are typically in congruence with the solution of the coordinate descent Lasso estimator.

Attributes

alphas_ [array, shape (n_alphas + 1,) | list of n_targets such arrays] Maximum of covariances (in absolute value) at each iteration. `n_alphas` is either `max_iter`, `n_features`, or the number of nodes in the path with correlation greater than `alpha`, whichever is smaller.

active_ [list, length = n_alphas | list of n_targets such lists] Indices of active variables at the end of the path.

coef_path_ [array, shape (n_features, n_alphas + 1) or list] If a list is passed it's expected to be one of n_targets such arrays. The varying values of the coefficients along the path. It is not present if the `fit_path` parameter is `False`.

coef_ [array, shape (n_features,) or (n_targets, n_features)] Parameter vector (w in the formulation formula).

intercept_ [float | array, shape (n_targets,)] Independent term in decision function.

n_iter_ [array-like or int.] The number of iterations taken by lars_path to find the grid of alphas for each target.

See also:

lars_path

lasso_path

Lasso

LassoCV

LassoLarsCV

LassoLarsIC

sklearn.decomposition.sparse_encode

Examples

```
>>> from sklearn import linear_model
>>> reg = linear_model.LassoLars(alpha=0.01)
>>> reg.fit([[-1, 1], [0, 0], [1, 1]], [-1, 0, -1])
...
LassoLars(alpha=0.01, copy_X=True, eps=..., fit_intercept=True,
     fit_path=True, max_iter=500, normalize=True, positive=False,
     precompute='auto', verbose=False)
>>> print(reg.coef_)
[ 0.         -0.963257...]
```

Methods

fit(self, X, y[, Xy])	Fit the model using X, y as training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *fit_intercept=True*, *verbose=False*, *normalize=True*, *precompute='auto'*, *max_iter=500*, *eps=2.220446049250313e-16*, *copy_X=True*, *fit_path=True*, *positive=False*)

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

Fit the model using X, y as training data.

Parameters

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

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values.

Xy [array-like, shape (n_samples,) or (n_samples, n_targets), optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

Returns

self [object] returns an instance of self.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)

Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

6.22.8 sklearn.linear_model.LinearRegression

class sklearn.linear_model.**LinearRegression** (*fit_intercept=True*, *normalize=False*, *copy_X=True*, *n_jobs=None*)

Ordinary least squares Linear Regression.

Parameters

fit_intercept [boolean, optional, default True] whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This will only provide speedup for n_targets > 1 and sufficient large problems. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

coef_ [array, shape (n_features,) or (n_targets, n_features)] Estimated coefficients for the linear regression problem. If multiple targets are passed during the fit (y 2D), this is a 2D array of shape (n_targets, n_features), while if only one target is passed, this is a 1D array of length n_features.

intercept_ [array] Independent term in the linear model.

Notes

From the implementation point of view, this is just plain Ordinary Least Squares (scipy.linalg.lstsq) wrapped as a predictor object.

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
>>> # y = 1 * x_0 + 2 * x_1 + 3
>>> y = np.dot(X, np.array([1, 2])) + 3
>>> reg = LinearRegression().fit(X, y)
>>> reg.score(X, y)
1.0
>>> reg.coef_
array([1., 2.])
>>> reg.intercept_
3.0000...
>>> reg.predict(np.array([[3, 5]]))
array([16.])
```

Methods

`fit`(self, X, y[, sample_weight])	Fit linear model.
`get_params`(self[, deep])	Get parameters for this estimator.
`predict`(self, X)	Predict using the linear model
`score`(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *n_jobs=None*)

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit linear model.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] Training data

y [array_like, shape (n_samples, n_targets)] Target values. Will be cast to X's dtype if necessary

sample_weight [numpy array of shape [n_samples]] Individual weights for each sample

New in version 0.17: parameter *sample_weight* support to LinearRegression.

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.linear_model.LinearRegression`

- *Isotonic Regression*
- *Face completion with a multi-output estimators*
- *Plot individual and voting regression predictions*
- *Ordinary Least Squares and Ridge Regression Variance*
- *Logistic function*
- *Linear Regression Example*
- *Robust linear model estimation using RANSAC*
- *Sparsity Example: Fitting only features 1 and 2*
- *Theil-Sen Regression*
- *Robust linear estimator fitting*
- *Automatic Relevance Determination Regression (ARD)*
- *Bayesian Ridge Regression*
- *Plotting Cross-Validated Predictions*
- *Underfitting vs. Overfitting*
- *Using KBinsDiscretizer to discretize continuous features*

6.22.9 sklearn.linear_model.LogisticRegression

class sklearn.linear_model.**LogisticRegression**(*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *random_state=None*, *solver='warn'*, *max_iter=100*, *multi_class='warn'*, *verbose=0*, *warm_start=False*, *n_jobs=None*, *l1_ratio=None*)

Logistic Regression (aka logit, MaxEnt) classifier.

In the multiclass case, the training algorithm uses the one-vs-rest (OvR) scheme if the 'multi_class' option is set to 'ovr', and uses the cross-entropy loss if the 'multi_class' option is set to 'multinomial'. (Currently the 'multinomial' option is supported only by the 'lbfgs', 'sag', 'saga' and 'newton-cg' solvers.)

This class implements regularized logistic regression using the 'liblinear' library, 'newton-cg', 'sag', 'saga' and 'lbfgs' solvers. **Note that regularization is applied by default**. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2 regularization with primal formulation, or no regularization. The 'liblinear' solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty. The Elastic-Net regularization is only supported by the 'saga' solver.

Read more in the *User Guide*.

Parameters

penalty [str, 'l1', 'l2', 'elasticnet' or 'none', optional (default='l2')] Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is only supported by the 'saga' solver. If 'none' (not supported by the liblinear solver), no regularization is applied.

New in version 0.19: l1 penalty with SAGA solver (allowing 'multinomial' + L1)

dual [bool, optional (default=False)] Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.

tol [float, optional (default=1e-4)] Tolerance for stopping criteria.

C [float, optional (default=1.0)] Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

fit_intercept [bool, optional (default=True)] Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

intercept_scaling [float, optional (default=1)] Useful only when the solver 'liblinear' is used and self.fit_intercept is set to True. In this case, x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equal to intercept_scaling is appended to the instance vector. The intercept becomes `intercept_scaling * synthetic_feature_weight`.

Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

class_weight [dict or 'balanced', optional (default=None)] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

New in version 0.17: *class_weight='balanced'*

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag' or 'liblinear'.

solver [str, {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, optional (default='liblinear').] Algorithm to use in the optimization problem.

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones.
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss; 'liblinear' is limited to one-versus-rest schemes.
- 'newton-cg', 'lbfgs', 'sag' and 'saga' handle L2 or no penalty
- 'liblinear' and 'saga' also handle L1 penalty
- 'saga' also supports 'elasticnet' penalty
- 'liblinear' does not handle no penalty

Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

New in version 0.17: Stochastic Average Gradient descent solver.

New in version 0.19: SAGA solver.

Changed in version 0.20: Default will change from 'liblinear' to 'lbfgs' in 0.22.

max_iter [int, optional (default=100)] Maximum number of iterations taken for the solvers to converge.

multi_class [str, {'ovr', 'multinomial', 'auto'}, optional (default='ovr')] If the option chosen is 'ovr', then a binary problem is fit for each label. For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. 'multinomial' is unavailable when solver='liblinear'. 'auto' selects 'ovr' if the data is binary, or if solver='liblinear', and otherwise selects 'multinomial'.

New in version 0.18: Stochastic Average Gradient descent solver for 'multinomial' case.

Changed in version 0.20: Default will change from 'ovr' to 'auto' in 0.22.

verbose [int, optional (default=0)] For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.

warm_start [bool, optional (default=False)] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Useless for liblinear solver. See *the Glossary*.

New in version 0.17: *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.

n_jobs [int or None, optional (default=None)] Number of CPU cores used when parallelizing over classes if multi_class='ovr'". This parameter is ignored when the `solver` is set to

'liblinear' regardless of whether 'multi_class' is specified or not. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. See *Glossary* for more details.

l1_ratio [float or None, optional (default=None)] The Elastic-Net mixing parameter, with 0 <= l1_ratio <= 1. Only used if penalty='elasticnet'`. Setting ``l1_ratio=0 is equivalent to using penalty='l2', while setting l1_ratio=1 is equivalent to using penalty='l1'. For 0 < l1_ratio <1, the penalty is a combination of L1 and L2.

Attributes

classes_ [array, shape (n_classes,)] A list of class labels known to the classifier.

coef_ [array, shape (1, n_features) or (n_classes, n_features)] Coefficient of the features in the decision function.

*coef_* is of shape (1, n_features) when the given problem is binary. In particular, when multi_class='multinomial', *coef_* corresponds to outcome 1 (True) and -coef_ corresponds to outcome 0 (False).

intercept_ [array, shape (1,) or (n_classes,)] Intercept (a.k.a. bias) added to the decision function.

If fit_intercept is set to False, the intercept is set to zero. intercept_ is of shape (1,) when the given problem is binary. In particular, when multi_class='multinomial', intercept_ corresponds to outcome 1 (True) and -intercept_ corresponds to outcome 0 (False).

n_iter_ [array, shape (n_classes,) or (1,)] Actual number of iterations for all classes. If binary or multinomial, it returns only 1 element. For liblinear solver, only the maximum number of iteration across all classes is given.

Changed in version 0.20: In SciPy <= 1.0.0 the number of lbfgs iterations may exceed max_iter. n_iter_ will now report at most max_iter.

See also:

SGDClassifier incrementally trained logistic regression (when given the parameter loss="log").

LogisticRegressionCV Logistic regression with built-in cross validation

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller tol parameter.

Predict output may not match that of standalone liblinear in certain cases. See *differences from liblinear* in the narrative documentation.

References

LIBLINEAR – A Library for Large Linear Classification https://www.csie.ntu.edu.tw/~cjlin/liblinear/

SAG – Mark Schmidt, Nicolas Le Roux, and Francis Bach Minimizing Finite Sums with the Stochastic Average Gradient https://hal.inria.fr/hal-00860051/document

SAGA – Defazio, A., Bach F. & Lacoste-Julien S. (2014). SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives https://arxiv.org/abs/1407.0202

Hsiang-Fu Yu, Fang-Lan Huang, Chih-Jen Lin (2011). Dual coordinate descent methods for logistic regression and maximum entropy models. Machine Learning 85(1-2):41-75. https://www.csie.ntu.edu.tw/~cjlin/papers/maxent_dual.pdf

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegression
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegression(random_state=0, solver='lbfgs',
...                          multi_class='multinomial').fit(X, y)
>>> clf.predict(X[:2, :])
array([0, 0])
>>> clf.predict_proba(X[:2, :])
array([[9.8...e-01, 1.8...e-02, 1.4...e-08],
       [9.7...e-01, 2.8...e-02, ...e-08]])
>>> clf.score(X, y)
0.97...
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, sample_weight])	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class labels for samples in X.
predict_log_proba(self, X)	Log of probability estimates.
predict_proba(self, X)	Probability estimates.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
sparsify(self)	Convert coefficient matrix to sparse format.

__init__ (*self*, *penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *random_state=None*, *solver='warn'*, *max_iter=100*, *multi_class='warn'*, *verbose=0*, *warm_start=False*, *n_jobs=None*, *l1_ratio=None*)

decision_function (*self*, *X*)
Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify(*self*)
Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self [estimator]

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the model according to the given training data.

Parameters

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

y [array-like, shape (n_samples,)] Target vector relative to X.

sample_weight [array-like, shape (n_samples,) optional] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

New in version 0.17: *sample_weight* support to LogisticRegression.

Returns

self [object]

Notes

The SAGA solver supports both float64 and float32 bit arrays.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

predict_log_proba(*self*, *X*)
Log of probability estimates.

The returned estimates for all classes are ordered by the label of classes.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

T [array-like, shape = [n_samples, n_classes]] Returns the log-probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

predict_proba (*self*, *X*)
Probability estimates.

The returned estimates for all classes are ordered by the label of classes.

For a multi_class problem, if multi_class is set to be "multinomial" the softmax function is used to find the predicted probability of each class. Else use a one-vs-rest approach, i.e calculate the probability of each class assuming it to be positive using the logistic function. and normalize these values across all the classes.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

T [array-like, shape = [n_samples, n_classes]] Returns the probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

sparsify (*self*)
Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self [estimator]

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.linear_model.LogisticRegression`

- *Compact estimator representations*
- *Comparison of Calibration of Classifiers*
- *Probability Calibration curves*
- *Plot classification probability*
- *Column Transformer with Mixed Types*
- *Plot class probabilities calculated by the VotingClassifier*
- *Feature transformations with ensembles of trees*
- *Digits Classification Exercise*
- *Regularization path of L1- Logistic Regression*
- *Logistic function*
- *Logistic Regression 3-class Classifier*
- *Comparing various online solvers*
- *MNIST classfification using multinomial logistic + L1*
- *Plot multinomial and One-vs-Rest Logistic Regression*
- *L1 Penalty and Sparsity in Logistic Regression*
- *Multiclass sparse logisitic regression on newgroups20*
- *Classifier Chain*
- *Restricted Boltzmann Machine features for digit classification*
- *Feature discretization*

6.22.10 `sklearn.linear_model`.MultiTaskLasso

class `sklearn.linear_model.`**`MultiTaskLasso`**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *random_state=None*, *selection='cyclic'*)

Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||Y - XW||^2_Fro + alpha * ||W||_21
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

alpha [float, optional] Constant that multiplies the L1/L2 term. Defaults to 1.0

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

warm_start [bool, optional] When set to `True`, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4

Attributes

coef_ [array, shape (n_tasks, n_features)] Parameter vector (W in the cost function formula). Note that `coef_` stores the transpose of `W`, `W.T`.

intercept_ [array, shape (n_tasks,)] independent term in decision function.

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance.

See also:

`MultiTaskLasso` Multi-task L1/L2 Lasso with built-in cross-validation

`Lasso`

`MultiTaskElasticNet`

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.MultiTaskLasso(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [[0, 0], [1, 1], [2, 2]])
...
MultiTaskLasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
        normalize=False, random_state=None, selection='cyclic', tol=0.0001,
        warm_start=False)
>>> print(clf.coef_)
[[0.89393398 0.        ]
 [0.89393398 0.        ]]
>>> print(clf.intercept_)
[0.10606602 0.10606602]
```

Methods

fit(self, X, y)	Fit MultiTaskElasticNet model with coordinate descent
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *random_state=None*, *selection='cyclic'*)

fit (*self*, *X*, *y*)

Fit MultiTaskElasticNet model with coordinate descent

Parameters

X [ndarray, shape (n_samples, n_features)] Data

y [ndarray, shape (n_samples, n_tasks)] Target. Will be cast to X's dtype if necessary

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a Fortran-contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

path(*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)

Compute elastic net path with coordinate descent

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values

l1_ratio [float, optional] float between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

predict (*self*, *X*)

Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

sparse_coef_
sparse representation of the fitted `coef_`

Examples using `sklearn.linear_model.MultiTaskLasso`

- *Joint feature selection with multi-task Lasso*

6.22.11 sklearn.linear_model.MultiTaskElasticNet

class sklearn.linear_model.**MultiTaskElasticNet**(*alpha=1.0*, *l1_ratio=0.5*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *random_state=None*, *selection='cyclic'*)

Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer

The optimization objective for MultiTaskElasticNet is:

```
(1 / (2 * n_samples)) * ||Y - XW||_Fro^2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = sum_i sqrt(sum_j w_ij ^ 2)
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

alpha [float, optional] Constant that multiplies the L1/L2 term. Defaults to 1.0

l1_ratio [float] The ElasticNet mixing parameter, with 0 < l1_ratio <= 1. For l1_ratio = 1 the penalty is an L1/L2 penalty. For l1_ratio = 0 it is an L2 penalty. For `0 < l1_ratio < 1`, the penalty is a combination of L1/L2 and L2.

fit_intercept [boolean] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

warm_start [bool, optional] When set to `True`, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

intercept_ [array, shape (n_tasks,)] Independent term in decision function.

coef_ [array, shape (n_tasks, n_features)] Parameter vector (W in the cost function formula). If a 1D y is passed in at fit (non multi-task usage), `coef_` is then a 1D array. Note that `coef_` stores the transpose of `W`, `W.T`.

n_iter_ [int] number of iterations run by the coordinate descent solver to reach the specified tolerance.

See also:

`MultiTaskElasticNet` Multi-task L1/L2 ElasticNet with built-in cross-validation.

`ElasticNet`

`MultiTaskLasso`

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.MultiTaskElasticNet(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [[0, 0], [1, 1], [2, 2]])
...
MultiTaskElasticNet(alpha=0.1, copy_X=True, fit_intercept=True,
        l1_ratio=0.5, max_iter=1000, normalize=False, random_state=None,
        selection='cyclic', tol=0.0001, warm_start=False)
>>> print(clf.coef_)
[[0.45663524 0.45612256]
 [0.45663524 0.45612256]]
>>> print(clf.intercept_)
[0.0872422 0.0872422]
```

Methods

fit(self, X, y)	Fit MultiTaskElasticNet model with coordinate descent
get_params(self[, deep])	Get parameters for this estimator.
path(X, y[, l1_ratio, eps, n_alphas, ...])	Compute elastic net path with coordinate descent
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *l1_ratio=0.5*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *random_state=None*, *selection='cyclic'*)

fit(*self*, *X*, *y*)

Fit MultiTaskElasticNet model with coordinate descent

Parameters

X [ndarray, shape (n_samples, n_features)] Data

y [ndarray, shape (n_samples, n_tasks)] Target. Will be cast to X's dtype if necessary

Notes

Coordinate descent is an algorithm that considers each column of data at a time hence it will automatically convert the X input as a Fortran-contiguous numpy array if necessary.

To avoid memory re-allocation it is advised to allocate the initial data in memory directly using that format.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

path(*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)

Compute elastic net path with coordinate descent

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values

l1_ratio [float, optional] float between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

predict(*self*, *X*)

Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

sparse_coef_
sparse representation of the fitted `coef_`

6.22.12 sklearn.linear_model.OrthogonalMatchingPursuit

class sklearn.linear_model.**OrthogonalMatchingPursuit**(*n_nonzero_coefs=None*, *tol=None*, *fit_intercept=True*, *normalize=True*, *precompute='auto'*)

Orthogonal Matching Pursuit model (OMP)

Read more in the *User Guide*.

Parameters

n_nonzero_coefs [int, optional] Desired number of non-zero entries in the solution. If None (by default) this value is set to 10% of n_features.

tol [float, optional] Maximum norm of the residual. If not None, overrides n_nonzero_coefs.

fit_intercept [boolean, optional] whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default True] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *sklearn.preprocessing.StandardScaler* before calling `fit` on an estimator with `normalize=False`.

precompute [{True, False, 'auto'}, default 'auto'] Whether to use a precomputed Gram and Xy matrix to speed up calculations. Improves performance when *n_targets* or *n_samples* is very large. Note that if you already have such matrices, you can pass them directly to the fit method.

Attributes

coef_ [array, shape (n_features,) or (n_targets, n_features)] parameter vector (w in the formula)

intercept_ [float or array, shape (n_targets,)] independent term in decision function.

n_iter_ [int or array-like] Number of active features across every target.

See also:

orthogonal_mp

orthogonal_mp_gram

lars_path

Lars

LassoLars

decomposition.sparse_encode

OrthogonalMatchingPursuitCV

Notes

Orthogonal matching pursuit was introduced in G. Mallat, Z. Zhang, Matching pursuits with time-frequency dictionaries, IEEE Transactions on Signal Processing, Vol. 41, No. 12. (December 1993), pp. 3397-3415. (http://blanche.polytechnique.fr/~mallat/papiers/MallatPursuit93.pdf)

This implementation is based on Rubinstein, R., Zibulevsky, M. and Elad, M., Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit Technical Report - CS Technion, April 2008. https://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

Examples

```
>>> from sklearn.linear_model import OrthogonalMatchingPursuit
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(noise=4, random_state=0)
>>> reg = OrthogonalMatchingPursuit().fit(X, y)
>>> reg.score(X, y)
0.9991...
>>> reg.predict(X[:1,])
array([-78.3854...])
```

Methods

fit(self, X, y)	Fit the model using X, y as training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_nonzero_coefs=None*, *tol=None*, *fit_intercept=True*, *normalize=True*, *precompute='auto'*)

fit(*self*, *X*, *y*)

Fit the model using X, y as training data.

Parameters

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

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Target values. Will be cast to X's dtype if necessary

Returns

self [object] returns an instance of self.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.linear_model.OrthogonalMatchingPursuit`

- *Orthogonal Matching Pursuit*

6.22.13 `sklearn.linear_model`.PassiveAggressiveClassifier

class `sklearn.linear_model.`**PassiveAggressiveClassifier**(*C=1.0*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *shuffle=True*, *verbose=0*, *loss='hinge'*, *n_jobs=None*, *random_state=None*, *warm_start=False*, *class_weight=None*, *average=False*)

Passive Aggressive Classifier

Read more in the *User Guide*.

Parameters

C [float] Maximum step size (regularization). Defaults to 1.0.

fit_intercept [bool, default=False] Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.

max_iter [int, optional (default=1000)] The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the *partial_fit*.

New in version 0.19.

tol [float or None, optional (default=1e-3)] The stopping criterion. If it is not None, the iterations will stop when (loss > previous_loss - tol).

New in version 0.19.

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation. score is not improving. If set to True, it will automatically set aside a stratified fraction of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.

New in version 0.20.

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change [int, default=5] Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

shuffle [bool, default=True] Whether or not the training data should be shuffled after each epoch.

verbose [integer, optional] The verbosity level

loss [string, optional] The loss function to be used: hinge: equivalent to PA-I in the reference paper. squared_hinge: equivalent to PA-II in the reference paper.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional, default=None] The seed of the pseudo random number generator to use when shuffling the data. 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`.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

Repeatedly calling fit or partial_fit when warm_start is True can result in a different solution than when calling fit a single time because of the way the data is shuffled.

class_weight [dict, {class_label: weight} or "balanced" or None, optional] Preset for the class_weight fit parameter.

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

New in version 0.17: parameter *class_weight* to automatically weight samples.

average [bool or int, optional] When set to True, computes the averaged SGD weights and stores the result in the `coef_` attribute. If set to an int greater than 1, averaging will begin once the total number of samples seen reaches average. So average=10 will begin averaging after seeing 10 samples.

New in version 0.19: parameter *average* to use weights averaging in SGD

Attributes

coef_ [array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]] Weights assigned to the features.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

n_iter_ [int] The actual number of iterations to reach the stopping criterion. For multiclass fits, it is the maximum over every binary fit.

See also:

SGDClassifier

Perceptron

References

Online Passive-Aggressive Algorithms <http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf> K. Crammer, O. Dekel, J. Keshat, S. Shalev-Shwartz, Y. Singer - JMLR (2006)

Examples

```
>>> from sklearn.linear_model import PassiveAggressiveClassifier
>>> from sklearn.datasets import make_classification
```

```
>>> X, y = make_classification(n_features=4, random_state=0)
>>> clf = PassiveAggressiveClassifier(max_iter=1000, random_state=0,
... tol=1e-3)
>>> clf.fit(X, y)
PassiveAggressiveClassifier(C=1.0, average=False, class_weight=None,
              early_stopping=False, fit_intercept=True, loss='hinge',
              max_iter=1000, n_iter_no_change=5, n_jobs=None,
              random_state=0, shuffle=True, tol=0.001,
              validation_fraction=0.1, verbose=0, warm_start=False)
>>> print(clf.coef_)
[[0.26642044 0.45070924 0.67251877 0.64185414]]
>>> print(clf.intercept_)
[1.84127814]
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, coef_init, intercept_init])	Fit linear model with Passive Aggressive algorithm.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes])	Fit linear model with Passive Aggressive algorithm.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, *args, **kwargs)	
sparsify(self)	Convert coefficient matrix to sparse format.

__init__(*self*, *C=1.0*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *shuffle=True*, *verbose=0*, *loss='hinge'*, *n_jobs=None*, *random_state=None*, *warm_start=False*, *class_weight=None*, *average=False*)

decision_function(*self*, *X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify(*self*)

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self [estimator]

fit(*self*, *X*, *y*, *coef_init=None*, *intercept_init=None*)

Fit linear model with Passive Aggressive algorithm.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training data

y [numpy array of shape [n_samples]] Target values

coef_init [array, shape = [n_classes,n_features]] The initial coefficients to warm-start the optimization.

intercept_init [array, shape = [n_classes]] The initial intercept to warm-start the optimization.

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X*, *y*, *classes=None*)

Fit linear model with Passive Aggressive algorithm.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Subset of the training data

y [numpy array of shape [n_samples]] Subset of the target values

classes [array, shape = [n_classes]] Classes across all calls to partial_fit. Can be obtained by via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in `classes`.

Returns

self [returns an instance of self.]

predict (*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

sparsify (*self*)
Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self [estimator]

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.linear_model.PassiveAggressiveClassifier`

- *Out-of-core classification of text documents*
- *Comparing various online solvers*
- *Classification of text documents using sparse features*

6.22.14 sklearn.linear_model.PassiveAggressiveRegressor

class sklearn.linear_model.**PassiveAggressiveRegressor**(*C=1.0*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *shuffle=True*, *verbose=0*, *loss='epsilon_insensitive'*, *epsilon=0.1*, *random_state=None*, *warm_start=False*, *average=False*)

Passive Aggressive Regressor

Read more in the *User Guide*.

Parameters

C [float] Maximum step size (regularization). Defaults to 1.0.

fit_intercept [bool] Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

max_iter [int, optional (default=1000)] The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the *partial_fit*.

New in version 0.19.

tol [float or None, optional (default=1e-3)] The stopping criterion. If it is not None, the iterations will stop when (loss > previous_loss - tol).

New in version 0.19.

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation. score is not improving. If set to True, it will automatically set aside a fraction of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.

New in version 0.20.

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change [int, default=5] Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

shuffle [bool, default=True] Whether or not the training data should be shuffled after each epoch.

verbose [integer, optional] The verbosity level

loss [string, optional] The loss function to be used: epsilon_insensitive: equivalent to PA-I in the reference paper. squared_epsilon_insensitive: equivalent to PA-II in the reference paper.

epsilon [float] If the difference between the current prediction and the correct label is below this threshold, the model is not updated.

random_state [int, RandomState instance or None, optional, default=None] The seed of the pseudo random number generator to use when shuffling the data. 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`.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

Repeatedly calling fit or partial_fit when warm_start is True can result in a different solution than when calling fit a single time because of the way the data is shuffled.

average [bool or int, optional] When set to True, computes the averaged SGD weights and stores the result in the `coef_` attribute. If set to an int greater than 1, averaging will begin once the total number of samples seen reaches average. So average=10 will begin averaging after seeing 10 samples.

New in version 0.19: parameter *average* to use weights averaging in SGD

Attributes

coef_ [array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]] Weights assigned to the features.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

n_iter_ [int] The actual number of iterations to reach the stopping criterion.

See also:

SGDRegressor

References

Online Passive-Aggressive Algorithms <http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf> K. Crammer, O. Dekel, J. Keshat, S. Shalev-Shwartz, Y. Singer - JMLR (2006)

Examples

```
>>> from sklearn.linear_model import PassiveAggressiveRegressor
>>> from sklearn.datasets import make_regression
```

```
>>> X, y = make_regression(n_features=4, random_state=0)
>>> regr = PassiveAggressiveRegressor(max_iter=100, random_state=0,
... tol=1e-3)
>>> regr.fit(X, y)
PassiveAggressiveRegressor(C=1.0, average=False, early_stopping=False,
              epsilon=0.1, fit_intercept=True, loss='epsilon_insensitive',
              max_iter=100, n_iter_no_change=5, random_state=0,
              shuffle=True, tol=0.001, validation_fraction=0.1,
              verbose=0, warm_start=False)
>>> print(regr.coef_)
[20.48736655 34.18818427 67.59122734 87.94731329]
>>> print(regr.intercept_)
[-0.02306214]
>>> print(regr.predict([[0, 0, 0, 0]]))
[-0.02306214]
```

Methods

densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, coef_init, intercept_init])	Fit linear model with Passive Aggressive algorithm.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y)	Fit linear model with Passive Aggressive algorithm.
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, *args, **kwargs)	
sparsify(self)	Convert coefficient matrix to sparse format.

__init__(*self*, *C=1.0*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *shuffle=True*, *verbose=0*, *loss='epsilon_insensitive'*, *epsilon=0.1*, *random_state=None*, *warm_start=False*, *average=False*)

densify(*self*)

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self [estimator]

fit(*self*, *X*, *y*, *coef_init=None*, *intercept_init=None*)

Fit linear model with Passive Aggressive algorithm.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training data

y [numpy array of shape [n_samples]] Target values

coef_init [array, shape = [n_features]] The initial coefficients to warm-start the optimization.

intercept_init [array, shape = [1]] The initial intercept to warm-start the optimization.

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X*, *y*)

Fit linear model with Passive Aggressive algorithm.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Subset of training data

y [numpy array of shape [n_samples]] Subset of target values

Returns

self [returns an instance of self.]

predict (*self*, *X*)

Predict using the linear model

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)]

Returns

array, shape (n_samples,) Predicted target values per element in X.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

sparsify (*self*)

Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self [estimator]

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

6.22.15 sklearn.linear_model.Perceptron

class sklearn.linear_model.**Perceptron**(*penalty=None*, *alpha=0.0001*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *eta0=1.0*, *n_jobs=None*, *random_state=0*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *class_weight=None*, *warm_start=False*)

Read more in the *User Guide*.

Parameters

penalty [None, 'l2' or 'l1' or 'elasticnet'] The penalty (aka regularization term) to be used. Defaults to None.

alpha [float] Constant that multiplies the regularization term if regularization is used. Defaults to 0.0001

fit_intercept [bool] Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

max_iter [int, optional (default=1000)] The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the *partial_fit*.

New in version 0.19.

tol [float or None, optional (default=1e-3)] The stopping criterion. If it is not None, the iterations will stop when (loss > previous_loss - tol).

New in version 0.19.

shuffle [bool, optional, default True] Whether or not the training data should be shuffled after each epoch.

verbose [integer, optional] The verbosity level

eta0 [double] Constant by which the updates are multiplied. Defaults to 1.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator to use when shuffling the data. 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`.

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation. score is not improving. If set to True, it will automatically set aside a stratified

fraction of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.

New in version 0.20.

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change [int, default=5] Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

class_weight [dict, {class_label: weight} or "balanced" or None, optional] Preset for the class_weight fit parameter.

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

Attributes

coef_ [array, shape = [1, n_features] if n_classes == 2 else [n_classes, n_features]] Weights assigned to the features.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

n_iter_ [int] The actual number of iterations to reach the stopping criterion. For multiclass fits, it is the maximum over every binary fit.

See also:

SGDClassifier

Notes

`Perceptron` is a classification algorithm which shares the same underlying implementation with `SGDClassifier`. In fact, `Perceptron()` is equivalent to `SGDClassifier(loss="perceptron", eta0=1, learning_rate="constant", penalty=None)`.

References

https://en.wikipedia.org/wiki/Perceptron and references therein.

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.linear_model import Perceptron
>>> X, y = load_digits(return_X_y=True)
>>> clf = Perceptron(tol=1e-3, random_state=0)
>>> clf.fit(X, y)
```

```
Perceptron(alpha=0.0001, class_weight=None, early_stopping=False, eta0=1.0,
      fit_intercept=True, max_iter=1000, n_iter_no_change=5, n_jobs=None,
      penalty=None, random_state=0, shuffle=True, tol=0.001,
      validation_fraction=0.1, verbose=0, warm_start=False)
>>> clf.score(X, y)
0.939...
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, coef_init, intercept_init, ...])	Fit linear model with Stochastic Gradient Descent.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Perform one epoch of stochastic gradient descent on given samples.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, *args, **kwargs)	
sparsify(self)	Convert coefficient matrix to sparse format.

__init__ (*self*, *penalty=None*, *alpha=0.0001*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *eta0=1.0*, *n_jobs=None*, *random_state=0*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *class_weight=None*, *warm_start=False*)

decision_function (*self*, *X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify (*self*)

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self [estimator]

fit (*self*, *X*, *y*, *coef_init=None*, *intercept_init=None*, *sample_weight=None*)

Fit linear model with Stochastic Gradient Descent.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data

y [numpy array, shape (n_samples,)] Target values

coef_init [array, shape (n_classes, n_features)] The initial coefficients to warm-start the optimization.

intercept_init [array, shape (n_classes,)] The initial intercept to warm-start the optimization.

sample_weight [array-like, shape (n_samples,), optional] Weights applied to individual samples. If not provided, uniform weights are assumed. These weights will be multiplied with class_weight (passed through the constructor) if class_weight is specified

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Perform one epoch of stochastic gradient descent on given samples.

Internally, this method uses `max_iter = 1`. Therefore, it is not guaranteed that a minimum of the cost function is reached after calling it once. Matters such as objective convergence and early stopping should be handled by the user.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Subset of the training data

y [numpy array, shape (n_samples,)] Subset of the target values

classes [array, shape (n_classes,)] Classes across all calls to partial_fit. Can be obtained by via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in `classes`.

sample_weight [array-like, shape (n_samples,), optional] Weights applied to individual samples. If not provided, uniform weights are assumed.

Returns

self [returns an instance of self.]

predict(*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

sparsify(*self*)
Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self [estimator]

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.linear_model.Perceptron`

- *Out-of-core classification of text documents*
- *Comparing various online solvers*
- *Classification of text documents using sparse features*

6.22.16 `sklearn.linear_model`.RANSACRegressor

class `sklearn.linear_model.`**RANSACRegressor**(*base_estimator=None*, *min_samples=None*, *residual_threshold=None*, *is_data_valid=None*, *is_model_valid=None*, *max_trials=100*, *max_skips=inf*, *stop_n_inliers=inf*, *stop_score=inf*, *stop_probability=0.99*, *loss='absolute_loss'*, *random_state=None*)
RANSAC (RANdom SAmple Consensus) algorithm.

RANSAC is an iterative algorithm for the robust estimation of parameters from a subset of inliers from the complete data set. More information can be found in the general documentation of linear models.

A detailed description of the algorithm can be found in the documentation of the `linear_model` sub-package.

Read more in the *User Guide*.

Parameters

base_estimator [object, optional] Base estimator object which implements the following methods:

- `fit(X, y)`: Fit model to given training data and target values.
- `score(X, y)`: Returns the mean accuracy on the given test data, which is used for the stop criterion defined by `stop_score`. Additionally, the score is used to decide which of two equally large consensus sets is chosen as the better one.
- `predict(X)`: Returns predicted values using the linear model, which is used to compute residual error using loss function.

If `base_estimator` is None, then `base_estimator=sklearn.linear_model.LinearRegression()` is used for target values of dtype float.

Note that the current implementation only supports regression estimators.

min_samples [int (>= 1) or float ([0, 1]), optional] Minimum number of samples chosen randomly from original data. Treated as an absolute number of samples for `min_samples >= 1`, treated as a relative number `ceil(min_samples * X.shape[0])` for `min_samples < 1`. This is typically chosen as the minimal number of samples necessary to estimate the given `base_estimator`. By default a `sklearn.linear_model.LinearRegression()` estimator is assumed and `min_samples` is chosen as `X.shape[1] + 1`.

residual_threshold [float, optional] Maximum residual for a data sample to be classified as an inlier. By default the threshold is chosen as the MAD (median absolute deviation) of the target values *y*.

is_data_valid [callable, optional] This function is called with the randomly selected data before the model is fitted to it: `is_data_valid(X, y)`. If its return value is False the current randomly chosen sub-sample is skipped.

is_model_valid [callable, optional] This function is called with the estimated model and the randomly selected data: `is_model_valid(model, X, y)`. If its return value is False the current randomly chosen sub-sample is skipped. Rejecting samples with this function is computationally costlier than with `is_data_valid`. `is_model_valid` should therefore only be used if the estimated model is needed for making the rejection decision.

max_trials [int, optional] Maximum number of iterations for random sample selection.

max_skips [int, optional] Maximum number of iterations that can be skipped due to finding zero inliers or invalid data defined by `is_data_valid` or invalid models defined by `is_model_valid`.

New in version 0.19.

stop_n_inliers [int, optional] Stop iteration if at least this number of inliers are found.

stop_score [float, optional] Stop iteration if score is greater equal than this threshold.

stop_probability [float in range [0, 1], optional] RANSAC iteration stops if at least one outlier-free set of the training data is sampled in RANSAC. This requires to generate at least N samples (iterations):

```
N >= log(1 - probability) / log(1 - e**m)
```

where the probability (confidence) is typically set to high value such as 0.99 (the default) and e is the current fraction of inliers w.r.t. the total number of samples.

loss [string, callable, optional, default "absolute_loss"] String inputs, "absolute_loss" and "squared_loss" are supported which find the absolute loss and squared loss per sample respectively.

If `loss` is a callable, then it should be a function that takes two arrays as inputs, the true and predicted value and returns a 1-D array with the i-th value of the array corresponding to the loss on `X[i]`.

If the loss on a sample is greater than the `residual_threshold`, then this sample is classified as an outlier.

random_state [int, RandomState instance or None, optional, default None] The generator used to initialize the centers. 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`.

Attributes

estimator_ [object] Best fitted model (copy of the `base_estimator` object).

n_trials_ [int] Number of random selection trials until one of the stop criteria is met. It is always `<= max_trials`.

inlier_mask_ [bool array of shape [n_samples]] Boolean mask of inliers classified as `True`.

n_skips_no_inliers_ [int] Number of iterations skipped due to finding zero inliers.

New in version 0.19.

n_skips_invalid_data_ [int] Number of iterations skipped due to invalid data defined by `is_data_valid`.

New in version 0.19.

n_skips_invalid_model_ [int] Number of iterations skipped due to an invalid model defined by `is_model_valid`.

New in version 0.19.

References

[R80ce5b25cf9d-1], *[R80ce5b25cf9d-2]*, *[R80ce5b25cf9d-3]*

Examples

```
>>> from sklearn.linear_model import RANSACRegressor
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(
...     n_samples=200, n_features=2, noise=4.0, random_state=0)
>>> reg = RANSACRegressor(random_state=0).fit(X, y)
>>> reg.score(X, y)
0.9885...
>>> reg.predict(X[:1,])
array([-31.9417...])
```

Methods

fit(self, X, y[, sample_weight])	Fit estimator using RANSAC algorithm.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the estimated model.
score(self, X, y)	Returns the score of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *base_estimator=None*, *min_samples=None*, *residual_threshold=None*, *is_data_valid=None*, *is_model_valid=None*, *max_trials=100*, *max_skips=inf*, *stop_n_inliers=inf*, *stop_score=inf*, *stop_probability=0.99*, *loss='absolute_loss'*, *random_state=None*)

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

Fit estimator using RANSAC algorithm.

Parameters

X [array-like or sparse matrix, shape [n_samples, n_features]] Training data.

y [array-like, shape = [n_samples] or [n_samples, n_targets]] Target values.

sample_weight [array-like, shape = [n_samples]] Individual weights for each sample raises error if sample_weight is passed and base_estimator fit method does not support it.

Raises

ValueError If no valid consensus set could be found. This occurs if `is_data_valid` and `is_model_valid` return False for all `max_trials` randomly chosen sub-samples.

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)

Predict using the estimated model.

This is a wrapper for `estimator_.predict(X)`.

Parameters

X [numpy array of shape [n_samples, n_features]]

Returns

y [array, shape = [n_samples] or [n_samples, n_targets]] Returns predicted values.

score (*self*, *X*, *y*)

Returns the score of the prediction.

This is a wrapper for `estimator_.score(X, y)`.

Parameters

X [numpy array or sparse matrix of shape [n_samples, n_features]] Training data.

y [array, shape = [n_samples] or [n_samples, n_targets]] Target values.

Returns

z [float] Score of the prediction.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.linear_model.RANSACRegressor`

- *Robust linear model estimation using RANSAC*
- *Theil-Sen Regression*
- *Robust linear estimator fitting*

6.22.17 `sklearn.linear_model`.Ridge

class `sklearn.linear_model.`**Ridge**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *solver='auto'*, *random_state=None*)
Linear least squares with l2 regularization.

Minimizes the objective function:

```
||y - Xw||^2_2 + alpha * ||w||^2_2
```

This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]).

Read more in the *User Guide*.

Parameters

alpha [{float, array-like}, shape (n_targets)] Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `C^-1` in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

fit_intercept [boolean] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

max_iter [int, optional] Maximum number of iterations for conjugate gradient solver. For 'sparse_cg' and 'lsqr' solvers, the default value is determined by scipy.sparse.linalg. For 'sag' solver, the default value is 1000.

tol [float] Precision of the solution.

solver [{'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga'}] Solver to use in the computational routines:

- 'auto' chooses the solver automatically based on the type of data.
- 'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. More stable for singular matrices than 'cholesky'.
- 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.
- 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set `tol` and *max_iter*).
- 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fastest and uses an iterative procedure.
- 'sag' uses a Stochastic Average Gradient descent, and 'saga' uses its improved, unbiased version named SAGA. Both methods also use an iterative procedure, and are often faster than other solvers when both n_samples and n_features are large. Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

All last five solvers support both dense and sparse data. However, only 'sag' and 'sparse_cg' supports sparse input when `fit_intercept` is True.

New in version 0.17: Stochastic Average Gradient descent solver.

New in version 0.19: SAGA solver.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag'.

New in version 0.17: *random_state* to support Stochastic Average Gradient.

Attributes

coef_ [array, shape (n_features,) or (n_targets, n_features)] Weight vector(s).

intercept_ [float | array, shape = (n_targets,)] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

n_iter_ [array or None, shape (n_targets,)] Actual number of iterations for each target. Available only for sag and lsqr solvers. Other solvers will return None.

New in version 0.17.

See also:

RidgeClassifier Ridge classifier

RidgeCV Ridge regression with built-in cross validation

sklearn.kernel_ridge.KernelRidge Kernel ridge regression combines ridge regression with the kernel trick

Examples

```
>>> from sklearn.linear_model import Ridge
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> clf = Ridge(alpha=1.0)
>>> clf.fit(X, y)
Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001)
```

Methods

fit(self, X, y[, sample_weight])	Fit Ridge regression model
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *solver='auto'*, *random_state=None*)

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit Ridge regression model

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training data

y [array-like, shape = [n_samples] or [n_samples, n_targets]] Target values

sample_weight [float or numpy array of shape [n_samples]] Individual weights for each sample

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)
Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.linear_model.Ridge`

- *Compressive sensing: tomography reconstruction with L1 prior (Lasso)*
- *Prediction Latency*
- *Plot Ridge coefficients as a function of the regularization*
- *Ordinary Least Squares and Ridge Regression Variance*
- *Plot Ridge coefficients as a function of the L2 regularization*
- *Polynomial interpolation*
- *HuberRegressor vs Ridge on dataset with strong outliers*

6.22.18 sklearn.linear_model.RidgeClassifier

class sklearn.linear_model.**RidgeClassifier**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *class_weight=None*, *solver='auto'*, *random_state=None*)

Classifier using Ridge regression.

Read more in the *User Guide*.

Parameters

alpha [float] Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `C^-1` in other linear models such as LogisticRegression or LinearSVC.

fit_intercept [boolean] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

max_iter [int, optional] Maximum number of iterations for conjugate gradient solver. The default value is determined by scipy.sparse.linalg.

tol [float] Precision of the solution.

class_weight [dict or 'balanced', optional] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

solver [{'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga'}] Solver to use in the computational routines:

- 'auto' chooses the solver automatically based on the type of data.
- 'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. More stable for singular matrices than 'cholesky'.
- 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.
- 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set `tol` and *max_iter*).
- 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fastest and uses an iterative procedure.
- 'sag' uses a Stochastic Average Gradient descent, and 'saga' uses its unbiased and more flexible version named SAGA. Both methods use an iterative procedure, and are often faster than other solvers when both n_samples and n_features are large. Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

New in version 0.17: Stochastic Average Gradient descent solver.

New in version 0.19: SAGA solver.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag'.

Attributes

coef_ [array, shape (1, n_features) or (n_classes, n_features)] Coefficient of the features in the decision function.

`coef_` is of shape (1, n_features) when the given problem is binary.

intercept_ [float | array, shape = (n_targets,)] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

n_iter_ [array or None, shape (n_targets,)] Actual number of iterations for each target. Available only for sag and lsqr solvers. Other solvers will return None.

See also:

Ridge Ridge regression

RidgeClassifierCV Ridge classifier with built-in cross validation

Notes

For multi-class classification, n_class classifiers are trained in a one-versus-all approach. Concretely, this is implemented by taking advantage of the multi-variate response support in Ridge.

Examples

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.linear_model import RidgeClassifier
>>> X, y = load_breast_cancer(return_X_y=True)
>>> clf = RidgeClassifier().fit(X, y)
>>> clf.score(X, y)
0.9595...
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
fit(self, X, y[, sample_weight])	Fit Ridge regression model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *class_weight=None*, *solver='auto'*, *random_state=None*)

decision_function(*self*, *X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

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

Fit Ridge regression model.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples,n_features]] Training data

y [array-like, shape = [n_samples]] Target values

sample_weight [float or numpy array of shape (n_samples,)] Sample weight.

New in version 0.17: *sample_weight* support to Classifier.

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)

Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

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

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.linear_model.RidgeClassifier`

- *Classification of text documents using sparse features*

6.22.19 `sklearn.linear_model`.SGDClassifier

class `sklearn.linear_model.`**SGDClassifier**(*loss='hinge'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *n_jobs=None*, *random_state=None*, *learning_rate='optimal'*, *eta0=0.0*, *power_t=0.5*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *class_weight=None*, *warm_start=False*, *average=False*)
Linear classifiers (SVM, logistic regression, a.o.) with SGD training.

This estimator implements regularized linear models with stochastic gradient descent (SGD) learning: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate). SGD allows minibatch (online/out-of-core) learning, see the partial_fit method. For best results using the default learning rate schedule, the data should have zero mean and unit variance.

This implementation works with data represented as dense or sparse arrays of floating point values for the features. The model it fits can be controlled with the loss parameter; by default, it fits a linear support vector machine (SVM).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

Read more in the *User Guide*.

Parameters

loss [str, default: 'hinge'] The loss function to be used. Defaults to 'hinge', which gives a linear SVM.

The possible options are 'hinge', 'log', 'modified_huber', 'squared_hinge', 'perceptron', or a regression loss: 'squared_loss', 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'.

The 'log' loss gives logistic regression, a probabilistic classifier. 'modified_huber' is another smooth loss that brings tolerance to outliers as well as probability estimates. 'squared_hinge' is like hinge but is quadratically penalized. 'perceptron' is the linear loss used by the perceptron algorithm. The other losses are designed for regression but can be useful in classification as well; see SGDRegressor for a description.

penalty [str, 'none', 'l2', 'l1', or 'elasticnet'] The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' might bring sparsity to the model (feature selection) not achievable with 'l2'.

alpha [float] Constant that multiplies the regularization term. Defaults to 0.0001 Also used to compute learning_rate when set to 'optimal'.

l1_ratio [float] The Elastic Net mixing parameter, with 0 <= l1_ratio <= 1. l1_ratio=0 corresponds to L2 penalty, l1_ratio=1 to L1. Defaults to 0.15.

fit_intercept [bool] Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

max_iter [int, optional (default=1000)] The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the *`partial_fit`*.

New in version 0.19.

tol [float or None, optional (default=1e-3)] The stopping criterion. If it is not None, the iterations will stop when (loss > best_loss - tol) for `n_iter_no_change` consecutive epochs.

New in version 0.19.

shuffle [bool, optional] Whether or not the training data should be shuffled after each epoch. Defaults to True.

verbose [integer, optional] The verbosity level

epsilon [float] Epsilon in the epsilon-insensitive loss functions; only if `loss` is 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'. For 'huber', determines the threshold at which it becomes less important to get the prediction exactly right. For epsilon-insensitive, any differences between the current prediction and the correct label are ignored if they are less than this threshold.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. 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`.

learning_rate [string, optional] The learning rate schedule:

'constant': eta = eta0

'optimal': [default] eta = 1.0 / (alpha * (t + t0)) where t0 is chosen by a heuristic proposed by Leon Bottou.

'invscaling': eta = eta0 / pow(t, power_t)

'adaptive': eta = eta0, as long as the training keeps decreasing. Each time n_iter_no_change consecutive epochs fail to decrease the training loss by tol or fail to

increase validation score by tol if early_stopping is True, the current learning rate is divided by 5.

eta0 [double] The initial learning rate for the 'constant', 'invscaling' or 'adaptive' schedules. The default value is 0.0 as eta0 is not used by the default schedule 'optimal'.

power_t [double] The exponent for inverse scaling learning rate [default 0.5].

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation score is not improving. If set to True, it will automatically set aside a stratified fraction of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.

New in version 0.20.

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change [int, default=5] Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

class_weight [dict, {class_label: weight} or "balanced" or None, optional] Preset for the class_weight fit parameter.

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

Repeatedly calling fit or partial_fit when warm_start is True can result in a different solution than when calling fit a single time because of the way the data is shuffled. If a dynamic learning rate is used, the learning rate is adapted depending on the number of samples already seen. Calling `fit` resets this counter, while `partial_fit` will result in increasing the existing counter.

average [bool or int, optional] When set to True, computes the averaged SGD weights and stores the result in the `coef_` attribute. If set to an int greater than 1, averaging will begin once the total number of samples seen reaches average. So `average=10` will begin averaging after seeing 10 samples.

Attributes

coef_ [array, shape (1, n_features) if n_classes == 2 else (n_classes, n_features)] Weights assigned to the features.

intercept_ [array, shape (1,) if n_classes == 2 else (n_classes,)] Constants in decision function.

n_iter_ [int] The actual number of iterations to reach the stopping criterion. For multiclass fits, it is the maximum over every binary fit.

loss_function_ [concrete `LossFunction`]

See also:

sklearn.svm.LinearSVC, ***LogisticRegression***, ***Perceptron***

Examples

```
>>> import numpy as np
>>> from sklearn import linear_model
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> Y = np.array([1, 1, 2, 2])
>>> clf = linear_model.SGDClassifier(max_iter=1000, tol=1e-3)
>>> clf.fit(X, Y)
...
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge', max_iter=1000,
       n_iter_no_change=5, n_jobs=None, penalty='l2', power_t=0.5,
       random_state=None, shuffle=True, tol=0.001, validation_fraction=0.1,
       verbose=0, warm_start=False)
```

```
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, coef_init, intercept_init, ...])	Fit linear model with Stochastic Gradient Descent.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Perform one epoch of stochastic gradient descent on given samples.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, *args, **kwargs)	
sparsify(self)	Convert coefficient matrix to sparse format.

__init__(*self*, *loss='hinge'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *n_jobs=None*, *random_state=None*, *learning_rate='optimal'*, *eta0=0.0*, *power_t=0.5*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *class_weight=None*, *warm_start=False*, *average=False*)

decision_function(*self*, *X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify(*self*)
Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self [estimator]

fit(*self*, *X*, *y*, *coef_init=None*, *intercept_init=None*, *sample_weight=None*)
Fit linear model with Stochastic Gradient Descent.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data

y [numpy array, shape (n_samples,)] Target values

coef_init [array, shape (n_classes, n_features)] The initial coefficients to warm-start the optimization.

intercept_init [array, shape (n_classes,)] The initial intercept to warm-start the optimization.

sample_weight [array-like, shape (n_samples,), optional] Weights applied to individual samples. If not provided, uniform weights are assumed. These weights will be multiplied with class_weight (passed through the constructor) if class_weight is specified

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Perform one epoch of stochastic gradient descent on given samples.

Internally, this method uses `max_iter = 1`. Therefore, it is not guaranteed that a minimum of the cost function is reached after calling it once. Matters such as objective convergence and early stopping should be handled by the user.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Subset of the training data

y [numpy array, shape (n_samples,)] Subset of the target values

classes [array, shape (n_classes,)] Classes across all calls to partial_fit. Can be obtained by via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in `classes`.

sample_weight [array-like, shape (n_samples,), optional] Weights applied to individual samples. If not provided, uniform weights are assumed.

Returns

self [returns an instance of self.]

predict (*self*, *X*)
Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

predict_log_proba
Log of probability estimates.

This method is only available for log loss and modified Huber loss.

When loss="modified_huber", probability estimates may be hard zeros and ones, so taking the logarithm is not possible.

See `predict_proba` for details.

Parameters

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

Returns

T [array-like, shape (n_samples, n_classes)] Returns the log-probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

predict_proba
Probability estimates.

This method is only available for log loss and modified Huber loss.

Multiclass probability estimates are derived from binary (one-vs.-rest) estimates by simple normalization, as recommended by Zadrozny and Elkan.

Binary probability estimates for loss="modified_huber" are given by (clip(decision_function(X), -1, 1) + 1) / 2. For other loss functions it is necessary to perform proper probability calibration by wrapping the classifier with *sklearn.calibration.CalibratedClassifierCV* instead.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)]

Returns

array, shape (n_samples, n_classes) Returns the probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

References

Zadrozny and Elkan, "Transforming classifier scores into multiclass probability estimates", SIGKDD'02, http://www.research.ibm.com/people/z/zadrozny/kdd2002-Transf.pdf

The justification for the formula in the loss="modified_huber" case is in the appendix B in: http://jmlr.csail.mit.edu/papers/volume2/zhang02c/zhang02c.pdf

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

sparsify(*self*)
Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self [estimator]

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.linear_model.SGDClassifier`

- *Model Complexity Influence*
- *Out-of-core classification of text documents*
- *Pipelining: chaining a PCA and a logistic regression*
- *SGD: Maximum margin separating hyperplane*
- *SGD: Weighted samples*
- *Comparing various online solvers*
- *Plot multi-class SGD on the iris dataset*
- *Early stopping of Stochastic Gradient Descent*
- *Sample pipeline for text feature extraction and evaluation*
- *Classification of text documents using sparse features*

6.22.20 sklearn.linear_model.SGDRegressor

class sklearn.linear_model.**SGDRegressor**(*loss='squared_loss'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *random_state=None*, *learning_rate='invscaling'*, *eta0=0.01*, *power_t=0.25*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *warm_start=False*, *average=False*)

Linear model fitted by minimizing a regularized empirical loss with SGD

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense numpy arrays of floating point values for the features.

Read more in the *User Guide*.

Parameters

loss [str, default: 'squared_loss'] The loss function to be used. The possible values are 'squared_loss', 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'

The 'squared_loss' refers to the ordinary least squares fit. 'huber' modifies 'squared_loss' to focus less on getting outliers correct by switching from squared to linear loss past a distance of epsilon. 'epsilon_insensitive' ignores errors less than epsilon and is linear past that; this is the loss function used in SVR. 'squared_epsilon_insensitive' is the same but becomes squared loss past a tolerance of epsilon.

penalty [str, 'none', 'l2', 'l1', or 'elasticnet'] The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' might bring sparsity to the model (feature selection) not achievable with 'l2'.

alpha [float] Constant that multiplies the regularization term. Defaults to 0.0001 Also used to compute learning_rate when set to 'optimal'.

l1_ratio [float] The Elastic Net mixing parameter, with 0 <= l1_ratio <= 1. l1_ratio=0 corresponds to L2 penalty, l1_ratio=1 to L1. Defaults to 0.15.

fit_intercept [bool] Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. Defaults to True.

max_iter [int, optional (default=1000)] The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the *partial_fit*.

New in version 0.19.

tol [float or None, optional (default=1e-3)] The stopping criterion. If it is not None, the iterations will stop when (loss > best_loss - tol) for `n_iter_no_change` consecutive epochs.

New in version 0.19.

shuffle [bool, optional] Whether or not the training data should be shuffled after each epoch. Defaults to True.

verbose [integer, optional] The verbosity level.

epsilon [float] Epsilon in the epsilon-insensitive loss functions; only if `loss` is 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'. For 'huber', determines the threshold at which it becomes less important to get the prediction exactly right. For epsilon-insensitive, any differences between the current prediction and the correct label are ignored if they are less than this threshold.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. 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`.

learning_rate [string, optional] The learning rate schedule:

'constant': eta = eta0

'optimal': eta = 1.0 / (alpha * (t + t0)) where t0 is chosen by a heuristic proposed by Leon Bottou.

'invscaling': [default] eta = eta0 / pow(t, power_t)

'adaptive': eta = eta0, as long as the training keeps decreasing. Each time n_iter_no_change consecutive epochs fail to decrease the training loss by tol or fail to increase validation score by tol if early_stopping is True, the current learning rate is divided by 5.

eta0 [double] The initial learning rate for the 'constant', 'invscaling' or 'adaptive' schedules. The default value is 0.01.

power_t [double] The exponent for inverse scaling learning rate [default 0.5].

early_stopping [bool, default=False] Whether to use early stopping to terminate training when validation score is not improving. If set to True, it will automatically set aside a fraction of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.

New in version 0.20.

validation_fraction [float, default=0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change [int, default=5] Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

Repeatedly calling fit or partial_fit when warm_start is True can result in a different solution than when calling fit a single time because of the way the data is shuffled. If a dynamic learning rate is used, the learning rate is adapted depending on the number of samples already seen. Calling `fit` resets this counter, while `partial_fit` will result in increasing the existing counter.

average [bool or int, optional] When set to True, computes the averaged SGD weights and stores the result in the `coef_` attribute. If set to an int greater than 1, averaging will begin once the total number of samples seen reaches average. So `average=10` will begin averaging after seeing 10 samples.

Attributes

coef_ [array, shape (n_features,)] Weights assigned to the features.

intercept_ [array, shape (1,)] The intercept term.

average_coef_ [array, shape (n_features,)] Averaged weights assigned to the features.

average_intercept_ [array, shape (1,)] The averaged intercept term.

n_iter_ [int] The actual number of iterations to reach the stopping criterion.

See also:

Ridge, *ElasticNet*, *Lasso*, *sklearn.svm.SVR*

Examples

```
>>> import numpy as np
>>> from sklearn import linear_model
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> clf = linear_model.SGDRegressor(max_iter=1000, tol=1e-3)
>>> clf.fit(X, y)
...
SGDRegressor(alpha=0.0001, average=False, early_stopping=False,
       epsilon=0.1, eta0=0.01, fit_intercept=True, l1_ratio=0.15,
       learning_rate='invscaling', loss='squared_loss', max_iter=1000,
       n_iter_no_change=5, penalty='l2', power_t=0.25, random_state=None,
       shuffle=True, tol=0.001, validation_fraction=0.1, verbose=0,
       warm_start=False)
```

Methods

densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, coef_init, intercept_init, ...])	Fit linear model with Stochastic Gradient Descent.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, sample_weight])	Perform one epoch of stochastic gradient descent on given samples.
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, *args, **kwargs)	
sparsify(self)	Convert coefficient matrix to sparse format.

__init__(*self*, *loss='squared_loss'*, *penalty='l2'*, *alpha=0.0001*, *l1_ratio=0.15*, *fit_intercept=True*, *max_iter=1000*, *tol=0.001*, *shuffle=True*, *verbose=0*, *epsilon=0.1*, *random_state=None*, *learning_rate='invscaling'*, *eta0=0.01*, *power_t=0.25*, *early_stopping=False*, *validation_fraction=0.1*, *n_iter_no_change=5*, *warm_start=False*, *average=False*)

densify(*self*)

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified;

otherwise, it is a no-op.

Returns

self [estimator]

fit (*self*, *X*, *y*, *coef_init=None*, *intercept_init=None*, *sample_weight=None*)
Fit linear model with Stochastic Gradient Descent.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data

y [numpy array, shape (n_samples,)] Target values

coef_init [array, shape (n_features,)] The initial coefficients to warm-start the optimization.

intercept_init [array, shape (1,)] The initial intercept to warm-start the optimization.

sample_weight [array-like, shape (n_samples,), optional] Weights applied to individual samples (1. for unweighted).

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit (*self*, *X*, *y*, *sample_weight=None*)
Perform one epoch of stochastic gradient descent on given samples.

Internally, this method uses `max_iter = 1`. Therefore, it is not guaranteed that a minimum of the cost function is reached after calling it once. Matters such as objective convergence and early stopping should be handled by the user.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Subset of training data

y [numpy array of shape (n_samples,)] Subset of target values

sample_weight [array-like, shape (n_samples,), optional] Weights applied to individual samples. If not provided, uniform weights are assumed.

Returns

self [returns an instance of self.]

predict (*self*, *X*)
Predict using the linear model

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)]

Returns

array, shape (n_samples,) Predicted target values per element in X.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

sparsify(*self*)

Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self [estimator]

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.linear_model.SGDRegressor`

- *Prediction Latency*

6.22.21 sklearn.linear_model.TheilSenRegressor

class sklearn.linear_model.**TheilSenRegressor**(*fit_intercept=True*, *copy_X=True*, *max_subpopulation=10000.0*, *n_subsamples=None*, *max_iter=300*, *tol=0.001*, *random_state=None*, *n_jobs=None*, *verbose=False*)

Theil-Sen Estimator: robust multivariate regression model.

The algorithm calculates least square solutions on subsets with size n_subsamples of the samples in X. Any value of n_subsamples between the number of features and samples leads to an estimator with a compromise between robustness and efficiency. Since the number of least square solutions is "n_samples choose n_subsamples", it can be extremely large and can therefore be limited with max_subpopulation. If this limit is reached, the subsets are chosen randomly. In a final step, the spatial median (or L1 median) is calculated of all least square solutions.

Read more in the *User Guide*.

Parameters

fit_intercept [boolean, optional, default True] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

max_subpopulation [int, optional, default 1e4] Instead of computing with a set of cardinality 'n choose k', where n is the number of samples and k is the number of subsamples (at least number of features), consider only a stochastic subpopulation of a given maximal size if 'n choose k' is larger than max_subpopulation. For other than small problem sizes this parameter will determine memory usage and runtime if n_subsamples is not changed.

n_subsamples [int, optional, default None] Number of samples to calculate the parameters. This is at least the number of features (plus 1 if fit_intercept=True) and the number of samples as a maximum. A lower number leads to a higher breakdown point and a low efficiency while a high number leads to a low breakdown point and a high efficiency. If None, take the minimum number of subsamples leading to maximal robustness. If n_subsamples is set to n_samples, Theil-Sen is identical to least squares.

max_iter [int, optional, default 300] Maximum number of iterations for the calculation of spatial median.

tol [float, optional, default 1.e-3] Tolerance when calculating spatial median.

random_state [int, RandomState instance or None, optional, default None] A random number generator instance to define the state of the random permutations generator. 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`.

n_jobs [int or None, optional (default=None)] Number of CPUs to use during the cross validation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [boolean, optional, default False] Verbose mode when fitting the model.

Attributes

coef_ [array, shape = (n_features)] Coefficients of the regression model (median of distribution).

intercept_ [float] Estimated intercept of regression model.

breakdown_ [float] Approximated breakdown point.

n_iter_ [int] Number of iterations needed for the spatial median.

n_subpopulation_ [int] Number of combinations taken into account from 'n choose k', where n is the number of samples and k is the number of subsamples.

References

- Theil-Sen Estimators in a Multiple Linear Regression Model, 2009 Xin Dang, Hanxiang Peng, Xueqin Wang and Heping Zhang http://home.olemiss.edu/~xdang/papers/MTSE.pdf

Examples

```
>>> from sklearn.linear_model import TheilSenRegressor
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(
...     n_samples=200, n_features=2, noise=4.0, random_state=0)
>>> reg = TheilSenRegressor(random_state=0).fit(X, y)
>>> reg.score(X, y)
0.9884...
>>> reg.predict(X[:1,])
array([-31.5871...])
```

Methods

fit(self, X, y)	Fit linear model.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *fit_intercept=True*, *copy_X=True*, *max_subpopulation=10000.0*, *n_subsamples=None*, *max_iter=300*, *tol=0.001*, *random_state=None*, *n_jobs=None*, *verbose=False*)

fit (*self*, *X*, *y*)
Fit linear model.

Parameters

X [numpy array of shape [n_samples, n_features]] Training data

y [numpy array of shape [n_samples]] Target values

Returns

self [returns an instance of self.]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.linear_model.TheilSenRegressor`

- *Theil-Sen Regression*
- *Robust linear estimator fitting*

linear_model.enet_path(X, y[, l1_ratio, ...])	Compute elastic net path with coordinate descent
linear_model.lars_path(X, y[, Xy, Gram, ...])	Compute Least Angle Regression or Lasso path using LARS algorithm [1]
linear_model.lars_path_gram(Xy, Gram, n_samples)	lars_path in the sufficient stats mode [1]
linear_model.lasso_path(X, y[, eps, ...])	Compute Lasso path with coordinate descent
linear_model.orthogonal_mp(X, y[, ...])	Orthogonal Matching Pursuit (OMP)
linear_model.orthogonal_mp_gram(Gram, Xy[, ...])	Gram Orthogonal Matching Pursuit (OMP)
linear_model.ridge_regression(X, y, alpha[, ...])	Solve the ridge equation by the method of normal equations.

6.22.22 sklearn.linear_model.enet_path

sklearn.linear_model.**enet_path**(*X*, *y*, *l1_ratio=0.5*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, *check_input=True*, ***params*)

Compute elastic net path with coordinate descent

The elastic net optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
1 / (2 * n_samples) * ||y - Xw||^2_2
+ alpha * l1_ratio * ||w||_1
+ 0.5 * alpha * (1 - l1_ratio) * ||w||^2_2
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^Fro_2
+ alpha * l1_ratio * ||W||_21
+ 0.5 * alpha * (1 - l1_ratio) * ||W||_Fro^2
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,) or (n_samples, n_outputs)] Target values

l1_ratio [float, optional] float between 0 and 1 passed to elastic net (scaling between l1 and l2 penalties). `l1_ratio=1` corresponds to the Lasso

eps [float] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If None alphas are set automatically

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

check_input [bool, default True] Skip input validation checks, including the Gram matrix when provided assuming there are handled by the caller when check_input=False.

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha. (Is returned when `return_n_iter` is set to True).

See also:

MultiTaskElasticNet

MultiTaskElasticNetCV

ElasticNet

ElasticNetCV

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

Examples using `sklearn.linear_model.enet_path`

- *Lasso and Elastic Net*

6.22.23 sklearn.linear_model.lars_path

sklearn.linear_model.**lars_path**(*X*, *y*, *Xy=None*, *Gram=None*, *max_iter=500*, *alpha_min=0*, *method='lar'*, *copy_X=True*, *eps=2.220446049250313e-16*, *copy_Gram=True*, *verbose=0*, *return_path=True*, *return_n_iter=False*, *positive=False*)

Compute Least Angle Regression or Lasso path using LARS algorithm [1]

The optimization objective for the case method='lasso' is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

in the case of method='lars', the objective function is only known in the form of an implicit equation (see discussion in [1])

Read more in the *User Guide*.

Parameters

X [None or array, shape (n_samples, n_features)] Input data. Note that if X is None then the Gram matrix must be specified, i.e., cannot be None or False.

Deprecated since version 0.21: The use of `X` is `None` in combination with `Gram` is not `None` will be removed in v0.23. Use *lars_path_gram* instead.

y [None or array, shape (n_samples,)] Input targets.

Xy [array-like, shape (n_samples,) or (n_samples, n_targets), optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

Gram [None, 'auto', array, shape (n_features, n_features), optional] Precomputed Gram matrix (X' * X), if `'auto'`, the Gram matrix is precomputed from the given X, if there are more samples than features.

Deprecated since version 0.21: The use of `X` is `None` in combination with `Gram` is not None will be removed in v0.23. Use *lars_path_gram* instead.

max_iter [integer, optional (default=500)] Maximum number of iterations to perform, set to infinity for no limit.

alpha_min [float, optional (default=0)] Minimum correlation along the path. It corresponds to the regularization parameter alpha parameter in the Lasso.

method [{'lar', 'lasso'}, optional (default='lar')] Specifies the returned model. Select `'lar'` for Least Angle Regression, `'lasso'` for the Lasso.

copy_X [bool, optional (default=True)] If `False`, `X` is overwritten.

eps [float, optional (default=''np.finfo(np.float).eps'')] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

copy_Gram [bool, optional (default=True)] If `False`, `Gram` is overwritten.

verbose [int (default=0)] Controls output verbosity.

return_path [bool, optional (default=True)] If `return_path==True` returns the entire path, else returns only the last point of the path.

return_n_iter [bool, optional (default=False)] Whether to return the number of iterations.

positive [boolean (default=False)] Restrict coefficients to be >= 0. This option is only allowed with method 'lasso'. Note that the model coefficients will not converge to the ordinary-least-squares solution for small values of alpha. Only coefficients up to the smallest alpha

value (`alphas_[alphas_ > 0.].min()` when fit_path=True) reached by the stepwise Lars-Lasso algorithm are typically in congruence with the solution of the coordinate descent lasso_path function.

Returns

alphas [array, shape (n_alphas + 1,)] Maximum of covariances (in absolute value) at each iteration. `n_alphas` is either `max_iter`, `n_features` or the number of nodes in the path with `alpha >= alpha_min`, whichever is smaller.

active [array, shape [n_alphas]] Indices of active variables at the end of the path.

coefs [array, shape (n_features, n_alphas + 1)] Coefficients along the path

n_iter [int] Number of iterations run. Returned only if return_n_iter is set to True.

See also:

lars_path_gram

lasso_path

lasso_path_gram

LassoLars

Lars

LassoLarsCV

LarsCV

sklearn.decomposition.sparse_encode

References

[1], *[2]*, *[3]*

Examples using `sklearn.linear_model.lars_path`

- *Lasso path using LARS*

6.22.24 `sklearn.linear_model`.lars_path_gram

`sklearn.linear_model.`**`lars_path_gram`**(*Xy*, *Gram*, *n_samples*, *max_iter=500*, *alpha_min=0*, *method='lar'*, *copy_X=True*, *eps=2.220446049250313e-16*, *copy_Gram=True*, *verbose=0*, *return_path=True*, *return_n_iter=False*, *positive=False*)

lars_path in the sufficient stats mode [1]

The optimization objective for the case method='lasso' is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

in the case of method='lars', the objective function is only known in the form of an implicit equation (see discussion in [1])

Read more in the *User Guide*.

Parameters

Xy [array-like, shape (n_samples,) or (n_samples, n_targets)] Xy = np.dot(X.T, y).

Gram [array, shape (n_features, n_features)] Gram = np.dot(X.T * X).

n_samples [integer or float] Equivalent size of sample.

max_iter [integer, optional (default=500)] Maximum number of iterations to perform, set to infinity for no limit.

alpha_min [float, optional (default=0)] Minimum correlation along the path. It corresponds to the regularization parameter alpha parameter in the Lasso.

method [{'lar', 'lasso'}, optional (default='lar')] Specifies the returned model. Select `'lar'` for Least Angle Regression, `'lasso'` for the Lasso.

copy_X [bool, optional (default=True)] If `False`, `X` is overwritten.

eps [float, optional (default=''np.finfo(np.float).eps'')] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

copy_Gram [bool, optional (default=True)] If `False`, `Gram` is overwritten.

verbose [int (default=0)] Controls output verbosity.

return_path [bool, optional (default=True)] If `return_path==True` returns the entire path, else returns only the last point of the path.

return_n_iter [bool, optional (default=False)] Whether to return the number of iterations.

positive [boolean (default=False)] Restrict coefficients to be >= 0. This option is only allowed with method 'lasso'. Note that the model coefficients will not converge to the ordinary-least-squares solution for small values of alpha. Only coefficients up to the smallest alpha value (`alphas_[alphas_ > 0.].min()` when fit_path=True) reached by the stepwise Lars-Lasso algorithm are typically in congruence with the solution of the coordinate descent lasso_path function.

Returns

alphas [array, shape (n_alphas + 1,)] Maximum of covariances (in absolute value) at each iteration. `n_alphas` is either `max_iter`, `n_features` or the number of nodes in the path with `alpha >= alpha_min`, whichever is smaller.

active [array, shape [n_alphas]] Indices of active variables at the end of the path.

coefs [array, shape (n_features, n_alphas + 1)] Coefficients along the path

n_iter [int] Number of iterations run. Returned only if return_n_iter is set to True.

See also:

lars_path

lasso_path

lasso_path_gram

LassoLars

Lars

LassoLarsCV

LarsCV

sklearn.decomposition.sparse_encode

References

[1], *[2]*, *[3]*

6.22.25 sklearn.linear_model.lasso_path

sklearn.linear_model.**lasso_path**(*X*, *y*, *eps=0.001*, *n_alphas=100*, *alphas=None*, *precompute='auto'*, *Xy=None*, *copy_X=True*, *coef_init=None*, *verbose=False*, *return_n_iter=False*, *positive=False*, ***params*)

Compute Lasso path with coordinate descent

The Lasso optimization function varies for mono and multi-outputs.

For mono-output tasks it is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

For multi-output tasks it is:

```
(1 / (2 * n_samples)) * ||Y - XW||^2_Fro + alpha * ||W||_21
```

Where:

```
||W||_21 = \sum_i \sqrt{\sum_j w_{ij}^2}
```

i.e. the sum of norm of each row.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training data. Pass directly as Fortran-contiguous data to avoid unnecessary memory duplication. If `y` is mono-output then `X` can be sparse.

y [ndarray, shape (n_samples,), or (n_samples, n_outputs)] Target values

eps [float, optional] Length of the path. `eps=1e-3` means that `alpha_min / alpha_max = 1e-3`

n_alphas [int, optional] Number of alphas along the regularization path

alphas [ndarray, optional] List of alphas where to compute the models. If `None` alphas are set automatically

precompute [True | False | 'auto' | array-like] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument.

Xy [array-like, optional] Xy = np.dot(X.T, y) that can be precomputed. It is useful only when the Gram matrix is precomputed.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

coef_init [array, shape (n_features,) | None] The initial values of the coefficients.

verbose [bool or integer] Amount of verbosity.

return_n_iter [bool] whether to return the number of iterations or not.

positive [bool, default False] If set to True, forces coefficients to be positive. (Only allowed when `y.ndim == 1`).

****params** [kwargs] keyword arguments passed to the coordinate descent solver.

Returns

alphas [array, shape (n_alphas,)] The alphas along the path where models are computed.

coefs [array, shape (n_features, n_alphas) or (n_outputs, n_features, n_alphas)] Coefficients along the path.

dual_gaps [array, shape (n_alphas,)] The dual gaps at the end of the optimization for each alpha.

n_iters [array-like, shape (n_alphas,)] The number of iterations taken by the coordinate descent optimizer to reach the specified tolerance for each alpha.

See also:

lars_path

Lasso

LassoLars

LassoCV

LassoLarsCV

sklearn.decomposition.sparse_encode

Notes

For an example, see *examples/linear_model/plot_lasso_coordinate_descent_path.py*.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Note that in certain cases, the Lars solver may be significantly faster to implement this functionality. In particular, linear interpolation can be used to retrieve model coefficients between the values output by lars_path

Examples

Comparing lasso_path and lars_path with interpolation:

```
>>> X = np.array([[1, 2, 3.1], [2.3, 5.4, 4.3]]).T
>>> y = np.array([1, 2, 3.1])
>>> # Use lasso_path to compute a coefficient path
>>> _, coef_path, _ = lasso_path(X, y, alphas=[5., 1., .5])
>>> print(coef_path)
[[0.         0.         0.46874778]
 [0.2159048  0.4425765  0.23689075]]
```

```
>>> # Now use lars_path and 1D linear interpolation to compute the
>>> # same path
>>> from sklearn.linear_model import lars_path
>>> alphas, active, coef_path_lars = lars_path(X, y, method='lasso')
>>> from scipy import interpolate
>>> coef_path_continuous = interpolate.interp1d(alphas[::-1],
```

```
...                                                 coef_path_lars[:, ::-1])
>>> print(coef_path_continuous([5., 1., .5]))
[[0.          0.          0.46915237]
 [0.2159048   0.4425765   0.23668876]]
```

Examples using `sklearn.linear_model.lasso_path`

- *Lasso and Elastic Net*

6.22.26 `sklearn.linear_model`.orthogonal_mp

`sklearn.linear_model.`**`orthogonal_mp`**(*X*, *y*, *n_nonzero_coefs=None*, *tol=None*, *precompute=False*, *copy_X=True*, *return_path=False*, *return_n_iter=False*)

Orthogonal Matching Pursuit (OMP)

Solves n_targets Orthogonal Matching Pursuit problems. An instance of the problem has the form:

When parametrized by the number of non-zero coefficients using `n_nonzero_coefs`: argmin ||y - Xgamma||^2 subject to ||gamma||_0 <= n_{nonzero coefs}

When parametrized by error using the parameter `tol`: argmin ||gamma||_0 subject to ||y - Xgamma||^2 <= tol

Read more in the *User Guide*.

Parameters

X [array, shape (n_samples, n_features)] Input data. Columns are assumed to have unit norm.

y [array, shape (n_samples,) or (n_samples, n_targets)] Input targets

n_nonzero_coefs [int] Desired number of non-zero entries in the solution. If None (by default) this value is set to 10% of n_features.

tol [float] Maximum norm of the residual. If not None, overrides n_nonzero_coefs.

precompute [{True, False, 'auto'},] Whether to perform precomputations. Improves performance when n_targets or n_samples is very large.

copy_X [bool, optional] Whether the design matrix X must be copied by the algorithm. A false value is only helpful if X is already Fortran-ordered, otherwise a copy is made anyway.

return_path [bool, optional. Default: False] Whether to return every value of the nonzero coefficients along the forward path. Useful for cross-validation.

return_n_iter [bool, optional default False] Whether or not to return the number of iterations.

Returns

coef [array, shape (n_features,) or (n_features, n_targets)] Coefficients of the OMP solution. If `return_path=True`, this contains the whole coefficient path. In this case its shape is (n_features, n_features) or (n_features, n_targets, n_features) and iterating over the last axis yields coefficients in increasing order of active features.

n_iters [array-like or int] Number of active features across every target. Returned only if `return_n_iter` is set to True.

See also:

`OrthogonalMatchingPursuit`

orthogonal_mp_gram

lars_path

decomposition.sparse_encode

Notes

Orthogonal matching pursuit was introduced in S. Mallat, Z. Zhang, Matching pursuits with time-frequency dictionaries, IEEE Transactions on Signal Processing, Vol. 41, No. 12. (December 1993), pp. 3397-3415. (http://blanche.polytechnique.fr/~mallat/papiers/MallatPursuit93.pdf)

This implementation is based on Rubinstein, R., Zibulevsky, M. and Elad, M., Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit Technical Report - CS Technion, April 2008. https://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

6.22.27 sklearn.linear_model.orthogonal_mp_gram

sklearn.linear_model.**orthogonal_mp_gram**(*Gram*, *Xy*, *n_nonzero_coefs=None*, *tol=None*, *norms_squared=None*, *copy_Gram=True*, *copy_Xy=True*, *return_path=False*, *return_n_iter=False*)

Gram Orthogonal Matching Pursuit (OMP)

Solves n_targets Orthogonal Matching Pursuit problems using only the Gram matrix X.T * X and the product X.T * y.

Read more in the *User Guide*.

Parameters

Gram [array, shape (n_features, n_features)] Gram matrix of the input data: X.T * X

Xy [array, shape (n_features,) or (n_features, n_targets)] Input targets multiplied by X: X.T * y

n_nonzero_coefs [int] Desired number of non-zero entries in the solution. If None (by default) this value is set to 10% of n_features.

tol [float] Maximum norm of the residual. If not None, overrides n_nonzero_coefs.

norms_squared [array-like, shape (n_targets,)] Squared L2 norms of the lines of y. Required if tol is not None.

copy_Gram [bool, optional] Whether the gram matrix must be copied by the algorithm. A false value is only helpful if it is already Fortran-ordered, otherwise a copy is made anyway.

copy_Xy [bool, optional] Whether the covariance vector Xy must be copied by the algorithm. If False, it may be overwritten.

return_path [bool, optional. Default: False] Whether to return every value of the nonzero coefficients along the forward path. Useful for cross-validation.

return_n_iter [bool, optional default False] Whether or not to return the number of iterations.

Returns

coef [array, shape (n_features,) or (n_features, n_targets)] Coefficients of the OMP solution. If `return_path=True`, this contains the whole coefficient path. In this case its shape is (n_features, n_features) or (n_features, n_targets, n_features) and iterating over the last axis yields coefficients in increasing order of active features.

n_iters [array-like or int] Number of active features across every target. Returned only if `return_n_iter` is set to True.

See also:

OrthogonalMatchingPursuit

orthogonal_mp

lars_path

decomposition.sparse_encode

Notes

Orthogonal matching pursuit was introduced in G. Mallat, Z. Zhang, Matching pursuits with time-frequency dictionaries, IEEE Transactions on Signal Processing, Vol. 41, No. 12. (December 1993), pp. 3397-3415. (http://blanche.polytechnique.fr/~mallat/papiers/MallatPursuit93.pdf)

This implementation is based on Rubinstein, R., Zibulevsky, M. and Elad, M., Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit Technical Report - CS Technion, April 2008. https://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

6.22.28 sklearn.linear_model.ridge_regression

sklearn.linear_model.**ridge_regression**(*X*, *y*, *alpha*, *sample_weight=None*, *solver='auto'*, *max_iter=None*, *tol=0.001*, *verbose=0*, *random_state=None*, *return_n_iter=False*, *return_intercept=False*, *check_input=True*)

Solve the ridge equation by the method of normal equations.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix, LinearOperator},] shape = [n_samples, n_features] Training data

y [array-like, shape = [n_samples] or [n_samples, n_targets]] Target values

alpha [{float, array-like},] shape = [n_targets] if array-like Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `C^-1` in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

sample_weight [float or numpy array of shape [n_samples]] Individual weights for each sample. If sample_weight is not None and solver='auto', the solver will be set to 'cholesky'.

New in version 0.17.

solver [{'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga'}] Solver to use in the computational routines:

- 'auto' chooses the solver automatically based on the type of data.
- 'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. More stable for singular matrices than 'cholesky'.

- 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution via a Cholesky decomposition of dot(X.T, X)
- 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set `tol` and *max_iter*).
- 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fastest and uses an iterative procedure.
- 'sag' uses a Stochastic Average Gradient descent, and 'saga' uses its improved, unbiased version named SAGA. Both methods also use an iterative procedure, and are often faster than other solvers when both n_samples and n_features are large. Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

All last five solvers support both dense and sparse data. However, only 'sag' and 'sparse_cg' supports sparse input when'fit_intercept' is True.

New in version 0.17: Stochastic Average Gradient descent solver.

New in version 0.19: SAGA solver.

max_iter [int, optional] Maximum number of iterations for conjugate gradient solver. For the 'sparse_cg' and 'lsqr' solvers, the default value is determined by scipy.sparse.linalg. For 'sag' and saga solver, the default value is 1000.

tol [float] Precision of the solution.

verbose [int] Verbosity level. Setting verbose > 0 will display additional information depending on the solver used.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag'.

return_n_iter [boolean, default False] If True, the method also returns `n_iter`, the actual number of iteration performed by the solver.

New in version 0.17.

return_intercept [boolean, default False] If True and if X is sparse, the method also returns the intercept, and the solver is automatically changed to 'sag'. This is only a temporary fix for fitting the intercept with sparse data. For dense data, use sklearn.linear_model._preprocess_data before your regression.

New in version 0.17.

check_input [boolean, default True] If False, the input arrays X and y will not be checked.

New in version 0.21.

Returns

coef [array, shape = [n_features] or [n_targets, n_features]] Weight vector(s).

n_iter [int, optional] The actual number of iteration performed by the solver. Only returned if `return_n_iter` is True.

intercept [float or array, shape = [n_targets]] The intercept of the model. Only returned if `return_intercept` is True and if X is a scipy sparse array.

Notes

This function won't compute the intercept.

6.23 sklearn.manifold: Manifold Learning

The *sklearn.manifold* module implements data embedding techniques.

User guide: See the *Manifold learning* section for further details.

manifold.Isomap([n_neighbors, n_components, ...])	Isomap Embedding
manifold.LocallyLinearEmbedding([...])	Locally Linear Embedding
manifold.MDS([n_components, metric, n_init, ...])	Multidimensional scaling
manifold.SpectralEmbedding([n_components, ...])	Spectral embedding for non-linear dimensionality reduction.
manifold.TSNE([n_components, perplexity, ...])	t-distributed Stochastic Neighbor Embedding.

6.23.1 sklearn.manifold.Isomap

class sklearn.manifold.**Isomap**(*n_neighbors=5*, *n_components=2*, *eigen_solver='auto'*, *tol=0*, *max_iter=None*, *path_method='auto'*, *neighbors_algorithm='auto'*, *n_jobs=None*)

Isomap Embedding

Non-linear dimensionality reduction through Isometric Mapping

Read more in the *User Guide*.

Parameters

n_neighbors [integer] number of neighbors to consider for each point.

n_components [integer] number of coordinates for the manifold

eigen_solver [['auto'|'arpack'|'dense']] 'auto' : Attempt to choose the most efficient solver for the given problem.

'arpack' : Use Arnoldi decomposition to find the eigenvalues and eigenvectors.

'dense' : Use a direct solver (i.e. LAPACK) for the eigenvalue decomposition.

tol [float] Convergence tolerance passed to arpack or lobpcg. not used if eigen_solver == 'dense'.

max_iter [integer] Maximum number of iterations for the arpack solver. not used if eigen_solver == 'dense'.

path_method [string ['auto'|'FW'|'D']] Method to use in finding shortest path.

'auto' : attempt to choose the best algorithm automatically.

'FW' : Floyd-Warshall algorithm.

'D' : Dijkstra's algorithm.

neighbors_algorithm [string ['auto'|'brute'|'kd_tree'|'ball_tree']] Algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance.

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

Attributes

embedding_ [array-like, shape (n_samples, n_components)] Stores the embedding vectors.

kernel_pca_ [object] *KernelPCA* object used to implement the embedding.

training_data_ [array-like, shape (n_samples, n_features)] Stores the training data.

nbrs_ [sklearn.neighbors.NearestNeighbors instance] Stores nearest neighbors instance, including BallTree or KDtree if applicable.

dist_matrix_ [array-like, shape (n_samples, n_samples)] Stores the geodesic distance matrix of training data.

References

[R7f4d308f5054-1]

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import Isomap
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = Isomap(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
```

Methods

fit(self, X[, y])	Compute the embedding vectors for data X
fit_transform(self, X[, y])	Fit the model from data in X and transform X.
get_params(self[, deep])	Get parameters for this estimator.
reconstruction_error(self)	Compute the reconstruction error for the embedding.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X.

__init__(*self*, *n_neighbors=5*, *n_components=2*, *eigen_solver='auto'*, *tol=0*, *max_iter=None*, *path_method='auto'*, *neighbors_algorithm='auto'*, *n_jobs=None*)

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

Compute the embedding vectors for data X

Parameters

X [{array-like, sparse matrix, BallTree, KDTree, NearestNeighbors}] Sample data, shape = (n_samples, n_features), in the form of a numpy array, precomputed tree, or NearestNeighbors object.

y [Ignored]

Returns

self [returns an instance of self.]

fit_transform(*self*, *X*, *y=None*)
Fit the model from data in X and transform X.

Parameters

X [{array-like, sparse matrix, BallTree, KDTree}] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [Ignored]

Returns

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

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

reconstruction_error(*self*)
Compute the reconstruction error for the embedding.

Returns

reconstruction_error [float]

Notes

The cost function of an isomap embedding is

```
E = frobenius_norm[K(D) - K(D_fit)] / n_samples
```

Where D is the matrix of distances for the input data X, D_fit is the matrix of distances for the output embedding X_fit, and K is the isomap kernel:

```
K(D) = -0.5 * (I - 1/n_samples) * D^2 * (I - 1/n_samples)
```

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform X.

This is implemented by linking the points X into the graph of geodesic distances of the training data. First the `n_neighbors` nearest neighbors of X are found in the training data, and from these the shortest geodesic distances from each point in X to each point in the training data are computed in order to construct the kernel. The embedding of X is the projection of this kernel onto the embedding vectors of the training set.

Parameters

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

Returns

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

Examples using `sklearn.manifold.Isomap`

- *Comparison of Manifold Learning methods*
- *Manifold Learning methods on a severed sphere*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

6.23.2 `sklearn.manifold`.LocallyLinearEmbedding

class `sklearn.manifold.`**`LocallyLinearEmbedding`**(*n_neighbors=5*, *n_components=2*, *reg=0.001*, *eigen_solver='auto'*, *tol=1e-06*, *max_iter=100*, *method='standard'*, *hessian_tol=0.0001*, *modified_tol=1e-12*, *neighbors_algorithm='auto'*, *random_state=None*, *n_jobs=None*)

Locally Linear Embedding

Read more in the *User Guide*.

Parameters

n_neighbors [integer] number of neighbors to consider for each point.

n_components [integer] number of coordinates for the manifold

reg [float] regularization constant, multiplies the trace of the local covariance matrix of the distances.

eigen_solver [string, {'auto', 'arpack', 'dense'}] auto : algorithm will attempt to choose the best method for input data

arpack [use arnoldi iteration in shift-invert mode.] For this method, M may be a dense matrix, sparse matrix, or general linear operator. Warning: ARPACK can be unstable for some problems. It is best to try several random seeds in order to check results.

dense [use standard dense matrix operations for the eigenvalue] decomposition. For this method, M must be an array or matrix type. This method should be avoided for large problems.

tol [float, optional] Tolerance for 'arpack' method Not used if eigen_solver=='dense'.

max_iter [integer] maximum number of iterations for the arpack solver. Not used if eigen_solver=='dense'.

method [string ('standard', 'hessian', 'modified' or 'ltsa')]

standard [use the standard locally linear embedding algorithm. see] reference [1]

hessian [use the Hessian eigenmap method. This method requires] `n_neighbors > n_components * (1 + (n_components + 1) / 2` see reference [2]

modified [use the modified locally linear embedding algorithm.] see reference [3]

ltsa [use local tangent space alignment algorithm] see reference [4]

hessian_tol [float, optional] Tolerance for Hessian eigenmapping method. Only used if `method == 'hessian'`

modified_tol [float, optional] Tolerance for modified LLE method. Only used if `method == 'modified'`

neighbors_algorithm [string ['auto'|'brute'|'kd_tree'|'ball_tree']] algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `eigen_solver` == 'arpack'.

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

Attributes

embedding_ [array-like, shape [n_samples, n_components]] Stores the embedding vectors

reconstruction_error_ [float] Reconstruction error associated with *embedding_*

nbrs_ [NearestNeighbors object] Stores nearest neighbors instance, including BallTree or KDtree if applicable.

References

[R62e36dd1b056-1], *[R62e36dd1b056-2]*, *[R62e36dd1b056-3]*, *[R62e36dd1b056-4]*

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import LocallyLinearEmbedding
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = LocallyLinearEmbedding(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
```

Methods

fit(self, X[, y])	Compute the embedding vectors for data X
fit_transform(self, X[, y])	Compute the embedding vectors for data X and transform X.

Continued on next page

Table 6.173 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform new points into embedding space.

__init__(*self*, *n_neighbors=5*, *n_components=2*, *reg=0.001*, *eigen_solver='auto'*, *tol=1e-06*, *max_iter=100*, *method='standard'*, *hessian_tol=0.0001*, *modified_tol=1e-12*, *neighbors_algorithm='auto'*, *random_state=None*, *n_jobs=None*)

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

Compute the embedding vectors for data X

Parameters

X [array-like of shape [n_samples, n_features]] training set.

y [Ignored]

Returns

self [returns an instance of self.]

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

Compute the embedding vectors for data X and transform X.

Parameters

X [array-like of shape [n_samples, n_features]] training set.

y [Ignored]

Returns

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

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)

Transform new points into embedding space.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

X_new [array, shape = [n_samples, n_components]]

Notes

Because of scaling performed by this method, it is discouraged to use it together with methods that are not scale-invariant (like SVMs)

Examples using `sklearn.manifold.LocallyLinearEmbedding`

- *Visualizing the stock market structure*
- *Comparison of Manifold Learning methods*
- *Manifold Learning methods on a severed sphere*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

6.23.3 `sklearn.manifold.MDS`

class `sklearn.manifold.MDS`(*n_components=2*, *metric=True*, *n_init=4*, *max_iter=300*, *verbose=0*, *eps=0.001*, *n_jobs=None*, *random_state=None*, *dissimilarity='euclidean'*)

Multidimensional scaling

Read more in the *User Guide*.

Parameters

n_components [int, optional, default: 2] Number of dimensions in which to immerse the dissimilarities.

metric [boolean, optional, default: True] If `True`, perform metric MDS; otherwise, perform nonmetric MDS.

n_init [int, optional, default: 4] Number of times the SMACOF algorithm will be run with different initializations. The final results will be the best output of the runs, determined by the run with the smallest final stress.

max_iter [int, optional, default: 300] Maximum number of iterations of the SMACOF algorithm for a single run.

verbose [int, optional, default: 0] Level of verbosity.

eps [float, optional, default: 1e-3] Relative tolerance with respect to stress at which to declare convergence.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. If multiple initializations are used (`n_init`), each run of the algorithm is computed in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional, default: None] The generator used to initialize the centers. 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`.

dissimilarity ['euclidean' | 'precomputed', optional, default: 'euclidean'] Dissimilarity measure to use:

- **'euclidean':** Pairwise Euclidean distances between points in the dataset.
- **'precomputed':** Pre-computed dissimilarities are passed directly to `fit` and `fit_transform`.

Attributes

embedding_ [array-like, shape (n_samples, n_components)] Stores the position of the dataset in the embedding space.

stress_ [float] The final value of the stress (sum of squared distance of the disparities and the distances for all constrained points).

References

"Modern Multidimensional Scaling - Theory and Applications" Borg, I.; Groenen P. Springer Series in Statistics (1997)

"Nonmetric multidimensional scaling: a numerical method" Kruskal, J. Psychometrika, 29 (1964)

"Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis" Kruskal, J. Psychometrika, 29, (1964)

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import MDS
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = MDS(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
```

Methods

fit(self, X[, y, init])	Computes the position of the points in the embedding space
fit_transform(self, X[, y, init])	Fit the data from X, and returns the embedded coordinates
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_components=2*, *metric=True*, *n_init=4*, *max_iter=300*, *verbose=0*, *eps=0.001*, *n_jobs=None*, *random_state=None*, *dissimilarity='euclidean'*)

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

Computes the position of the points in the embedding space

Parameters

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] Input data. If `dissimilarity=='precomputed'`, the input should be the dissimilarity matrix.

y [Ignored]

init [ndarray, shape (n_samples,), optional, default: None] Starting configuration of the embedding to initialize the SMACOF algorithm. By default, the algorithm is initialized with a randomly chosen array.

fit_transform(*self*, *X*, *y=None*, *init=None*)
Fit the data from X, and returns the embedded coordinates

Parameters

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] Input data. If `dissimilarity=='precomputed'`, the input should be the dissimilarity matrix.

y [Ignored]

init [ndarray, shape (n_samples,), optional, default: None] Starting configuration of the embedding to initialize the SMACOF algorithm. By default, the algorithm is initialized with a randomly chosen array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

Examples using `sklearn.manifold.MDS`

- *Multi-dimensional scaling*
- *Comparison of Manifold Learning methods*
- *Manifold Learning methods on a severed sphere*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

6.23.4 sklearn.manifold.SpectralEmbedding

class sklearn.manifold.**SpectralEmbedding**(*n_components=2*, *affinity='nearest_neighbors'*, *gamma=None*, *random_state=None*, *eigen_solver=None*, *n_neighbors=None*, *n_jobs=None*)

Spectral embedding for non-linear dimensionality reduction.

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

Note : Laplacian Eigenmaps is the actual algorithm implemented here.

Read more in the *User Guide*.

Parameters

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

affinity [string or callable, default]

How to construct the affinity matrix.

- 'nearest_neighbors' : construct affinity matrix by knn graph
- 'rbf' : construct affinity matrix by rbf kernel
- 'precomputed' : interpret X as precomputed affinity matrix
- callable : use passed in function as affinity the function takes in data matrix (n_samples, n_features) and return affinity matrix (n_samples, n_samples).

gamma [float, optional, default] Kernel coefficient for rbf kernel.

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

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

n_neighbors [int, default] Number of nearest neighbors for nearest_neighbors graph building.

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

Attributes

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

affinity_matrix_ [array, shape = (n_samples, n_samples)] Affinity_matrix constructed from samples or precomputed.

References

- A Tutorial on Spectral Clustering, 2007 Ulrike von Luxburg http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.9323

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

Examples

```
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import SpectralEmbedding
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = SpectralEmbedding(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
```

Methods

fit(self, X[, y])	Fit the model from data in X.
fit_transform(self, X[, y])	Fit the model from data in X and transform X.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_components=2*, *affinity='nearest_neighbors'*, *gamma=None*, *random_state=None*, *eigen_solver=None*, *n_neighbors=None*, *n_jobs=None*)

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

Fit the model from data in X.

Parameters

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

If affinity is "precomputed" X : array-like, shape (n_samples, n_samples), Interpret X as precomputed adjacency graph computed from samples.

Returns

self [object] Returns the instance itself.

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

Fit the model from data in X and transform X.

Parameters

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

If affinity is "precomputed" X : array-like, shape (n_samples, n_samples), Interpret X as precomputed adjacency graph computed from samples.

Returns

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

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

Examples using `sklearn.manifold.SpectralEmbedding`

- *Various Agglomerative Clustering on a 2D embedding of digits*
- *Comparison of Manifold Learning methods*
- *Manifold Learning methods on a severed sphere*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

6.23.5 sklearn.manifold.TSNE

class sklearn.manifold.**TSNE**(*n_components=2*, *perplexity=30.0*, *early_exaggeration=12.0*, *learning_rate=200.0*, *n_iter=1000*, *n_iter_without_progress=300*, *min_grad_norm=1e-07*, *metric='euclidean'*, *init='random'*, *verbose=0*, *random_state=None*, *method='barnes_hut'*, *angle=0.5*)
t-distributed Stochastic Neighbor Embedding.

t-SNE [1] is a tool to visualize high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. t-SNE has a cost function that is not convex, i.e. with different initializations we can get different results.

It is highly recommended to use another dimensionality reduction method (e.g. PCA for dense data or TruncatedSVD for sparse data) to reduce the number of dimensions to a reasonable amount (e.g. 50) if the number of features is very high. This will suppress some noise and speed up the computation of pairwise distances between samples. For more tips see Laurens van der Maaten's FAQ [2].

Read more in the *User Guide*.

Parameters

n_components [int, optional (default: 2)] Dimension of the embedded space.

perplexity [float, optional (default: 30)] The perplexity is related to the number of nearest neighbors that is used in other manifold learning algorithms. Larger datasets usually require a larger perplexity. Consider selecting a value between 5 and 50. Different values can result in significanlty different results.

early_exaggeration [float, optional (default: 12.0)] Controls how tight natural clusters in the original space are in the embedded space and how much space will be between them. For larger values, the space between natural clusters will be larger in the embedded space. Again, the choice of this parameter is not very critical. If the cost function increases during initial optimization, the early exaggeration factor or the learning rate might be too high.

learning_rate [float, optional (default: 200.0)] The learning rate for t-SNE is usually in the range [10.0, 1000.0]. If the learning rate is too high, the data may look like a 'ball' with any point approximately equidistant from its nearest neighbours. If the learning rate is too low, most points may look compressed in a dense cloud with few outliers. If the cost function gets stuck in a bad local minimum increasing the learning rate may help.

n_iter [int, optional (default: 1000)] Maximum number of iterations for the optimization. Should be at least 250.

n_iter_without_progress [int, optional (default: 300)] Maximum number of iterations without progress before we abort the optimization, used after 250 initial iterations with early exaggeration. Note that progress is only checked every 50 iterations so this value is rounded to the next multiple of 50.

New in version 0.17: parameter *n_iter_without_progress* to control stopping criteria.

min_grad_norm [float, optional (default: 1e-7)] If the gradient norm is below this threshold, the optimization will be stopped.

metric [string or callable, optional] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by scipy.spatial.distance.pdist for its metric parameter, or a metric listed in pairwise.PAIRWISE_DISTANCE_FUNCTIONS. If metric is "precomputed", X is assumed to be a distance matrix. Alternatively, 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 from X as input and return a value indicating the distance between them. The default is "euclidean" which is interpreted as squared euclidean distance.

init [string or numpy array, optional (default: "random")] Initialization of embedding. Possible options are 'random', 'pca', and a numpy array of shape (n_samples, n_components). PCA initialization cannot be used with precomputed distances and is usually more globally stable than random initialization.

verbose [int, optional (default: 0)] Verbosity level.

random_state [int, RandomState instance or None, optional (default: None)] 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`. Note that different initializations might result in different local minima of the cost function.

method [string (default: 'barnes_hut')] By default the gradient calculation algorithm uses Barnes-Hut approximation running in O(NlogN) time. method='exact' will run on the slower, but exact, algorithm in O(N^2) time. The exact algorithm should be used when nearest-neighbor errors need to be better than 3%. However, the exact method cannot scale to millions of examples.

New in version 0.17: Approximate optimization *method* via the Barnes-Hut.

angle [float (default: 0.5)] Only used if method='barnes_hut' This is the trade-off between speed and accuracy for Barnes-Hut T-SNE. 'angle' is the angular size (referred to as theta in [3]) of a distant node as measured from a point. If this size is below 'angle' then it is used as a summary node of all points contained within it. This method is not very sensitive to

changes in this parameter in the range of 0.2 - 0.8. Angle less than 0.2 has quickly increasing computation time and angle greater 0.8 has quickly increasing error.

Attributes

embedding_ [array-like, shape (n_samples, n_components)] Stores the embedding vectors.

kl_divergence_ [float] Kullback-Leibler divergence after optimization.

n_iter_ [int] Number of iterations run.

References

[1] van der Maaten, L.J.P.; Hinton, G.E. Visualizing High-Dimensional Data Using t-SNE. Journal of Machine Learning Research 9:2579-2605, 2008.

[2] van der Maaten, L.J.P. t-Distributed Stochastic Neighbor Embedding https://lvdmaaten.github.io/tsne/

[3] L.J.P. van der Maaten. Accelerating t-SNE using Tree-Based Algorithms. Journal of Machine Learning Research 15(Oct):3221-3245, 2014. https://lvdmaaten.github.io/publications/papers/JMLR_2014.pdf

Examples

```
>>> import numpy as np
>>> from sklearn.manifold import TSNE
>>> X = np.array([[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]])
>>> X_embedded = TSNE(n_components=2).fit_transform(X)
>>> X_embedded.shape
(4, 2)
```

Methods

fit(self, X[, y])	Fit X into an embedded space.
fit_transform(self, X[, y])	Fit X into an embedded space and return that transformed output.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_components=2*, *perplexity=30.0*, *early_exaggeration=12.0*, *learning_rate=200.0*, *n_iter=1000*, *n_iter_without_progress=300*, *min_grad_norm=1e-07*, *metric='euclidean'*, *init='random'*, *verbose=0*, *random_state=None*, *method='barnes_hut'*, *angle=0.5*)

fit(*self*, *X*, *y=None*)
Fit X into an embedded space.

Parameters

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] If the metric is 'precomputed' X must be a square distance matrix. Otherwise it contains a sample per row. If the method is 'exact', X may be a sparse matrix of type 'csr', 'csc' or 'coo'.

y [Ignored]

fit_transform(*self*, *X*, *y=None*)
Fit X into an embedded space and return that transformed output.

Parameters

X [array, shape (n_samples, n_features) or (n_samples, n_samples)] If the metric is 'precomputed' X must be a square distance matrix. Otherwise it contains a sample per row.

y [Ignored]

Returns

X_new [array, shape (n_samples, n_components)] Embedding of the training data in low-dimensional space.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

Examples using `sklearn.manifold.TSNE`

- *t-SNE: The effect of various perplexity values on the shape*
- *Comparison of Manifold Learning methods*
- *Manifold Learning methods on a severed sphere*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

manifold.locally_linear_embedding(X, ...[, ...])	Perform a Locally Linear Embedding analysis on the data.
manifold.smacof(dissimilarities[, metric, ...])	Computes multidimensional scaling using the SMACOF algorithm.
manifold.spectral_embedding(adjacency[, ...])	Project the sample on the first eigenvectors of the graph Laplacian.

6.23.6 `sklearn.manifold`.locally_linear_embedding

sklearn.manifold.**locally_linear_embedding**(*X*, *n_neighbors*, *n_components*, *reg=0.001*, *eigen_solver='auto'*, *tol=1e-06*, *max_iter=100*, *method='standard'*, *hessian_tol=0.0001*, *modified_tol=1e-12*, *random_state=None*, *n_jobs=None*)
Perform a Locally Linear Embedding analysis on the data.

Read more in the *User Guide*.

Parameters

X [{array-like, NearestNeighbors}] Sample data, shape = (n_samples, n_features), in the form of a numpy array or a NearestNeighbors object.

n_neighbors [integer] number of neighbors to consider for each point.

n_components [integer] number of coordinates for the manifold.

reg [float] regularization constant, multiplies the trace of the local covariance matrix of the distances.

eigen_solver [string, {'auto', 'arpack', 'dense'}] auto : algorithm will attempt to choose the best method for input data

arpack [use arnoldi iteration in shift-invert mode.] For this method, M may be a dense matrix, sparse matrix, or general linear operator. Warning: ARPACK can be unstable for some problems. It is best to try several random seeds in order to check results.

dense [use standard dense matrix operations for the eigenvalue] decomposition. For this method, M must be an array or matrix type. This method should be avoided for large problems.

tol [float, optional] Tolerance for 'arpack' method Not used if eigen_solver=='dense'.

max_iter [integer] maximum number of iterations for the arpack solver.

method [{'standard', 'hessian', 'modified', 'ltsa'}]

standard [use the standard locally linear embedding algorithm.] see reference *[1]*

hessian [use the Hessian eigenmap method. This method requires] n_neighbors > n_components * (1 + (n_components + 1) / 2. see reference *[2]*

modified [use the modified locally linear embedding algorithm.] see reference *[3]*

ltsa [use local tangent space alignment algorithm] see reference *[4]*

hessian_tol [float, optional] Tolerance for Hessian eigenmapping method. Only used if method == 'hessian'

modified_tol [float, optional] Tolerance for modified LLE method. Only used if method == 'modified'

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'arpack'.

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

Y [array-like, shape [n_samples, n_components]] Embedding vectors.

squared_error [float] Reconstruction error for the embedding vectors. Equivalent to `norm(Y - W Y, 'fro')**2`, where W are the reconstruction weights.

References

[1], *[2]*, *[3]*, *[4]*

Examples using `sklearn.manifold.locally_linear_embedding`

- *Swiss Roll reduction with LLE*

6.23.7 `sklearn.manifold`.smacof

`sklearn.manifold.`**`smacof`**(*dissimilarities*, *metric=True*, *n_components=2*, *init=None*, *n_init=8*, *n_jobs=None*, *max_iter=300*, *verbose=0*, *eps=0.001*, *random_state=None*, *return_n_iter=False*)

Computes multidimensional scaling using the SMACOF algorithm.

The SMACOF (Scaling by MAjorizing a COmplicated Function) algorithm is a multidimensional scaling algorithm which minimizes an objective function (the *stress*) using a majorization technique. Stress majorization, also known as the Guttman Transform, guarantees a monotone convergence of stress, and is more powerful than traditional techniques such as gradient descent.

The SMACOF algorithm for metric MDS can summarized by the following steps:

1. Set an initial start configuration, randomly or not.
2. Compute the stress
3. Compute the Guttman Transform
4. Iterate 2 and 3 until convergence.

The nonmetric algorithm adds a monotonic regression step before computing the stress.

Parameters

dissimilarities [ndarray, shape (n_samples, n_samples)] Pairwise dissimilarities between the points. Must be symmetric.

metric [boolean, optional, default: True] Compute metric or nonmetric SMACOF algorithm.

n_components [int, optional, default: 2] Number of dimensions in which to immerse the dissimilarities. If an `init` array is provided, this option is overridden and the shape of `init` is used to determine the dimensionality of the embedding space.

init [ndarray, shape (n_samples, n_components), optional, default: None] Starting configuration of the embedding to initialize the algorithm. By default, the algorithm is initialized with a randomly chosen array.

n_init [int, optional, default: 8] Number of times the SMACOF algorithm will be run with different initializations. The final results will be the best output of the runs, determined by the run with the smallest final stress. If `init` is provided, this option is overridden and a single run is performed.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. If multiple initializations are used (`n_init`), each run of the algorithm is computed in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

max_iter [int, optional, default: 300] Maximum number of iterations of the SMACOF algorithm for a single run.

verbose [int, optional, default: 0] Level of verbosity.

eps [float, optional, default: 1e-3] Relative tolerance with respect to stress at which to declare convergence.

random_state [int, RandomState instance or None, optional, default: None] The generator used to initialize the centers. 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`.

return_n_iter [bool, optional, default: False] Whether or not to return the number of iterations.

Returns

X [ndarray, shape (n_samples, n_components)] Coordinates of the points in a `n_components`-space.

stress [float] The final value of the stress (sum of squared distance of the disparities and the distances for all constrained points).

n_iter [int] The number of iterations corresponding to the best stress. Returned only if `return_n_iter` is set to `True`.

Notes

"Modern Multidimensional Scaling - Theory and Applications" Borg, I.; Groenen P. Springer Series in Statistics (1997)

"Nonmetric multidimensional scaling: a numerical method" Kruskal, J. Psychometrika, 29 (1964)

"Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis" Kruskal, J. Psychometrika, 29, (1964)

6.23.8 `sklearn.manifold`.spectral_embedding

`sklearn.manifold.`**`spectral_embedding`**(*adjacency*, *n_components=8*, *eigen_solver=None*, *random_state=None*, *eigen_tol=0.0*, *norm_laplacian=True*, *drop_first=True*)

Project the sample on the first eigenvectors of the graph Laplacian.

The adjacency matrix is used to compute a normalized graph Laplacian whose spectrum (especially the eigenvectors associated to the smallest eigenvalues) has an interpretation in terms of minimal number of cuts necessary to split the graph into comparably sized components.

This embedding can also 'work' even if the `adjacency` variable is not strictly the adjacency matrix of a graph but more generally an affinity or similarity matrix between samples (for instance the heat kernel of a euclidean distance matrix or a k-NN matrix).

However care must taken to always make the affinity matrix symmetric so that the eigenvector decomposition works as expected.

Note : Laplacian Eigenmaps is the actual algorithm implemented here.

Read more in the *User Guide*.

Parameters

adjacency [array-like or sparse matrix, shape: (n_samples, n_samples)] The adjacency matrix of the graph to embed.

n_components [integer, optional, default 8] The dimension of the projection subspace.

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

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

eigen_tol [float, optional, default=0.0] Stopping criterion for eigendecomposition of the Laplacian matrix when using arpack eigen_solver.

norm_laplacian [bool, optional, default=True] If True, then compute normalized Laplacian.

drop_first [bool, optional, default=True] Whether to drop the first eigenvector. For spectral embedding, this should be True as the first eigenvector should be constant vector for connected graph, but for spectral clustering, this should be kept as False to retain the first eigenvector.

Returns

embedding [array, shape=(n_samples, n_components)] The reduced samples.

Notes

Spectral Embedding (Laplacian Eigenmaps) is most useful when the graph has one connected component. If there graph has many components, the first few eigenvectors will simply uncover the connected components of the graph.

References

- https://en.wikipedia.org/wiki/LOBPCG
- Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method Andrew V. Knyazev https://doi.org/10.1137%2FS1064827500366124

6.24 `sklearn.metrics`: Metrics

See the *Model evaluation: quantifying the quality of predictions* section and the *Pairwise metrics, Affinities and Kernels* section of the user guide for further details. The `sklearn.metrics` module includes score functions, performance metrics and pairwise metrics and distance computations.

6.24.1 Model Selection Interface

See the *The scoring parameter: defining model evaluation rules* section of the user guide for further details.

`metrics.check_scoring`(estimator[, scoring, ...])	Determine scorer from user options.
`metrics.get_scorer`(scoring)	Get a scorer from string

Continued on next page

Table 6.178 – continued from previous page

metrics.make_scorer(score_func[, ...])	Make a scorer from a performance metric or loss function.

sklearn.metrics.check_scoring

sklearn.metrics.**check_scoring**(*estimator*, *scoring=None*, *allow_none=False*)

Determine scorer from user options.

A TypeError will be thrown if the estimator cannot be scored.

Parameters

estimator [estimator object implementing 'fit'] The object to use to fit the data.

scoring [string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

allow_none [boolean, optional, default: False] If no scoring is specified and the estimator has no score function, we can either return None or raise an exception.

Returns

scoring [callable] A scorer callable object / function with signature `scorer(estimator, X, y)`.

sklearn.metrics.get_scorer

sklearn.metrics.**get_scorer**(*scoring*)

Get a scorer from string

Parameters

scoring [str | callable] scoring method as string. If callable it is returned as is.

Returns

scorer [callable] The scorer.

sklearn.metrics.make_scorer

sklearn.metrics.**make_scorer**(*score_func*, *greater_is_better=True*, *needs_proba=False*, *needs_threshold=False*, ***kwargs*)

Make a scorer from a performance metric or loss function.

This factory function wraps scoring functions for use in GridSearchCV and cross_val_score. It takes a score function, such as `accuracy_score`, `mean_squared_error`, `adjusted_rand_index` or `average_precision` and returns a callable that scores an estimator's output.

Read more in the *User Guide*.

Parameters

score_func [callable,] Score function (or loss function) with signature `score_func(y, y_pred, **kwargs)`.

greater_is_better [boolean, default=True] Whether score_func is a score function (default), meaning high is good, or a loss function, meaning low is good. In the latter case, the scorer object will sign-flip the outcome of the score_func.

needs_proba [boolean, default=False] Whether score_func requires predict_proba to get probability estimates out of a classifier.

If True, for binary `y_true`, the score function is supposed to accept a 1D `y_pred` (i.e., probability of the positive class, shape `(n_samples,)`).

needs_threshold [boolean, default=False] Whether score_func takes a continuous decision certainty. This only works for binary classification using estimators that have either a decision_function or predict_proba method.

If True, for binary `y_true`, the score function is supposed to accept a 1D `y_pred` (i.e., probability of the positive class or the decision function, shape `(n_samples,)`).

For example `average_precision` or the area under the roc curve can not be computed using discrete predictions alone.

****kwargs** [additional arguments] Additional parameters to be passed to score_func.

Returns

scorer [callable] Callable object that returns a scalar score; greater is better.

Examples

```
>>> from sklearn.metrics import fbeta_score, make_scorer
>>> ftwo_scorer = make_scorer(fbeta_score, beta=2)
>>> ftwo_scorer
make_scorer(fbeta_score, beta=2)
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import LinearSVC
>>> grid = GridSearchCV(LinearSVC(), param_grid={'C': [1, 10]},
...                     scoring=ftwo_scorer)
```

Examples using `sklearn.metrics.make_scorer`

- *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV*

6.24.2 Classification metrics

See the *Classification metrics* section of the user guide for further details.

metrics.accuracy_score(y_true, y_pred[, ...])	Accuracy classification score.
metrics.auc(x, y[, reorder])	Compute Area Under the Curve (AUC) using the trapezoidal rule
metrics.average_precision_score(y_true, y_score)	Compute average precision (AP) from prediction scores
metrics.balanced_accuracy_score(y_true, y_pred)	Compute the balanced accuracy
metrics.brier_score_loss(y_true, y_prob[, ...])	Compute the Brier score.
metrics.classification_report(y_true, y_pred)	Build a text report showing the main classification metrics
metrics.cohen_kappa_score(y1, y2[, labels, ...])	Cohen's kappa: a statistic that measures inter-annotator agreement.

Continued on next page

Table 6.179 – continued from previous page

metrics.confusion_matrix(y_true, y_pred[, ...])	Compute confusion matrix to evaluate the accuracy of a classification
metrics.f1_score(y_true, y_pred[, labels, ...])	Compute the F1 score, also known as balanced F-score or F-measure
metrics.fbeta_score(y_true, y_pred, beta[, ...])	Compute the F-beta score
metrics.hamming_loss(y_true, y_pred[, ...])	Compute the average Hamming loss.
metrics.hinge_loss(y_true, pred_decision[, ...])	Average hinge loss (non-regularized)
metrics.jaccard_score(y_true, y_pred[, ...])	Jaccard similarity coefficient score
metrics.log_loss(y_true, y_pred[, eps, ...])	Log loss, aka logistic loss or cross-entropy loss.
metrics.matthews_corrcoef(y_true, y_pred[, ...])	Compute the Matthews correlation coefficient (MCC)
metrics.multilabel_confusion_matrix(y_true, ...)	Compute a confusion matrix for each class or sample
metrics.precision_recall_curve(y_true, ...)	Compute precision-recall pairs for different probability thresholds
metrics.precision_recall_fscore_support(...)	Compute precision, recall, F-measure and support for each class
metrics.precision_score(y_true, y_pred[, ...])	Compute the precision
metrics.recall_score(y_true, y_pred[, ...])	Compute the recall
metrics.roc_auc_score(y_true, y_score[, ...])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.
metrics.roc_curve(y_true, y_score[, ...])	Compute Receiver operating characteristic (ROC)
metrics.zero_one_loss(y_true, y_pred[, ...])	Zero-one classification loss.

sklearn.metrics.accuracy_score

sklearn.metrics.**accuracy_score**(*y_true*, *y_pred*, *normalize=True*, *sample_weight=None*)

Accuracy classification score.

In multilabel classification, this function computes subset accuracy: the set of labels predicted for a sample must *exactly* match the corresponding set of labels in y_true.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array / sparse matrix] Predicted labels, as returned by a classifier.

normalize [bool, optional (default=True)] If `False`, return the number of correctly classified samples. Otherwise, return the fraction of correctly classified samples.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

score [float] If `normalize == True`, return the fraction of correctly classified samples (float), else returns the number of correctly classified samples (int).

The best performance is 1 with `normalize == True` and the number of samples with `normalize == False`.

See also:

jaccard_score*, *hamming_loss*, *zero_one_loss

Notes

In binary and multiclass classification, this function is equal to the `jaccard_score` function.

Examples

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
0.5
>>> accuracy_score(y_true, y_pred, normalize=False)
2
```

In the multilabel case with binary label indicators:

```
>>> import numpy as np
>>> accuracy_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
```

Examples using `sklearn.metrics.accuracy_score`

- *Plot classification probability*
- *Multi-class AdaBoosted Decision Trees*
- *Probabilistic predictions with Gaussian process classification (GPC)*
- *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV*
- *Importance of Feature Scaling*
- *Classification of text documents using sparse features*

`sklearn.metrics.auc`

`sklearn.metrics.auc`(*x*, *y*, *reorder='deprecated'*)

Compute Area Under the Curve (AUC) using the trapezoidal rule

This is a general function, given points on a curve. For computing the area under the ROC-curve, see *roc_auc_score*. For an alternative way to summarize a precision-recall curve, see *average_precision_score*.

Parameters

x [array, shape = [n]] x coordinates. These must be either monotonic increasing or monotonic decreasing.

y [array, shape = [n]] y coordinates.

reorder [boolean, optional (default='deprecated')] Whether to sort x before computing. If False, assume that x must be either monotonic increasing or monotonic decreasing. If True, y is used to break ties when sorting x. Make sure that y has a monotonic relation to x when setting reorder to True.

Deprecated since version 0.20: Parameter `reorder` has been deprecated in version 0.20 and will be removed in 0.22. It's introduced for roc_auc_score (not for general use) and is

no longer used there. What's more, the result from auc will be significantly influenced if x is sorted unexpectedly due to slight floating point error (See issue #9786). Future (and default) behavior is equivalent to `reorder=False`.

Returns

auc [float]

See also:

roc_auc_score Compute the area under the ROC curve

average_precision_score Compute average precision from prediction scores

precision_recall_curve Compute precision-recall pairs for different probability thresholds

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> pred = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, pred, pos_label=2)
>>> metrics.auc(fpr, tpr)
0.75
```

Examples using `sklearn.metrics.auc`

- *Species distribution modeling*
- *Receiver Operating Characteristic (ROC) with cross validation*
- *Receiver Operating Characteristic (ROC)*

`sklearn.metrics.average_precision_score`

`sklearn.metrics.`**`average_precision_score`**(*y_true*, *y_score*, *average='macro'*, *pos_label=1*, *sample_weight=None*)

Compute average precision (AP) from prediction scores

AP summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight:

$$\text{AP} = \sum_n (R_n - R_{n-1}) P_n$$

where P_n and R_n are the precision and recall at the nth threshold *[1]*. This implementation is not interpolated and is different from computing the area under the precision-recall curve with the trapezoidal rule, which uses linear interpolation and can be too optimistic.

Note: this implementation is restricted to the binary classification task or multilabel classification task.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples] or [n_samples, n_classes]] True binary labels or binary label indicators.

y_score [array, shape = [n_samples] or [n_samples, n_classes]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

average [string, [None, 'micro', 'macro' (default), 'samples', 'weighted']] If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

'micro': Calculate metrics globally by considering each element of the label indicator matrix as a label.

'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).

'samples': Calculate metrics for each instance, and find their average.

Will be ignored when `y_true` is binary.

pos_label [int or str (default=1)] The label of the positive class. Only applied to binary `y_true`. For multilabel-indicator `y_true`, `pos_label` is fixed to 1.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

average_precision [float]

See also:

roc_auc_score Compute the area under the ROC curve

precision_recall_curve Compute precision-recall pairs for different probability thresholds

Notes

Changed in version 0.19: Instead of linearly interpolating between operating points, precisions are weighted by the change in recall since the last operating point.

References

[1]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import average_precision_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> average_precision_score(y_true, y_scores)
0.83...
```

Examples using `sklearn.metrics.average_precision_score`

- *Precision-Recall*

`sklearn.metrics.balanced_accuracy_score`

`sklearn.metrics.balanced_accuracy_score(y_true, y_pred, sample_weight=None, adjusted=False)`

Compute the balanced accuracy

The balanced accuracy in binary and multiclass classification problems to deal with imbalanced datasets. It is defined as the average of recall obtained on each class.

The best value is 1 and the worst value is 0 when `adjusted=False`.

Read more in the *User Guide*.

Parameters

y_true [1d array-like] Ground truth (correct) target values.

y_pred [1d array-like] Estimated targets as returned by a classifier.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

adjusted [bool, default=False] When true, the result is adjusted for chance, so that random performance would score 0, and perfect performance scores 1.

Returns

balanced_accuracy [float]

See also:

recall_score, ***roc_auc_score***

Notes

Some literature promotes alternative definitions of balanced accuracy. Our definition is equivalent to *accuracy_score* with class-balanced sample weights, and shares desirable properties with the binary case. See the *User Guide*.

References

[1], *[2]*

Examples

```
>>> from sklearn.metrics import balanced_accuracy_score
>>> y_true = [0, 1, 0, 0, 1, 0]
>>> y_pred = [0, 1, 0, 0, 0, 1]
>>> balanced_accuracy_score(y_true, y_pred)
0.625
```

sklearn.metrics.brier_score_loss

sklearn.metrics.**brier_score_loss**(*y_true*, *y_prob*, *sample_weight=None*, *pos_label=None*)

Compute the Brier score. The smaller the Brier score, the better, hence the naming with "loss". Across all items in a set N predictions, the Brier score measures the mean squared difference between (1) the predicted probability assigned to the possible outcomes for item i, and (2) the actual outcome. Therefore, the lower the Brier score is for a set of predictions, the better the predictions are calibrated. Note that the Brier score always takes on a value between zero and one, since this is the largest possible difference between a predicted probability (which must be between zero and one) and the actual outcome (which can take on values of only 0 and 1). The Brier loss is composed of refinement loss and calibration loss. The Brier score is appropriate for binary and categorical outcomes that can be structured as true or false, but is inappropriate for ordinal variables which can take on three or more values (this is because the Brier score assumes that all possible outcomes are equivalently "distant" from one another). Which label is considered to be the positive label is controlled via the parameter pos_label, which defaults to 1. Read more in the *User Guide*.

Parameters

y_true [array, shape (n_samples,)] True targets.

y_prob [array, shape (n_samples,)] Probabilities of the positive class.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

pos_label [int or str, default=None] Label of the positive class. Defaults to the greater label unless y_true is all 0 or all -1 in which case pos_label defaults to 1.

Returns

score [float] Brier score

References

[1]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss
>>> y_true = np.array([0, 1, 1, 0])
>>> y_true_categorical = np.array(["spam", "ham", "ham", "spam"])
>>> y_prob = np.array([0.1, 0.9, 0.8, 0.3])
>>> brier_score_loss(y_true, y_prob)
0.037...
>>> brier_score_loss(y_true, 1-y_prob, pos_label=0)
0.037...
>>> brier_score_loss(y_true_categorical, y_prob,                                pos_
↪label="ham")
0.037...
>>> brier_score_loss(y_true, np.array(y_prob) > 0.5)
0.0
```

Examples using sklearn.metrics.brier_score_loss

- *Probability Calibration curves*

- *Probability calibration of classifiers*

sklearn.metrics.classification_report

sklearn.metrics.**classification_report**(*y_true*, *y_pred*, *labels=None*, *target_names=None*, *sample_weight=None*, *digits=2*, *output_dict=False*)

Build a text report showing the main classification metrics

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

labels [array, shape = [n_labels]] Optional list of label indices to include in the report.

target_names [list of strings] Optional display names matching the labels (same order).

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

digits [int] Number of digits for formatting output floating point values. When `output_dict` is `True`, this will be ignored and the returned values will not be rounded.

output_dict [bool (default = False)] If True, return output as dict

Returns

report [string / dict] Text summary of the precision, recall, F1 score for each class. Dictionary returned if output_dict is True. Dictionary has the following structure:

```
{'label 1': {'precision':0.5,
             'recall':1.0,
             'f1-score':0.67,
             'support':1},
 'label 2': { ... },
  ...
}
```

The reported averages include macro average (averaging the unweighted mean per label), weighted average (averaging the support-weighted mean per label), sample average (only for multilabel classification) and micro average (averaging the total true positives, false negatives and false positives) it is only shown for multi-label or multi-class with a subset of classes because it is accuracy otherwise. See also:func:*precision_recall_fscore_support* for more details on averages.

Note that in binary classification, recall of the positive class is also known as "sensitivity"; recall of the negative class is "specificity".

See also:

precision_recall_fscore_support, ***confusion_matrix***

multilabel_confusion_matrix

Examples

```
>>> from sklearn.metrics import classification_report
>>> y_true = [0, 1, 2, 2, 2]
>>> y_pred = [0, 0, 2, 2, 1]
>>> target_names = ['class 0', 'class 1', 'class 2']
>>> print(classification_report(y_true, y_pred, target_names=target_names))
              precision    recall  f1-score   support

     class 0       0.50      1.00      0.67         1
     class 1       0.00      0.00      0.00         1
     class 2       1.00      0.67      0.80         3

    accuracy                           0.60         5
   macro avg       0.50      0.56      0.49         5
weighted avg       0.70      0.60      0.61         5

>>> y_pred = [1, 1, 0]
>>> y_true = [1, 1, 1]
>>> print(classification_report(y_true, y_pred, labels=[1, 2, 3]))
              precision    recall  f1-score   support

           1       1.00      0.67      0.80         3
           2       0.00      0.00      0.00         0
           3       0.00      0.00      0.00         0

   micro avg       1.00      0.67      0.80         3
   macro avg       0.33      0.22      0.27         3
weighted avg       1.00      0.67      0.80         3
```

Examples using `sklearn.metrics.classification_report`

- *Faces recognition example using eigenfaces and SVMs*
- *Recognizing hand-written digits*
- *Column Transformer with Heterogeneous Data Sources*
- *Pipeline Anova SVM*
- *Parameter estimation using grid search with cross-validation*
- *Restricted Boltzmann Machine features for digit classification*
- *Label Propagation digits: Demonstrating performance*
- *Label Propagation digits active learning*
- *Classification of text documents using sparse features*

`sklearn.metrics`.cohen_kappa_score

`sklearn.metrics.`**`cohen_kappa_score`**(*y1*, *y2*, *labels=None*, *weights=None*, *sample_weight=None*)

Cohen's kappa: a statistic that measures inter-annotator agreement.

This function computes Cohen's kappa *[1]*, a score that expresses the level of agreement between two annotators on a classification problem. It is defined as

$$\kappa = (p_o - p_e)/(1 - p_e)$$

where p_o is the empirical probability of agreement on the label assigned to any sample (the observed agreement ratio), and p_e is the expected agreement when both annotators assign labels randomly. p_e is estimated using a per-annotator empirical prior over the class labels *[2]*.

Read more in the *User Guide*.

Parameters

y1 [array, shape = [n_samples]] Labels assigned by the first annotator.

y2 [array, shape = [n_samples]] Labels assigned by the second annotator. The kappa statistic is symmetric, so swapping `y1` and `y2` doesn't change the value.

labels [array, shape = [n_classes], optional] List of labels to index the matrix. This may be used to select a subset of labels. If None, all labels that appear at least once in `y1` or `y2` are used.

weights [str, optional] List of weighting type to calculate the score. None means no weighted; "linear" means linear weighted; "quadratic" means quadratic weighted.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

kappa [float] The kappa statistic, which is a number between -1 and 1. The maximum value means complete agreement; zero or lower means chance agreement.

References

[1], *[2]*, *[3]*

sklearn.metrics.confusion_matrix

sklearn.metrics.**confusion_matrix**(*y_true*, *y_pred*, *labels=None*, *sample_weight=None*)

Compute confusion matrix to evaluate the accuracy of a classification

By definition a confusion matrix C is such that $C_{i,j}$ is equal to the number of observations known to be in group i but predicted to be in group j.

Thus in binary classification, the count of true negatives is $C_{0,0}$, false negatives is $C_{1,0}$, true positives is $C_{1,1}$ and false positives is $C_{0,1}$.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples]] Ground truth (correct) target values.

y_pred [array, shape = [n_samples]] Estimated targets as returned by a classifier.

labels [array, shape = [n_classes], optional] List of labels to index the matrix. This may be used to reorder or select a subset of labels. If none is given, those that appear at least once in `y_true` or `y_pred` are used in sorted order.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

C [array, shape = [n_classes, n_classes]] Confusion matrix

References

[1]

Examples

```
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

```
>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> confusion_matrix(y_true, y_pred, labels=["ant", "bird", "cat"])
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

In the binary case, we can extract true positives, etc as follows:

```
>>> tn, fp, fn, tp = confusion_matrix([0, 1, 0, 1], [1, 1, 1, 0]).ravel()
>>> (tn, fp, fn, tp)
(0, 2, 1, 1)
```

Examples using `sklearn.metrics.confusion_matrix`

- *Faces recognition example using eigenfaces and SVMs*
- *Recognizing hand-written digits*
- *Confusion matrix*
- *Label Propagation digits: Demonstrating performance*
- *Label Propagation digits active learning*
- *Classification of text documents using sparse features*

`sklearn.metrics.f1_score`

`sklearn.metrics.`**`f1_score`**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

Compute the F1 score, also known as balanced F-score or F-measure

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

```
F1 = 2 * (precision * recall) / (precision + recall)
```

In the multi-class and multi-label case, this is the average of the F1 score of each class with weighting depending on the `average` parameter.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

Changed in version 0.17: parameter *labels* improved for multiclass problem.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']] This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

`'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

`'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.

`'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

`'weighted'`: Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

`'samples'`: Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *`accuracy_score`*).

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

f1_score [float or array of float, shape = [n_unique_labels]] F1 score of the positive class in binary classification or weighted average of the F1 scores of each class for the multiclass task.

See also:

`fbeta_score`, ***`precision_recall_fscore_support`***, ***`jaccard_score`***

`multilabel_confusion_matrix`

Notes

When `true positive + false positive == 0` or `true positive + false negative == 0`, f-score returns 0 and raises `UndefinedMetricWarning`.

References

[1]

Examples

```
>>> from sklearn.metrics import f1_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> f1_score(y_true, y_pred, average='macro')
0.26...
>>> f1_score(y_true, y_pred, average='micro')
0.33...
>>> f1_score(y_true, y_pred, average='weighted')
0.26...
>>> f1_score(y_true, y_pred, average=None)
array([0.8, 0. , 0. ])
```

Examples using `sklearn.metrics.f1_score`

- *Probability Calibration curves*

`sklearn.metrics.fbeta_score`

`sklearn.metrics.`**`fbeta_score`**(*y_true*, *y_pred*, *beta*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

Compute the F-beta score

The F-beta score is the weighted harmonic mean of precision and recall, reaching its optimal value at 1 and its worst value at 0.

The `beta` parameter determines the weight of recall in the combined score. `beta < 1` lends more weight to precision, while `beta > 1` favors recall (`beta -> 0` considers only precision, `beta -> inf` only recall).

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

beta [float] Weight of precision in harmonic mean.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in

the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

Changed in version 0.17: parameter *labels* improved for multiclass problem.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']] This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

`'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

`'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.

`'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

`'weighted'`: Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

`'samples'`: Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *`accuracy_score`*).

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

fbeta_score [float (if average is not None) or array of float, shape = [n_unique_labels]] F-beta score of the positive class in binary classification or weighted average of the F-beta score of each class for the multiclass task.

See also:

`precision_recall_fscore_support`*, *`multilabel_confusion_matrix`

Notes

When `true positive + false positive == 0` or `true positive + false negative == 0`, f-score returns 0 and raises `UndefinedMetricWarning`.

References

[1], *[2]*

Examples

```
>>> from sklearn.metrics import fbeta_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> fbeta_score(y_true, y_pred, average='macro', beta=0.5)
```

```
...
0.23...
>>> fbeta_score(y_true, y_pred, average='micro', beta=0.5)
...
0.33...
>>> fbeta_score(y_true, y_pred, average='weighted', beta=0.5)
...
0.23...
>>> fbeta_score(y_true, y_pred, average=None, beta=0.5)
...
array([0.71..., 0.        , 0.        ])
```

sklearn.metrics.hamming_loss

sklearn.metrics.**hamming_loss**(*y_true*, *y_pred*, *labels=None*, *sample_weight=None*)

Compute the average Hamming loss.

The Hamming loss is the fraction of labels that are incorrectly predicted.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array / sparse matrix] Predicted labels, as returned by a classifier.

labels [array, shape = [n_labels], optional (default='deprecated')] Integer array of labels. If not provided, labels will be inferred from y_true and y_pred.

New in version 0.18.

Deprecated since version 0.21: This parameter `labels` is deprecated in version 0.21 and will be removed in version 0.23. Hamming loss uses `y_true.shape[1]` for the number of labels when y_true is binary label indicators, so it is unnecessary for the user to specify.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

New in version 0.18.

Returns

loss [float or int,] Return the average Hamming loss between element of `y_true` and `y_pred`.

See also:

accuracy_score, *jaccard_score*, *zero_one_loss*

Notes

In multiclass classification, the Hamming loss corresponds to the Hamming distance between `y_true` and `y_pred` which is equivalent to the subset `zero_one_loss` function, when *normalize* parameter is set to True.

In multilabel classification, the Hamming loss is different from the subset zero-one loss. The zero-one loss considers the entire set of labels for a given sample incorrect if it does not entirely match the true set of labels. Hamming loss is more forgiving in that it penalizes only the individual labels.

The Hamming loss is upperbounded by the subset zero-one loss, when *normalize* parameter is set to True. It is always between 0 and 1, lower being better.

References

[1], *[2]*

Examples

```
>>> from sklearn.metrics import hamming_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> hamming_loss(y_true, y_pred)
0.25
```

In the multilabel case with binary label indicators:

```
>>> import numpy as np
>>> hamming_loss(np.array([[0, 1], [1, 1]]), np.zeros((2, 2)))
0.75
```

Examples using `sklearn.metrics.hamming_loss`

- *Model Complexity Influence*

`sklearn.metrics.hinge_loss`

sklearn.metrics.**hinge_loss**(*y_true*, *pred_decision*, *labels=None*, *sample_weight=None*)

Average hinge loss (non-regularized)

In binary class case, assuming labels in y_true are encoded with +1 and -1, when a prediction mistake is made, `margin = y_true * pred_decision` is always negative (since the signs disagree), implying `1 - margin` is always greater than 1. The cumulated hinge loss is therefore an upper bound of the number of mistakes made by the classifier.

In multiclass case, the function expects that either all the labels are included in y_true or an optional labels argument is provided which contains all the labels. The multilabel margin is calculated according to Crammer-Singer's method. As in the binary case, the cumulated hinge loss is an upper bound of the number of mistakes made by the classifier.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples]] True target, consisting of integers of two values. The positive label must be greater than the negative label.

pred_decision [array, shape = [n_samples] or [n_samples, n_classes]] Predicted decisions, as output by decision_function (floats).

labels [array, optional, default None] Contains all the labels for the problem. Used in multiclass hinge loss.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

loss [float]

References

[1], *[2]*, *[3]*

Examples

```
>>> from sklearn import svm
>>> from sklearn.metrics import hinge_loss
>>> X = [[0], [1]]
>>> y = [-1, 1]
>>> est = svm.LinearSVC(random_state=0)
>>> est.fit(X, y)
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=0.0001,
     verbose=0)
>>> pred_decision = est.decision_function([[-2], [3], [0.5]])
>>> pred_decision
array([-2.18...,  2.36...,  0.09...])
>>> hinge_loss([-1, 1, 1], pred_decision)
0.30...
```

In the multiclass case:

```
>>> import numpy as np
>>> X = np.array([[0], [1], [2], [3]])
>>> Y = np.array([0, 1, 2, 3])
>>> labels = np.array([0, 1, 2, 3])
>>> est = svm.LinearSVC()
>>> est.fit(X, Y)
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
>>> pred_decision = est.decision_function([[-1], [2], [3]])
>>> y_true = [0, 2, 3]
>>> hinge_loss(y_true, pred_decision, labels)
0.56...
```

sklearn.metrics.jaccard_score

sklearn.metrics.**jaccard_score**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

Jaccard similarity coefficient score

The Jaccard index [1], or Jaccard similarity coefficient, defined as the size of the intersection divided by the size of the union of two label sets, is used to compare set of predicted labels for a sample to the corresponding set of labels in `y_true`.

Read more in the User Guide.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array / sparse matrix] Predicted labels, as returned by a classifier.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']] If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

`'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

`'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.

`'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

`'weighted'`: Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance.

`'samples'`: Calculate metrics for each instance, and find their average (only meaningful for multilabel classification).

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

score [float (if average is not None) or array of floats, shape = [n_unique_labels]]

See also:

accuracy_score*, f_score, *multilabel_confusion_matrix

Notes

jaccard_score may be a poor metric if there are no positives for some samples or classes. Jaccard is undefined if there are no true or predicted labels, and our implementation will return a score of 0 with a warning.

References

[1]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import jaccard_score
>>> y_true = np.array([[0, 1, 1],
...                    [1, 1, 0]])
>>> y_pred = np.array([[1, 1, 1],
...                    [1, 0, 0]])
```

In the binary case:

```
>>> jaccard_score(y_true[0], y_pred[0])
0.6666...
```

In the multilabel case:

```
>>> jaccard_score(y_true, y_pred, average='samples')
0.5833...
>>> jaccard_score(y_true, y_pred, average='macro')
0.6666...
>>> jaccard_score(y_true, y_pred, average=None)
array([0.5, 0.5, 1. ])
```

In the multiclass case:

```
>>> y_pred = [0, 2, 1, 2]
>>> y_true = [0, 1, 2, 2]
>>> jaccard_score(y_true, y_pred, average=None)
...
array([1. , 0. , 0.33...])
```

Examples using `sklearn.metrics.jaccard_score`

- *Classifier Chain*

`sklearn.metrics`.log_loss

`sklearn.metrics.`**`log_loss`**(*y_true*, *y_pred*, *eps=1e-15*, *normalize=True*, *sample_weight=None*, *labels=None*)

Log loss, aka logistic loss or cross-entropy loss.

This is the loss function used in (multinomial) logistic regression and extensions of it such as neural networks, defined as the negative log-likelihood of the true labels given a probabilistic classifier's predictions. The log loss is only defined for two or more labels. For a single sample with true label yt in {0,1} and estimated probability yp that yt = 1, the log loss is

-log P(ytlyp) = -(yt log(yp) + (1 - yt) log(1 - yp))

Read more in the *User Guide*.

Parameters

y_true [array-like or label indicator matrix] Ground truth (correct) labels for n_samples samples.

y_pred [array-like of float, shape = (n_samples, n_classes) or (n_samples,)] Predicted probabilities, as returned by a classifier's predict_proba method. If `y_pred.shape = (n_samples,)` the probabilities provided are assumed to be that of the positive class. The labels in `y_pred` are assumed to be ordered alphabetically, as done by `preprocessing.LabelBinarizer`.

eps [float] Log loss is undefined for p=0 or p=1, so probabilities are clipped to max(eps, min(1 - eps, p)).

normalize [bool, optional (default=True)] If true, return the mean loss per sample. Otherwise, return the sum of the per-sample losses.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

labels [array-like, optional (default=None)] If not provided, labels will be inferred from y_true. If `labels` is `None` and `y_pred` has shape (n_samples,) the labels are assumed to be binary and are inferred from `y_true`. .. versionadded:: 0.18

Returns

loss [float]

Notes

The logarithm used is the natural logarithm (base-e).

References

C.M. Bishop (2006). Pattern Recognition and Machine Learning. Springer, p. 209.

Examples

```
>>> from sklearn.metrics import log_loss
>>> log_loss(["spam", "ham", "ham", "spam"],
...          [[.1, .9], [.9, .1], [.8, .2], [.35, .65]])
0.21616...
```

Examples using `sklearn.metrics.log_loss`

- *Probability Calibration for 3-class classification*
- *Probabilistic predictions with Gaussian process classification (GPC)*

`sklearn.metrics`.matthews_corrcoef

`sklearn.metrics.`**`matthews_corrcoef`**(*y_true*, *y_pred*, *sample_weight=None*)

Compute the Matthews correlation coefficient (MCC)

The Matthews correlation coefficient is used in machine learning as a measure of the quality of binary and multiclass classifications. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes. The MCC is in essence a correlation coefficient value between -1 and +1. A coefficient of +1 represents a perfect prediction, 0 an average

random prediction and -1 an inverse prediction. The statistic is also known as the phi coefficient. [source: Wikipedia]

Binary and multiclass labels are supported. Only in the binary case does this relate to information about true and false positives and negatives. See references below.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples]] Ground truth (correct) target values.

y_pred [array, shape = [n_samples]] Estimated targets as returned by a classifier.

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

Returns

mcc [float] The Matthews correlation coefficient (+1 represents a perfect prediction, 0 an average random prediction and -1 and inverse prediction).

References

[1], *[2]*, *[3]*, *[4]*

Examples

```
>>> from sklearn.metrics import matthews_corrcoef
>>> y_true = [+1, +1, +1, -1]
>>> y_pred = [+1, -1, +1, +1]
>>> matthews_corrcoef(y_true, y_pred)
-0.33...
```

sklearn.metrics.multilabel_confusion_matrix

sklearn.metrics.**multilabel_confusion_matrix**(*y_true*, *y_pred*, *sample_weight=None*, *labels=None*, *samplewise=False*)

Compute a confusion matrix for each class or sample

New in version 0.21.

Compute class-wise (default) or sample-wise (samplewise=True) multilabel confusion matrix to evaluate the accuracy of a classification, and output confusion matrices for each class or sample.

In multilabel confusion matrix MCM, the count of true negatives is $MCM_{:,0,0}$, false negatives is $MCM_{:,1,0}$, true positives is $MCM_{:,1,1}$ and false positives is $MCM_{:,0,1}$.

Multiclass data will be treated as if binarized under a one-vs-rest transformation. Returned confusion matrices will be in the order of sorted unique labels in the union of (y_true, y_pred).

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] of shape (n_samples, n_outputs) or (n_samples,) Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] of shape (n_samples, n_outputs) or (n_samples,) Estimated targets as returned by a classifier

sample_weight [array-like of shape = (n_samples,), optional] Sample weights

labels [array-like] A list of classes or column indices to select some (or to force inclusion of classes absent from the data)

samplewise [bool, default=False] In the multilabel case, this calculates a confusion matrix per sample

Returns

multi_confusion [array, shape (n_outputs, 2, 2)] A 2x2 confusion matrix corresponding to each output in the input. When calculating class-wise multi_confusion (default), then n_outputs = n_labels; when calculating sample-wise multi_confusion (samplewise=True), n_outputs = n_samples. If `labels` is defined, the results will be returned in the order specified in `labels`, otherwise the results will be returned in sorted order by default.

See also:

confusion_matrix

Notes

The multilabel_confusion_matrix calculates class-wise or sample-wise multilabel confusion matrices, and in multiclass tasks, labels are binarized under a one-vs-rest way; while confusion_matrix calculates one confusion matrix for confusion between every two classes.

Examples

Multilabel-indicator case:

```
>>> import numpy as np
>>> from sklearn.metrics import multilabel_confusion_matrix
>>> y_true = np.array([[1, 0, 1],
...                    [0, 1, 0]])
>>> y_pred = np.array([[1, 0, 0],
...                    [0, 1, 1]])
>>> multilabel_confusion_matrix(y_true, y_pred)
array([[[1, 0],
        [0, 1]],

       [[1, 0],
        [0, 1]],

       [[0, 1],
        [1, 0]]])
```

Multiclass case:

```
>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> multilabel_confusion_matrix(y_true, y_pred,
...                             labels=["ant", "bird", "cat"])
array([[[3, 1],
        [0, 2]],

       [[5, 0],
        [1, 0]],
```

```
       [[2, 1],
        [1, 2]]])
```

sklearn.metrics.precision_recall_curve

sklearn.metrics.**precision_recall_curve**(*y_true*, *probas_pred*, *pos_label=None*, *sample_weight=None*)

Compute precision-recall pairs for different probability thresholds

Note: this implementation is restricted to the binary classification task.

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The last precision and recall values are 1. and 0. respectively and do not have a corresponding threshold. This ensures that the graph starts on the y axis.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples]] True binary labels. If labels are not either {-1, 1} or {0, 1}, then pos_label should be explicitly given.

probas_pred [array, shape = [n_samples]] Estimated probabilities or decision function.

pos_label [int or str, default=None] The label of the positive class. When `pos_label=None`, if y_true is in {-1, 1} or {0, 1}, `pos_label` is set to 1, otherwise an error will be raised.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

precision [array, shape = [n_thresholds + 1]] Precision values such that element i is the precision of predictions with score >= thresholds[i] and the last element is 1.

recall [array, shape = [n_thresholds + 1]] Decreasing recall values such that element i is the recall of predictions with score >= thresholds[i] and the last element is 0.

thresholds [array, shape = [n_thresholds <= len(np.unique(probas_pred))]] Increasing thresholds on the decision function used to compute precision and recall.

See also:

average_precision_score Compute average precision from prediction scores

roc_curve Compute Receiver operating characteristic (ROC) curve

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import precision_recall_curve
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
```

```
>>> precision, recall, thresholds = precision_recall_curve(
...     y_true, y_scores)
>>> precision
array([0.66666667, 0.5       , 1.        , 1.        ])
>>> recall
array([1. , 0.5, 0.5, 0. ])
>>> thresholds
array([0.35, 0.4 , 0.8 ])
```

Examples using `sklearn.metrics.precision_recall_curve`

- *Precision-Recall*

`sklearn.metrics.`precision_recall_fscore_support

`sklearn.metrics.`**`precision_recall_fscore_support`**(*y_true*, *y_pred*, *beta=1.0*, *labels=None*, *pos_label=1*, *average=None*, *warn_for=('precision', 'recall', 'f-score')*, *sample_weight=None*)

Compute precision, recall, F-measure and support for each class

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0.

The F-beta score weights recall more than precision by a factor of `beta`. `beta == 1.0` means recall and precision are equally important.

The support is the number of occurrences of each class in `y_true`.

If `pos_label is None` and in binary classification, this function returns the average precision, recall and F-measure if `average` is one of `'micro'`, `'macro'`, `'weighted'` or `'samples'`.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

beta [float, 1.0 by default] The strength of recall versus precision in the F-score.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None (default), 'binary', 'micro', 'macro', 'samples', 'weighted']] If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

`'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

`'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.

`'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

`'weighted'`: Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

`'samples'`: Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *`accuracy_score`*).

warn_for [tuple or set, for internal use] This determines which warnings will be made in the case that this function is being used to return only one of its metrics.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

precision [float (if average is not None) or array of float, shape = [n_unique_labels]]

recall [float (if average is not None) or array of float, , shape = [n_unique_labels]]

fbeta_score [float (if average is not None) or array of float, shape = [n_unique_labels]]

support [int (if average is not None) or array of int, shape = [n_unique_labels]] The number of occurrences of each label in `y_true`.

Notes

When `true positive + false positive == 0`, precision is undefined; When `true positive + false negative == 0`, recall is undefined. In such cases, the metric will be set to 0, as will f-score, and `UndefinedMetricWarning` will be raised.

References

[1], *[2]*, *[3]*

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import precision_recall_fscore_support
>>> y_true = np.array(['cat', 'dog', 'pig', 'cat', 'dog', 'pig'])
>>> y_pred = np.array(['cat', 'pig', 'dog', 'cat', 'cat', 'dog'])
```

```
>>> precision_recall_fscore_support(y_true, y_pred, average='macro')
...
(0.22..., 0.33..., 0.26..., None)
>>> precision_recall_fscore_support(y_true, y_pred, average='micro')
...
(0.33..., 0.33..., 0.33..., None)
>>> precision_recall_fscore_support(y_true, y_pred, average='weighted')
...
(0.22..., 0.33..., 0.26..., None)
```

It is possible to compute per-label precisions, recalls, F1-scores and supports instead of averaging:

```
>>> precision_recall_fscore_support(y_true, y_pred, average=None,
... labels=['pig', 'dog', 'cat'])
...
(array([0.        , 0.        , 0.66...]),
 array([0., 0., 1.]), array([0. , 0. , 0.8]),
 array([2, 2, 2]))
```

sklearn.metrics.precision_score

sklearn.metrics.**precision_score**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

Compute the precision

The precision is the ratio `tp / (tp + fp)` where `tp` is the number of true positives and `fp` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The best value is 1 and the worst value is 0.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

Changed in version 0.17: parameter *labels* improved for multiclass problem.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']] This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

'binary': Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *accuracy_score*).

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

precision [float (if average is not None) or array of float, shape = [n_unique_labels]] Precision of the positive class in binary classification or weighted average of the precision of each class for the multiclass task.

See also:

precision_recall_fscore_support, ***multilabel_confusion_matrix***

Notes

When `true positive + false positive == 0`, precision returns 0 and raises `UndefinedMetricWarning`.

Examples

```
>>> from sklearn.metrics import precision_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> precision_score(y_true, y_pred, average='macro')
0.22...
>>> precision_score(y_true, y_pred, average='micro')
0.33...
>>> precision_score(y_true, y_pred, average='weighted')
...
0.22...
>>> precision_score(y_true, y_pred, average=None)
array([0.66..., 0.        , 0.        ])
```

Examples using `sklearn.metrics.precision_score`

- *Probability Calibration curves*

sklearn.metrics.recall_score

sklearn.metrics.**recall_score**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

Compute the recall

The recall is the ratio `tp / (tp + fn)` where `tp` is the number of true positives and `fn` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The best value is 1 and the worst value is 0.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) target values.

y_pred [1d array-like, or label indicator array / sparse matrix] Estimated targets as returned by a classifier.

labels [list, optional] The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

Changed in version 0.17: parameter *labels* improved for multiclass problem.

pos_label [str or int, 1 by default] The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.

average [string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']] This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

`'binary'`: Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

`'micro'`: Calculate metrics globally by counting the total true positives, false negatives and false positives.

`'macro'`: Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

`'weighted'`: Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

`'samples'`: Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *accuracy_score*).

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

recall [float (if average is not None) or array of float, shape = [n_unique_labels]] Recall of the positive class in binary classification or weighted average of the recall of each class for the multiclass task.

See also:

precision_recall_fscore_support, *balanced_accuracy_score*

multilabel_confusion_matrix

Notes

When `true positive + false negative == 0`, recall returns 0 and raises `UndefinedMetricWarning`.

Examples

```
>>> from sklearn.metrics import recall_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> recall_score(y_true, y_pred, average='macro')
0.33...
>>> recall_score(y_true, y_pred, average='micro')
0.33...
>>> recall_score(y_true, y_pred, average='weighted')
0.33...
>>> recall_score(y_true, y_pred, average=None)
array([1., 0., 0.])
```

Examples using `sklearn.metrics.recall_score`

- *Probability Calibration curves*

`sklearn.metrics.roc_auc_score`

`sklearn.metrics.`**`roc_auc_score`**(*y_true*, *y_score*, *average='macro'*, *sample_weight=None*, *max_fpr=None*)

Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

Note: this implementation is restricted to the binary classification task or multilabel classification task in label indicator format.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples] or [n_samples, n_classes]] True binary labels or binary label indicators.

y_score [array, shape = [n_samples] or [n_samples, n_classes]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers). For binary y_true, y_score is supposed to be the score of the class with greater label.

average [string, [None, 'micro', 'macro' (default), 'samples', 'weighted']] If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

`'micro'`: Calculate metrics globally by considering each element of the label indicator matrix as a label.

'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).

'samples': Calculate metrics for each instance, and find their average.

Will be ignored when `y_true` is binary.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

max_fpr [float > 0 and <= 1, optional] If not `None`, the standardized partial AUC *[3]* over the range [0, max_fpr] is returned.

Returns

auc [float]

See also:

average_precision_score Area under the precision-recall curve

roc_curve Compute Receiver operating characteristic (ROC) curve

References

[1], *[2]*, *[3]*

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import roc_auc_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> roc_auc_score(y_true, y_scores)
0.75
```

sklearn.metrics.roc_curve

sklearn.metrics.**roc_curve**(*y_true*, *y_score*, *pos_label=None*, *sample_weight=None*, *drop_intermediate=True*)

Compute Receiver operating characteristic (ROC)

Note: this implementation is restricted to the binary classification task.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples]] True binary labels. If labels are not either {-1, 1} or {0, 1}, then pos_label should be explicitly given.

y_score [array, shape = [n_samples]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

pos_label [int or str, default=None] The label of the positive class. When `pos_label=None`, if y_true is in {-1, 1} or {0, 1}, `pos_label` is set to 1, otherwise an error will be raised.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

drop_intermediate [boolean, optional (default=True)] Whether to drop some suboptimal thresholds which would not appear on a plotted ROC curve. This is useful in order to create lighter ROC curves.

New in version 0.17: parameter *drop_intermediate*.

Returns

fpr [array, shape = [>2]] Increasing false positive rates such that element i is the false positive rate of predictions with score >= thresholds[i].

tpr [array, shape = [>2]] Increasing true positive rates such that element i is the true positive rate of predictions with score >= thresholds[i].

thresholds [array, shape = [n_thresholds]] Decreasing thresholds on the decision function used to compute fpr and tpr. `thresholds[0]` represents no instances being predicted and is arbitrarily set to `max(y_score) + 1`.

See also:

roc_auc_score Compute the area under the ROC curve

Notes

Since the thresholds are sorted from low to high values, they are reversed upon returning them to ensure they correspond to both `fpr` and `tpr`, which are sorted in reversed order during their calculation.

References

[1], *[2]*

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, scores, pos_label=2)
>>> fpr
array([0. , 0. , 0.5, 0.5, 1. ])
>>> tpr
array([0. , 0.5, 0.5, 1. , 1. ])
>>> thresholds
array([1.8 , 0.8 , 0.4 , 0.35, 0.1 ])
```

Examples using `sklearn.metrics.roc_curve`

- *Species distribution modeling*
- *Feature transformations with ensembles of trees*
- *Receiver Operating Characteristic (ROC) with cross validation*

- *Receiver Operating Characteristic (ROC)*

sklearn.metrics.zero_one_loss

sklearn.metrics.**zero_one_loss**(*y_true*, *y_pred*, *normalize=True*, *sample_weight=None*)

Zero-one classification loss.

If normalize is `True`, return the fraction of misclassifications (float), else it returns the number of misclassifications (int). The best performance is 0.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array / sparse matrix] Predicted labels, as returned by a classifier.

normalize [bool, optional (default=True)] If `False`, return the number of misclassifications. Otherwise, return the fraction of misclassifications.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

loss [float or int,] If `normalize == True`, return the fraction of misclassifications (float), else it returns the number of misclassifications (int).

See also:

accuracy_score*, *hamming_loss*, *jaccard_score

Notes

In multilabel classification, the zero_one_loss function corresponds to the subset zero-one loss: for each sample, the entire set of labels must be correctly predicted, otherwise the loss for that sample is equal to one.

Examples

```
>>> from sklearn.metrics import zero_one_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> zero_one_loss(y_true, y_pred)
0.25
>>> zero_one_loss(y_true, y_pred, normalize=False)
1
```

In the multilabel case with binary label indicators:

```
>>> import numpy as np
>>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
```

Examples using `sklearn.metrics.zero_one_loss`

- *Discrete versus Real AdaBoost*

6.24.3 Regression metrics

See the *Regression metrics* section of the user guide for further details.

metrics.explained_variance_score(y_true, y_pred)	Explained variance regression score function
metrics.max_error(y_true, y_pred)	max_error metric calculates the maximum residual error.
metrics.mean_absolute_error(y_true, y_pred)	Mean absolute error regression loss
metrics.mean_squared_error(y_true, y_pred[, ...])	Mean squared error regression loss
metrics.mean_squared_log_error(y_true, y_pred)	Mean squared logarithmic error regression loss
metrics.median_absolute_error(y_true, y_pred)	Median absolute error regression loss
metrics.r2_score(y_true, y_pred[, ...])	R^2 (coefficient of determination) regression score function.

`sklearn.metrics.`explained_variance_score

sklearn.metrics.**explained_variance_score**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

Explained variance regression score function

Best possible score is 1.0, lower values are worse.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average', 'variance_weighted'] or array-like of shape (n_outputs)] Defines aggregating of multiple output scores. Array-like value defines weights used to average scores.

'raw_values' : Returns a full set of scores in case of multioutput input.

'uniform_average' : Scores of all outputs are averaged with uniform weight.

'variance_weighted' : Scores of all outputs are averaged, weighted by the variances of each individual output.

Returns

score [float or ndarray of floats] The explained variance or ndarray if 'multioutput' is 'raw_values'.

Notes

This is not a symmetric function.

Examples

```
>>> from sklearn.metrics import explained_variance_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> explained_variance_score(y_true, y_pred)
0.957...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> explained_variance_score(y_true, y_pred, multioutput='uniform_average')
...
0.983...
```

sklearn.metrics.max_error

sklearn.metrics.**max_error**(*y_true*, *y_pred*)

max_error metric calculates the maximum residual error.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples)] Estimated target values.

Returns

max_error [float] A positive floating point value (the best value is 0.0).

Examples

```
>>> from sklearn.metrics import max_error
>>> y_true = [3, 2, 7, 1]
>>> y_pred = [4, 2, 7, 1]
>>> max_error(y_true, y_pred)
1
```

sklearn.metrics.mean_absolute_error

sklearn.metrics.**mean_absolute_error**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

Mean absolute error regression loss

Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average']] or array-like of shape (n_outputs) Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors in case of multioutput input.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss [float or ndarray of floats] If multioutput is 'raw_values', then mean absolute error is returned for each output separately. If multioutput is 'uniform_average' or an ndarray of weights, then the weighted average of all output errors is returned.

MAE output is non-negative floating point. The best value is 0.0.

Examples

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_absolute_error(y_true, y_pred)
0.5
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_absolute_error(y_true, y_pred)
0.75
>>> mean_absolute_error(y_true, y_pred, multioutput='raw_values')
array([0.5, 1. ])
>>> mean_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.85...
```

sklearn.metrics.mean_squared_error

sklearn.metrics.**mean_squared_error**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

Mean squared error regression loss

Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average']] or array-like of shape (n_outputs) Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors in case of multioutput input.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss [float or ndarray of floats] A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.

Examples

```
>>> from sklearn.metrics import mean_squared_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred)
0.375
>>> y_true = [[0.5, 1],[-1, 1],[7, -6]]
>>> y_pred = [[0, 2],[-1, 2],[8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.708...
>>> mean_squared_error(y_true, y_pred, multioutput='raw_values')
...
array([0.41666667, 1.        ])
>>> mean_squared_error(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.825...
```

Examples using `sklearn.metrics.mean_squared_error`

- *Model Complexity Influence*
- *Gradient Boosting regression*
- *Plot Ridge coefficients as a function of the L2 regularization*
- *Linear Regression Example*
- *Robust linear estimator fitting*

`sklearn.metrics`.mean_squared_log_error

`sklearn.metrics.`**`mean_squared_log_error`**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

Mean squared logarithmic error regression loss

Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average'] or array-like of shape = (n_outputs)] Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors when the input is of multioutput format.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss [float or ndarray of floats] A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.

Examples

```
>>> from sklearn.metrics import mean_squared_log_error
>>> y_true = [3, 5, 2.5, 7]
>>> y_pred = [2.5, 5, 4, 8]
>>> mean_squared_log_error(y_true, y_pred)
0.039...
>>> y_true = [[0.5, 1], [1, 2], [7, 6]]
>>> y_pred = [[0.5, 2], [1, 2.5], [8, 8]]
>>> mean_squared_log_error(y_true, y_pred)
0.044...
>>> mean_squared_log_error(y_true, y_pred, multioutput='raw_values')
...
array([0.00462428, 0.08377444])
>>> mean_squared_log_error(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.060...
```

sklearn.metrics.median_absolute_error

sklearn.metrics.**median_absolute_error**(*y_true*, *y_pred*)

Median absolute error regression loss

Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples)] Estimated target values.

Returns

loss [float] A positive floating point value (the best value is 0.0).

Examples

```
>>> from sklearn.metrics import median_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> median_absolute_error(y_true, y_pred)
0.5
```

Examples using sklearn.metrics.median_absolute_error

- *Effect of transforming the targets in regression model*

sklearn.metrics.r2_score

sklearn.metrics.**r2_score**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

R^2 (coefficient of determination) regression score function.

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.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average', 'variance_weighted'] or None or array-like of shape (n_outputs)] Defines aggregating of multiple output scores. Array-like value defines weights used to average scores. Default is "uniform_average".

'raw_values' : Returns a full set of scores in case of multioutput input.

'uniform_average' : Scores of all outputs are averaged with uniform weight.

'variance_weighted' : Scores of all outputs are averaged, weighted by the variances of each individual output.

Changed in version 0.19: Default value of multioutput is 'uniform_average'.

Returns

z [float or ndarray of floats] The R^2 score or ndarray of scores if 'multioutput' is 'raw_values'.

Notes

This is not a symmetric function.

Unlike most other scores, R^2 score may be negative (it need not actually be the square of a quantity R).

This metric is not well-defined for single samples and will return a NaN value if n_samples is less than two.

References

[1]

Examples

```
>>> from sklearn.metrics import r2_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> r2_score(y_true, y_pred)
0.948...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred,
...          multioutput='variance_weighted')
```

```
0.938...
>>> y_true = [1, 2, 3]
>>> y_pred = [1, 2, 3]
>>> r2_score(y_true, y_pred)
1.0
>>> y_true = [1, 2, 3]
>>> y_pred = [2, 2, 2]
>>> r2_score(y_true, y_pred)
0.0
>>> y_true = [1, 2, 3]
>>> y_pred = [3, 2, 1]
>>> r2_score(y_true, y_pred)
-3.0
```

Examples using `sklearn.metrics.r2_score`

- *Effect of transforming the targets in regression model*
- *Linear Regression Example*
- *Lasso and Elastic Net for Sparse Signals*

6.24.4 Multilabel ranking metrics

See the *Multilabel ranking metrics* section of the user guide for further details.

metrics.coverage_error(y_true, y_score[, ...])	Coverage error measure
metrics.label_ranking_average_precision_score(y_true,)	Compute ranking-based average precision
metrics.label_ranking_loss(y_true, y_score)	Compute Ranking loss measure

`sklearn.metrics.coverage_error`

sklearn.metrics.**coverage_error**(*y_true*, *y_score*, *sample_weight=None*)

Coverage error measure

Compute how far we need to go through the ranked scores to cover all true labels. The best value is equal to the average number of labels in `y_true` per sample.

Ties in `y_scores` are broken by giving maximal rank that would have been assigned to all tied values.

Note: Our implementation's score is 1 greater than the one given in Tsoumakas et al., 2010. This extends it to handle the degenerate case in which an instance has 0 true labels.

Read more in the *User Guide*.

Parameters

y_true [array, shape = [n_samples, n_labels]] True binary labels in binary indicator format.

y_score [array, shape = [n_samples, n_labels]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

coverage_error [float]

References

[1]

sklearn.metrics.label_ranking_average_precision_score

sklearn.metrics.**label_ranking_average_precision_score**(*y_true*, *y_score*, *sample_weight=None*)

Compute ranking-based average precision

Label ranking average precision (LRAP) is the average over each ground truth label assigned to each sample, of the ratio of true vs. total labels with lower score.

This metric is used in multilabel ranking problem, where the goal is to give better rank to the labels associated to each sample.

The obtained score is always strictly greater than 0 and the best value is 1.

Read more in the *User Guide*.

Parameters

y_true [array or sparse matrix, shape = [n_samples, n_labels]] True binary labels in binary indicator format.

y_score [array, shape = [n_samples, n_labels]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

score [float]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import label_ranking_average_precision_score
>>> y_true = np.array([[1, 0, 0], [0, 0, 1]])
>>> y_score = np.array([[0.75, 0.5, 1], [1, 0.2, 0.1]])
>>> label_ranking_average_precision_score(y_true, y_score)
0.416...
```

sklearn.metrics.label_ranking_loss

sklearn.metrics.**label_ranking_loss**(*y_true*, *y_score*, *sample_weight=None*)

Compute Ranking loss measure

Compute the average number of label pairs that are incorrectly ordered given y_score weighted by the size of the label set and the number of labels not in the label set.

This is similar to the error set size, but weighted by the number of relevant and irrelevant labels. The best performance is achieved with a ranking loss of zero.

Read more in the *User Guide*.

New in version 0.17: A function *label_ranking_loss*

Parameters

y_true [array or sparse matrix, shape = [n_samples, n_labels]] True binary labels in binary indicator format.

y_score [array, shape = [n_samples, n_labels]] Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

loss [float]

References

[1]

6.24.5 Clustering metrics

See the *Clustering performance evaluation* section of the user guide for further details. The `sklearn.metrics.cluster` submodule contains evaluation metrics for cluster analysis results. There are two forms of evaluation:

- supervised, which uses a ground truth class values for each sample.
- unsupervised, which does not and measures the 'quality' of the model itself.

`metrics.adjusted_mutual_info_score`(... [, ...])	Adjusted Mutual Information between two clusterings.
`metrics.adjusted_rand_score`(labels_true, ...)	Rand index adjusted for chance.
`metrics.calinski_harabasz_score`(X, labels)	Compute the Calinski and Harabasz score.
`metrics.davies_bouldin_score`(X, labels)	Computes the Davies-Bouldin score.
`metrics.completeness_score`(labels_true, ...)	Completeness metric of a cluster labeling given a ground truth.
`metrics.cluster.contingency_matrix`(... [, ...])	Build a contingency matrix describing the relationship between labels.
`metrics.fowlkes_mallows_score`(labels_true, ...)	Measure the similarity of two clusterings of a set of points.
`metrics.homogeneity_completeness_v_measure`(...)	Compute the homogeneity and completeness and V-Measure scores at once.
`metrics.homogeneity_score`(labels_true, ...)	Homogeneity metric of a cluster labeling given a ground truth.
`metrics.mutual_info_score`(labels_true, ...)	Mutual Information between two clusterings.
`metrics.normalized_mutual_info_score`(... [, ...])	Normalized Mutual Information between two clusterings.
`metrics.silhouette_score`(X, labels[, ...])	Compute the mean Silhouette Coefficient of all samples.
`metrics.silhouette_samples`(X, labels[, metric])	Compute the Silhouette Coefficient for each sample.
`metrics.v_measure_score`(labels_true, labels_pred)	V-measure cluster labeling given a ground truth.

sklearn.metrics.adjusted_mutual_info_score

sklearn.metrics.**adjusted_mutual_info_score**(*labels_true*, *labels_pred*, *average_method='warn'*)

Adjusted Mutual Information between two clusterings.

Adjusted Mutual Information (AMI) is an adjustment of the Mutual Information (MI) score to account for chance. It accounts for the fact that the MI is generally higher for two clusterings with a larger number of clusters, regardless of whether there is actually more information shared. For two clusterings U and V, the AMI is given as:

```
AMI(U, V) = [MI(U, V) - E(MI(U, V))] / [avg(H(U), H(V)) - E(MI(U, V))]
```

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Be mindful that this function is an order of magnitude slower than other metrics, such as the Adjusted Rand Index.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] A clustering of the data into disjoint subsets.

labels_pred [array, shape = [n_samples]] A clustering of the data into disjoint subsets.

average_method [string, optional (default: 'warn')] How to compute the normalizer in the denominator. Possible options are 'min', 'geometric', 'arithmetic', and 'max'. If 'warn', 'max' will be used. The default will change to 'arithmetic' in version 0.22.

New in version 0.20.

Returns

ami: float (upperlimited by 1.0) The AMI returns a value of 1 when the two partitions are identical (ie perfectly matched). Random partitions (independent labellings) have an expected AMI around 0 on average hence can be negative.

See also:

adjusted_rand_score Adjusted Rand Index

mutual_info_score Mutual Information (not adjusted for chance)

References

[1], *[2]*

Examples

Perfect labelings are both homogeneous and complete, hence have score 1.0:

```
>>> from sklearn.metrics.cluster import adjusted_mutual_info_score
>>> adjusted_mutual_info_score([0, 0, 1, 1], [0, 0, 1, 1])
...
1.0
>>> adjusted_mutual_info_score([0, 0, 1, 1], [1, 1, 0, 0])
...
1.0
```

If classes members are completely split across different clusters, the assignment is totally in-complete, hence the AMI is null:

```
>>> adjusted_mutual_info_score([0, 0, 0, 0], [0, 1, 2, 3])
...
0.0
```

Examples using `sklearn.metrics.adjusted_mutual_info_score`

- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *Adjustment for chance in clustering performance evaluation*
- *A demo of K-Means clustering on the handwritten digits data*

`sklearn.metrics.adjusted_rand_score`

sklearn.metrics.**adjusted_rand_score**(*labels_true*, *labels_pred*)

Rand index adjusted for chance.

The Rand Index computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.

The raw RI score is then "adjusted for chance" into the ARI score using the following scheme:

```
ARI = (RI - Expected_RI) / (max(RI) - Expected_RI)
```

The adjusted Rand index is thus ensured to have a value close to 0.0 for random labeling independently of the number of clusters and samples and exactly 1.0 when the clusterings are identical (up to a permutation).

ARI is a symmetric measure:

```
adjusted_rand_score(a, b) == adjusted_rand_score(b, a)
```

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] Ground truth class labels to be used as a reference

labels_pred [array, shape = [n_samples]] Cluster labels to evaluate

Returns

ari [float] Similarity score between -1.0 and 1.0. Random labelings have an ARI close to 0.0. 1.0 stands for perfect match.

See also:

adjusted_mutual_info_score Adjusted Mutual Information

References

[Hubert1985], *[wk]*

Examples

Perfectly matching labelings have a score of 1 even

```
>>> from sklearn.metrics.cluster import adjusted_rand_score
>>> adjusted_rand_score([0, 0, 1, 1], [0, 0, 1, 1])
1.0
>>> adjusted_rand_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

Labelings that assign all classes members to the same clusters are complete be not always pure, hence penalized:

```
>>> adjusted_rand_score([0, 0, 1, 2], [0, 0, 1, 1])
0.57...
```

ARI is symmetric, so labelings that have pure clusters with members coming from the same classes but unnecessary splits are penalized:

```
>>> adjusted_rand_score([0, 0, 1, 1], [0, 0, 1, 2])
0.57...
```

If classes members are completely split across different clusters, the assignment is totally incomplete, hence the ARI is very low:

```
>>> adjusted_rand_score([0, 0, 0, 0], [0, 1, 2, 3])
0.0
```

Examples using `sklearn.metrics.adjusted_rand_score`

- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *Adjustment for chance in clustering performance evaluation*
- *A demo of K-Means clustering on the handwritten digits data*
- *Clustering text documents using k-means*

`sklearn.metrics`.calinski_harabasz_score

sklearn.metrics.**calinski_harabasz_score**(*X*, *labels*)

Compute the Calinski and Harabasz score.

It is also known as the Variance Ratio Criterion.

The score is defined as ratio between the within-cluster dispersion and the between-cluster dispersion.

Read more in the *User Guide*.

Parameters

X [array-like, shape (`n_samples`, `n_features`)] List of `n_features`-dimensional data points. Each row corresponds to a single data point.

labels [array-like, shape (`n_samples`,)] Predicted labels for each sample.

Returns

score [float] The resulting Calinski-Harabasz score.

References

[1]

sklearn.metrics.davies_bouldin_score

sklearn.metrics.**davies_bouldin_score**(*X*, *labels*)

Computes the Davies-Bouldin score.

The score is defined as the average similarity measure of each cluster with its most similar cluster, where similarity is the ratio of within-cluster distances to between-cluster distances. Thus, clusters which are farther apart and less dispersed will result in a better score.

The minimum score is zero, with lower values indicating better clustering.

Read more in the *User Guide*.

Parameters

X [array-like, shape (`n_samples`, `n_features`)] List of `n_features`-dimensional data points. Each row corresponds to a single data point.

labels [array-like, shape (`n_samples`,)] Predicted labels for each sample.

Returns

score: float The resulting Davies-Bouldin score.

References

[1]

sklearn.metrics.completeness_score

sklearn.metrics.**completeness_score**(*labels_true*, *labels_pred*)

Completeness metric of a cluster labeling given a ground truth.

A clustering result satisfies completeness if all the data points that are members of a given class are elements of the same cluster.

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is not symmetric: switching `label_true` with `label_pred` will return the *`homogeneity_score`* which will be different in general.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] ground truth class labels to be used as a reference

labels_pred [array, shape = [n_samples]] cluster labels to evaluate

Returns

completeness [float] score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

See also:

homogeneity_score

v_measure_score

References

[1]

Examples

Perfect labelings are complete:

```
>>> from sklearn.metrics.cluster import completeness_score
>>> completeness_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

Non-perfect labelings that assign all classes members to the same clusters are still complete:

```
>>> print(completeness_score([0, 0, 1, 1], [0, 0, 0, 0]))
1.0
>>> print(completeness_score([0, 1, 2, 3], [0, 0, 1, 1]))
0.999...
```

If classes members are split across different clusters, the assignment cannot be complete:

```
>>> print(completeness_score([0, 0, 1, 1], [0, 1, 0, 1]))
0.0
>>> print(completeness_score([0, 0, 0, 0], [0, 1, 2, 3]))
0.0
```

Examples using `sklearn.metrics.completeness_score`

- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *A demo of K-Means clustering on the handwritten digits data*
- *Clustering text documents using k-means*

sklearn.metrics.cluster.contingency_matrix

sklearn.metrics.cluster.**contingency_matrix**(*labels_true*, *labels_pred*, *eps=None*, *sparse=False*)

Build a contingency matrix describing the relationship between labels.

Parameters

labels_true [int array, shape = [n_samples]] Ground truth class labels to be used as a reference

labels_pred [array, shape = [n_samples]] Cluster labels to evaluate

eps [None or float, optional.] If a float, that value is added to all values in the contingency matrix. This helps to stop NaN propagation. If `None`, nothing is adjusted.

sparse [boolean, optional.] If True, return a sparse CSR continency matrix. If `eps is not None`, and `sparse is True`, will throw ValueError.

New in version 0.18.

Returns

contingency [{array-like, sparse}, shape=[n_classes_true, n_classes_pred]] Matrix C such that $C_{i,j}$ is the number of samples in true class i and in predicted class j. If `eps is None`, the dtype of this array will be integer. If `eps` is given, the dtype will be float. Will be a `scipy.sparse.csr_matrix` if `sparse=True`.

sklearn.metrics.fowlkes_mallows_score

sklearn.metrics.**fowlkes_mallows_score**(*labels_true*, *labels_pred*, *sparse=False*)

Measure the similarity of two clusterings of a set of points.

The Fowlkes-Mallows index (FMI) is defined as the geometric mean between of the precision and recall:

```
FMI = TP / sqrt((TP + FP) * (TP + FN))
```

Where `TP` is the number of **True Positive** (i.e. the number of pair of points that belongs in the same clusters in both `labels_true` and `labels_pred`), `FP` is the number of **False Positive** (i.e. the number of pair of points that belongs in the same clusters in `labels_true` and not in `labels_pred`) and `FN` is the number of **False Negative** (i.e the number of pair of points that belongs in the same clusters in `labels_pred` and not in `labels_True`).

The score ranges from 0 to 1. A high value indicates a good similarity between two clusters.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = (`n_samples`,)] A clustering of the data into disjoint subsets.

labels_pred [array, shape = (`n_samples`,)] A clustering of the data into disjoint subsets.

sparse [bool] Compute contingency matrix internally with sparse matrix.

Returns

score [float] The resulting Fowlkes-Mallows score.

References

[1], *[2]*

Examples

Perfect labelings are both homogeneous and complete, hence have score 1.0:

```
>>> from sklearn.metrics.cluster import fowlkes_mallows_score
>>> fowlkes_mallows_score([0, 0, 1, 1], [0, 0, 1, 1])
1.0
>>> fowlkes_mallows_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

If classes members are completely split across different clusters, the assignment is totally random, hence the FMI is null:

```
>>> fowlkes_mallows_score([0, 0, 0, 0], [0, 1, 2, 3])
0.0
```

sklearn.metrics.homogeneity_completeness_v_measure

sklearn.metrics.**homogeneity_completeness_v_measure**(*labels_true*, *labels_pred*, *beta=1.0*)

Compute the homogeneity and completeness and V-Measure scores at once.

Those metrics are based on normalized conditional entropy measures of the clustering labeling to evaluate given the knowledge of a Ground Truth class labels of the same samples.

A clustering result satisfies homogeneity if all of its clusters contain only data points which are members of a single class.

A clustering result satisfies completeness if all the data points that are members of a given class are elements of the same cluster.

Both scores have positive values between 0.0 and 1.0, larger values being desirable.

Those 3 metrics are independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score values in any way.

V-Measure is furthermore symmetric: swapping `labels_true` and `label_pred` will give the same score. This does not hold for homogeneity and completeness. V-Measure is identical to *`normalized_mutual_info_score`* with the arithmetic averaging method.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] ground truth class labels to be used as a reference

labels_pred [array, shape = [n_samples]] cluster labels to evaluate

beta [float] Ratio of weight attributed to `homogeneity` vs `completeness`. If `beta` is greater than 1, `completeness` is weighted more strongly in the calculation. If `beta` is less than 1, `homogeneity` is weighted more strongly.

Returns

homogeneity [float] score between 0.0 and 1.0. 1.0 stands for perfectly homogeneous labeling

completeness [float] score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

v_measure [float] harmonic mean of the first two

See also:

homogeneity_score

completeness_score

v_measure_score

sklearn.metrics.homogeneity_score

sklearn.metrics.**homogeneity_score**(*labels_true*, *labels_pred*)

Homogeneity metric of a cluster labeling given a ground truth.

A clustering result satisfies homogeneity if all of its clusters contain only data points which are members of a single class.

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is not symmetric: switching `label_true` with `label_pred` will return the *completeness_score* which will be different in general.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] ground truth class labels to be used as a reference

labels_pred [array, shape = [n_samples]] cluster labels to evaluate

Returns

homogeneity [float] score between 0.0 and 1.0. 1.0 stands for perfectly homogeneous labeling

See also:

completeness_score

v_measure_score

References

[1]

Examples

Perfect labelings are homogeneous:

```
>>> from sklearn.metrics.cluster import homogeneity_score
>>> homogeneity_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

Non-perfect labelings that further split classes into more clusters can be perfectly homogeneous:

```
>>> print("%.6f" % homogeneity_score([0, 0, 1, 1], [0, 0, 1, 2]))
...
1.000000
>>> print("%.6f" % homogeneity_score([0, 0, 1, 1], [0, 1, 2, 3]))
...
1.000000
```

Clusters that include samples from different classes do not make for an homogeneous labeling:

```
>>> print("%.6f" % homogeneity_score([0, 0, 1, 1], [0, 1, 0, 1]))
...
0.0...
>>> print("%.6f" % homogeneity_score([0, 0, 1, 1], [0, 0, 0, 0]))
...
0.0...
```

Examples using `sklearn.metrics.homogeneity_score`

- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *A demo of K-Means clustering on the handwritten digits data*
- *Clustering text documents using k-means*

`sklearn.metrics.mutual_info_score`

`sklearn.metrics.`**`mutual_info_score`**(*labels_true*, *labels_pred*, *contingency=None*)

Mutual Information between two clusterings.

The Mutual Information is a measure of the similarity between two labels of the same data. Where $|U_i|$ is the number of the samples in cluster U_i and $|V_j|$ is the number of the samples in cluster V_j, the Mutual Information between clusterings U and V is given as:

$$MI(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} \frac{|U_i \cap V_j|}{N} \log \frac{N|U_i \cap V_j|}{|U_i||V_j|}$$

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] A clustering of the data into disjoint subsets.

labels_pred [array, shape = [n_samples]] A clustering of the data into disjoint subsets.

contingency [{None, array, sparse matrix}, shape = [n_classes_true, n_classes_pred]] A contingency matrix given by the `contingency_matrix` function. If value is `None`, it will be computed, otherwise the given value is used, with `labels_true` and `labels_pred` ignored.

Returns

mi [float] Mutual information, a non-negative value

See also:

adjusted_mutual_info_score Adjusted against chance Mutual Information

normalized_mutual_info_score Normalized Mutual Information

Examples using `sklearn.metrics.mutual_info_score`

- *Adjustment for chance in clustering performance evaluation*

sklearn.metrics.normalized_mutual_info_score

sklearn.metrics.**normalized_mutual_info_score**(*labels_true*, *labels_pred*, *average_method='warn'*)

Normalized Mutual Information between two clusterings.

Normalized Mutual Information (NMI) is a normalization of the Mutual Information (MI) score to scale the results between 0 (no mutual information) and 1 (perfect correlation). In this function, mutual information is normalized by some generalized mean of `H(labels_true)` and `H(labels_pred))`, defined by the `average_method`.

This measure is not adjusted for chance. Therefore *adjusted_mutual_info_score* might be preferred.

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] A clustering of the data into disjoint subsets.

labels_pred [array, shape = [n_samples]] A clustering of the data into disjoint subsets.

average_method [string, optional (default: 'warn')] How to compute the normalizer in the denominator. Possible options are 'min', 'geometric', 'arithmetic', and 'max'. If 'warn', 'geometric' will be used. The default will change to 'arithmetic' in version 0.22.

New in version 0.20.

Returns

nmi [float] score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

See also:

v_measure_score V-Measure (NMI with arithmetic mean option.)

adjusted_rand_score Adjusted Rand Index

adjusted_mutual_info_score Adjusted Mutual Information (adjusted against chance)

Examples

Perfect labelings are both homogeneous and complete, hence have score 1.0:

```
>>> from sklearn.metrics.cluster import normalized_mutual_info_score
>>> normalized_mutual_info_score([0, 0, 1, 1], [0, 0, 1, 1])
...
1.0
>>> normalized_mutual_info_score([0, 0, 1, 1], [1, 1, 0, 0])
```

```
...
1.0
```

If classes members are completely split across different clusters, the assignment is totally in-complete, hence the NMI is null:

```
>>> normalized_mutual_info_score([0, 0, 0, 0], [0, 1, 2, 3])
...
0.0
```

sklearn.metrics.silhouette_score

sklearn.metrics.**silhouette_score**(*X*, *labels*, *metric='euclidean'*, *sample_size=None*, *random_state=None*, ***kwds*)

Compute the mean Silhouette Coefficient of all samples.

The Silhouette Coefficient is calculated using the mean intra-cluster distance (`a`) and the mean nearest-cluster distance (`b`) for each sample. The Silhouette Coefficient for a sample is `(b - a) / max(a, b)`. To clarify, `b` is the distance between a sample and the nearest cluster that the sample is not a part of. Note that Silhouette Coefficient is only defined if number of labels is 2 <= n_labels <= n_samples - 1.

This function returns the mean Silhouette Coefficient over all samples. To obtain the values for each sample, use *silhouette_samples*.

The best value is 1 and the worst value is -1. Values near 0 indicate overlapping clusters. Negative values generally indicate that a sample has been assigned to the wrong cluster, as a different cluster is more similar.

Read more in the *User Guide*.

Parameters

X [array [n_samples_a, n_samples_a] if metric == "precomputed", or, [n_samples_a, n_features] otherwise] Array of pairwise distances between samples, or a feature array.

labels [array, shape = [n_samples]] Predicted labels for each sample.

metric [string, or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by `metrics.pairwise.pairwise_distances`. If X is the distance array itself, use `metric="precomputed"`.

sample_size [int or None] The size of the sample to use when computing the Silhouette Coefficient on a random subset of the data. If `sample_size is None`, no sampling is used.

random_state [int, RandomState instance or None, optional (default=None)] The generator used to randomly select a subset of samples. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `sample_size is not None`.

****kwds** [optional keyword parameters] Any further parameters are passed directly to the distance function. If using a scipy.spatial.distance metric, the parameters are still metric dependent. See the scipy docs for usage examples.

Returns

silhouette [float] Mean Silhouette Coefficient for all samples.

References

[1], *[2]*

Examples using `sklearn.metrics.silhouette_score`

- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *A demo of K-Means clustering on the handwritten digits data*
- *Selecting the number of clusters with silhouette analysis on KMeans clustering*
- *Clustering text documents using k-means*

`sklearn.metrics`.silhouette_samples

`sklearn.metrics.`**`silhouette_samples`**(*X*, *labels*, *metric='euclidean'*, ***kwds*)

Compute the Silhouette Coefficient for each sample.

The Silhouette Coefficient is a measure of how well samples are clustered with samples that are similar to themselves. Clustering models with a high Silhouette Coefficient are said to be dense, where samples in the same cluster are similar to each other, and well separated, where samples in different clusters are not very similar to each other.

The Silhouette Coefficient is calculated using the mean intra-cluster distance (`a`) and the mean nearest-cluster distance (`b`) for each sample. The Silhouette Coefficient for a sample is `(b - a) / max(a, b)`. Note that Silhouette Coefficient is only defined if number of labels is 2 <= n_labels <= n_samples - 1.

This function returns the Silhouette Coefficient for each sample.

The best value is 1 and the worst value is -1. Values near 0 indicate overlapping clusters.

Read more in the *User Guide*.

Parameters

X [array [n_samples_a, n_samples_a] if metric == "precomputed", or, [n_samples_a, n_features] otherwise] Array of pairwise distances between samples, or a feature array.

labels [array, shape = [n_samples]] label values for each sample

metric [string, or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by `sklearn.metrics.pairwise.pairwise_distances`. If X is the distance array itself, use "precomputed" as the metric.

'kwds'** [optional keyword parameters] Any further parameters are passed directly to the distance function. If using a `scipy.spatial.distance` metric, the parameters are still metric dependent. See the scipy docs for usage examples.

Returns

silhouette [array, shape = [n_samples]] Silhouette Coefficient for each samples.

References

[1], *[2]*

Examples using `sklearn.metrics.silhouette_samples`

- *Selecting the number of clusters with silhouette analysis on KMeans clustering*

sklearn.metrics.v_measure_score

sklearn.metrics.**v_measure_score**(*labels_true*, *labels_pred*, *beta=1.0*)

V-measure cluster labeling given a ground truth.

This score is identical to *normalized_mutual_info_score* with the `'arithmetic'` option for averaging.

The V-measure is the harmonic mean between homogeneity and completeness:

```
v = (1 + beta) * homogeneity * completeness
     / (beta * homogeneity + completeness)
```

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.

This metric is furthermore symmetric: switching `label_true` with `label_pred` will return the same score value. This can be useful to measure the agreement of two independent label assignments strategies on the same dataset when the real ground truth is not known.

Read more in the *User Guide*.

Parameters

labels_true [int array, shape = [n_samples]] ground truth class labels to be used as a reference

labels_pred [array, shape = [n_samples]] cluster labels to evaluate

beta [float] Ratio of weight attributed to `homogeneity` vs `completeness`. If `beta` is greater than 1, `completeness` is weighted more strongly in the calculation. If `beta` is less than 1, `homogeneity` is weighted more strongly.

Returns

v_measure [float] score between 0.0 and 1.0. 1.0 stands for perfectly complete labeling

See also:

homogeneity_score

completeness_score

normalized_mutual_info_score

References

[1]

Examples

Perfect labelings are both homogeneous and complete, hence have score 1.0:

```
>>> from sklearn.metrics.cluster import v_measure_score
>>> v_measure_score([0, 0, 1, 1], [0, 0, 1, 1])
1.0
>>> v_measure_score([0, 0, 1, 1], [1, 1, 0, 0])
1.0
```

Labelings that assign all classes members to the same clusters are complete be not homogeneous, hence penalized:

```
>>> print("%.6f" % v_measure_score([0, 0, 1, 2], [0, 0, 1, 1]))
...
0.8...
>>> print("%.6f" % v_measure_score([0, 1, 2, 3], [0, 0, 1, 1]))
...
0.66...
```

Labelings that have pure clusters with members coming from the same classes are homogeneous but unnecessary splits harms completeness and thus penalize V-measure as well:

```
>>> print("%.6f" % v_measure_score([0, 0, 1, 1], [0, 0, 1, 2]))
...
0.8...
>>> print("%.6f" % v_measure_score([0, 0, 1, 1], [0, 1, 2, 3]))
...
0.66...
```

If classes members are completely split across different clusters, the assignment is totally incomplete, hence the V-Measure is null:

```
>>> print("%.6f" % v_measure_score([0, 0, 0, 0], [0, 1, 2, 3]))
...
0.0...
```

Clusters that include samples from totally different classes totally destroy the homogeneity of the labeling, hence:

```
>>> print("%.6f" % v_measure_score([0, 0, 1, 1], [0, 0, 0, 0]))
...
0.0...
```

Examples using `sklearn.metrics.v_measure_score`

- *Biclustering documents with the Spectral Co-clustering algorithm*
- *Demo of affinity propagation clustering algorithm*
- *Demo of DBSCAN clustering algorithm*
- *Adjustment for chance in clustering performance evaluation*
- *A demo of K-Means clustering on the handwritten digits data*
- *Clustering text documents using k-means*

6.24.6 Biclustering metrics

See the *Biclustering evaluation* section of the user guide for further details.

metrics.consensus_score(a, b[, similarity])	The similarity of two sets of biclusters.

sklearn.metrics.consensus_score

sklearn.metrics.**consensus_score**(*a*, *b*, *similarity='jaccard'*)

The similarity of two sets of biclusters.

Similarity between individual biclusters is computed. Then the best matching between sets is found using the Hungarian algorithm. The final score is the sum of similarities divided by the size of the larger set.

Read more in the *User Guide*.

Parameters

a [(rows, columns)] Tuple of row and column indicators for a set of biclusters.

b [(rows, columns)] Another set of biclusters like `a`.

similarity [string or function, optional, default: "jaccard"] May be the string "jaccard" to use the Jaccard coefficient, or any function that takes four arguments, each of which is a 1d indicator vector: (a_rows, a_columns, b_rows, b_columns).

References

- Hochreiter, Bodenhofer, et. al., 2010. FABIA: factor analysis for bicluster acquisition.

Examples using sklearn.metrics.consensus_score

- *A demo of the Spectral Co-Clustering algorithm*
- *A demo of the Spectral Biclustering algorithm*

6.24.7 Pairwise metrics

See the *Pairwise metrics, Affinities and Kernels* section of the user guide for further details.

metrics.pairwise.additive_chi2_kernel(X[, Y])	Computes the additive chi-squared kernel between observations in X and Y
metrics.pairwise.chi2_kernel(X[, Y, gamma])	Computes the exponential chi-squared kernel X and Y.
metrics.pairwise.cosine_similarity(X[, Y, ...])	Compute cosine similarity between samples in X and Y.
metrics.pairwise.cosine_distances(X[, Y])	Compute cosine distance between samples in X and Y.
metrics.pairwise.distance_metrics()	Valid metrics for pairwise_distances.
metrics.pairwise.euclidean_distances(X[, Y, ...])	Considering the rows of X (and Y=X) as vectors, compute the distance matrix between each pair of vectors.
metrics.pairwise.haversine_distances(X[, Y])	Compute the Haversine distance between samples in X and Y
metrics.pairwise.kernel_metrics()	Valid metrics for pairwise_kernels
metrics.pairwise.laplacian_kernel(X[, Y, gamma])	Compute the laplacian kernel between X and Y.
metrics.pairwise.linear_kernel(X[, Y, ...])	Compute the linear kernel between X and Y.

Continued on next page

Table 6.184 – continued from previous page

metrics.pairwise.manhattan_distances(X[, Y, ...])	Compute the L1 distances between the vectors in X and Y.
metrics.pairwise.pairwise_kernels(X[, Y, ...])	Compute the kernel between arrays X and optional array Y.
metrics.pairwise.polynomial_kernel(X[, Y, ...])	Compute the polynomial kernel between X and Y:
metrics.pairwise.rbf_kernel(X[, Y, gamma])	Compute the rbf (gaussian) kernel between X and Y:
metrics.pairwise.sigmoid_kernel(X[, Y, ...])	Compute the sigmoid kernel between X and Y:
metrics.pairwise.paired_euclidean_distances(X, Y)	Computes the paired euclidean distances between X and Y
metrics.pairwise.paired_manhattan_distances(X, Y)	Compute the L1 distances between the vectors in X and Y.
metrics.pairwise.paired_cosine_distances(X, Y)	Computes the paired cosine distances between X and Y
metrics.pairwise.paired_distances(X, Y[, metric])	Computes the paired distances between X and Y.
metrics.pairwise_distances(X[, Y, metric, ...])	Compute the distance matrix from a vector array X and optional Y.
metrics.pairwise_distances_argmin(X, Y[, ...])	Compute minimum distances between one point and a set of points.
metrics.pairwise_distances_argmin_min(X, Y)	Compute minimum distances between one point and a set of points.
metrics.pairwise_distances_chunked(X[, Y, ...])	Generate a distance matrix chunk by chunk with optional reduction

sklearn.metrics.pairwise.additive_chi2_kernel

sklearn.metrics.pairwise.**additive_chi2_kernel**(*X*, *Y=None*)

Computes the additive chi-squared kernel between observations in X and Y

The chi-squared kernel is computed between each pair of rows in X and Y. X and Y have to be non-negative. This kernel is most commonly applied to histograms.

The chi-squared kernel is given by:

```
k(x, y) = -Sum [(x - y)^2 / (x + y)]
```

It can be interpreted as a weighted difference per entry.

Read more in the *User Guide*.

Parameters

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

Y [array of shape (n_samples_Y, n_features)]

Returns

kernel_matrix [array of shape (n_samples_X, n_samples_Y)]

See also:

chi2_kernel The exponentiated version of the kernel, which is usually preferable.

sklearn.kernel_approximation.AdditiveChi2Sampler A Fourier approximation to this kernel.

Notes

As the negative of a distance, this kernel is only conditionally positive definite.

References

- Zhang, J. and Marszalek, M. and Lazebnik, S. and Schmid, C. Local features and kernels for classification of texture and object categories: A comprehensive study International Journal of Computer Vision 2007 https://research.microsoft.com/en-us/um/people/manik/projects/trade-off/papers/ZhangIJCV06.pdf

sklearn.metrics.pairwise.chi2_kernel

sklearn.metrics.pairwise.**chi2_kernel**(*X*, *Y=None*, *gamma=1.0*)

Computes the exponential chi-squared kernel X and Y.

The chi-squared kernel is computed between each pair of rows in X and Y. X and Y have to be non-negative. This kernel is most commonly applied to histograms.

The chi-squared kernel is given by:

```
k(x, y) = exp(-gamma Sum [(x - y)^2 / (x + y)])
```

It can be interpreted as a weighted difference per entry.

Read more in the *User Guide*.

Parameters

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

Y [array of shape (n_samples_Y, n_features)]

gamma [float, default=1.] Scaling parameter of the chi2 kernel.

Returns

kernel_matrix [array of shape (n_samples_X, n_samples_Y)]

See also:

additive_chi2_kernel The additive version of this kernel

sklearn.kernel_approximation.AdditiveChi2Sampler A Fourier approximation to the additive version of this kernel.

References

- Zhang, J. and Marszalek, M. and Lazebnik, S. and Schmid, C. Local features and kernels for classification of texture and object categories: A comprehensive study International Journal of Computer Vision 2007 https://research.microsoft.com/en-us/um/people/manik/projects/trade-off/papers/ZhangIJCV06.pdf

sklearn.metrics.pairwise.cosine_similarity

sklearn.metrics.pairwise.**cosine_similarity**(*X*, *Y=None*, *dense_output=True*)

Compute cosine similarity between samples in X and Y.

Cosine similarity, or the cosine kernel, computes similarity as the normalized dot product of X and Y:

K(X, Y) = <X, Y> / (||X||*||Y||)

On L2-normalized data, this function is equivalent to linear_kernel.

Read more in the *User Guide*.

Parameters

X [ndarray or sparse array, shape: (n_samples_X, n_features)] Input data.

Y [ndarray or sparse array, shape: (n_samples_Y, n_features)] Input data. If `None`, the output will be the pairwise similarities between all samples in `X`.

dense_output [boolean (optional), default True] Whether to return dense output even when the input is sparse. If `False`, the output is sparse if both input arrays are sparse.

New in version 0.17: parameter `dense_output` for dense output.

Returns

kernel matrix [array] An array with shape (n_samples_X, n_samples_Y).

sklearn.metrics.pairwise.cosine_distances

sklearn.metrics.pairwise.**cosine_distances**(*X*, *Y=None*)

Compute cosine distance between samples in X and Y.

Cosine distance is defined as 1.0 minus the cosine similarity.

Read more in the *User Guide*.

Parameters

X [array_like, sparse matrix] with shape (n_samples_X, n_features).

Y [array_like, sparse matrix (optional)] with shape (n_samples_Y, n_features).

Returns

distance matrix [array] An array with shape (n_samples_X, n_samples_Y).

See also:

sklearn.metrics.pairwise.cosine_similarity

scipy.spatial.distance.cosine dense matrices only

sklearn.metrics.pairwise.distance_metrics

sklearn.metrics.pairwise.**distance_metrics**()

Valid metrics for pairwise_distances.

This function simply returns the valid pairwise distance metrics. It exists to allow for a description of the mapping for each of the valid strings.

The valid distance metrics, and the function they map to, are:

metric	Function
'cityblock'	metrics.pairwise.manhattan_distances
'cosine'	metrics.pairwise.cosine_distances
'euclidean'	metrics.pairwise.euclidean_distances
'haversine'	metrics.pairwise.haversine_distances
'l1'	metrics.pairwise.manhattan_distances
'l2'	metrics.pairwise.euclidean_distances
'manhattan'	metrics.pairwise.manhattan_distances

Read more in the *User Guide*.

sklearn.metrics.pairwise.euclidean_distances

sklearn.metrics.pairwise.**euclidean_distances**(*X*, *Y=None*, *Y_norm_squared=None*, *squared=False*, *X_norm_squared=None*)

Considering the rows of X (and Y=X) as vectors, compute the distance matrix between each pair of vectors.

For efficiency reasons, the euclidean distance between a pair of row vector x and y is computed as:

```
dist(x, y) = sqrt(dot(x, x) - 2 * dot(x, y) + dot(y, y))
```

This formulation has two advantages over other ways of computing distances. First, it is computationally efficient when dealing with sparse data. Second, if one argument varies but the other remains unchanged, then `dot(x, x)` and/or `dot(y, y)` can be pre-computed.

However, this is not the most precise way of doing this computation, and the distance matrix returned by this function may not be exactly symmetric as required by, e.g., `scipy.spatial.distance` functions.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape (n_samples_1, n_features)]

Y [{array-like, sparse matrix}, shape (n_samples_2, n_features)]

Y_norm_squared [array-like, shape (n_samples_2,), optional] Pre-computed dot-products of vectors in Y (e.g., `(Y**2).sum(axis=1)`) May be ignored in some cases, see the note below.

squared [boolean, optional] Return squared Euclidean distances.

X_norm_squared [array-like, shape = [n_samples_1], optional] Pre-computed dot-products of vectors in X (e.g., `(X**2).sum(axis=1)`) May be ignored in some cases, see the note below.

Returns

distances [array, shape (n_samples_1, n_samples_2)]

See also:

paired_distances distances betweens pairs of elements of X and Y.

Notes

To achieve better accuracy, `X_norm_squared` and `Y_norm_squared` may be unused if they are passed as `float32`.

Examples

```
>>> from sklearn.metrics.pairwise import euclidean_distances
>>> X = [[0, 1], [1, 1]]
>>> # distance between rows of X
>>> euclidean_distances(X, X)
array([[0., 1.],
       [1., 0.]])
>>> # get distance to origin
>>> euclidean_distances(X, [[0, 0]])
array([[1.        ],
       [1.41421356]])
```

sklearn.metrics.pairwise.haversine_distances

sklearn.metrics.pairwise.**haversine_distances**(*X*, *Y=None*)

Compute the Haversine distance between samples in X and Y

The Haversine (or great circle) distance is the angular distance between two points on the surface of a sphere. The first distance of each point is assumed to be the latitude, the second is the longitude, given in radians. The dimension of the data must be 2.

$$D(x, y) = 2\arcsin[\sqrt{\sin^2((x1 - y1)/2) + \cos(x1)\cos(y1)\sin^2((x2 - y2)/2)}]$$

Parameters

X [array_like, shape (n_samples_1, 2)]

Y [array_like, shape (n_samples_2, 2), optional]

Returns

distance [{array}, shape (n_samples_1, n_samples_2)]

Notes

As the Earth is nearly spherical, the haversine formula provides a good approximation of the distance between two points of the Earth surface, with a less than 1% error on average.

Examples

We want to calculate the distance between the Ezeiza Airport (Buenos Aires, Argentina) and the Charles de Gaulle Airport (Paris, France)

```
>>> from sklearn.metrics.pairwise import haversine_distances
>>> bsas = [-34.83333, -58.5166646]
>>> paris = [49.0083899664, 2.53844117956]
>>> result = haversine_distances([bsas, paris])
>>> result * 6371000/1000  # multiply by Earth radius to get kilometers
array([[    0.        , 11279.45379464],
       [11279.45379464,     0.        ]])
```

sklearn.metrics.pairwise.kernel_metrics

sklearn.metrics.pairwise.**kernel_metrics**()
Valid metrics for pairwise_kernels

This function simply returns the valid pairwise distance metrics. It exists, however, to allow for a verbose description of the mapping for each of the valid strings.

The valid distance metrics, and the function they map to, are:

metric	Function
'additive_chi2'	sklearn.pairwise.additive_chi2_kernel
'chi2'	sklearn.pairwise.chi2_kernel
'linear'	sklearn.pairwise.linear_kernel
'poly'	sklearn.pairwise.polynomial_kernel
'polynomial'	sklearn.pairwise.polynomial_kernel
'rbf'	sklearn.pairwise.rbf_kernel
'laplacian'	sklearn.pairwise.laplacian_kernel
'sigmoid'	sklearn.pairwise.sigmoid_kernel
'cosine'	sklearn.pairwise.cosine_similarity

Read more in the *User Guide*.

sklearn.metrics.pairwise.laplacian_kernel

sklearn.metrics.pairwise.**laplacian_kernel**(*X*, *Y=None*, *gamma=None*)
Compute the laplacian kernel between X and Y.

The laplacian kernel is defined as:

```
K(x, y) = exp(-gamma ||x-y||_1)
```

for each pair of rows x in X and y in Y. Read more in the *User Guide*.

New in version 0.17.

Parameters

X [array of shape (n_samples_X, n_features)]

Y [array of shape (n_samples_Y, n_features)]

gamma [float, default None] If None, defaults to 1.0 / n_features

Returns

kernel_matrix [array of shape (n_samples_X, n_samples_Y)]

sklearn.metrics.pairwise.linear_kernel

sklearn.metrics.pairwise.**linear_kernel**(*X*, *Y=None*, *dense_output=True*)
Compute the linear kernel between X and Y.

Read more in the *User Guide*.

Parameters

X [array of shape (n_samples_1, n_features)]

Y [array of shape (n_samples_2, n_features)]

dense_output [boolean (optional), default True] Whether to return dense output even when the input is sparse. If `False`, the output is sparse if both input arrays are sparse.

New in version 0.20.

Returns

Gram matrix [array of shape (n_samples_1, n_samples_2)]

sklearn.metrics.pairwise.manhattan_distances

sklearn.metrics.pairwise.**manhattan_distances**(*X*, *Y=None*, *sum_over_features=True*)

Compute the L1 distances between the vectors in X and Y.

With sum_over_features equal to False it returns the componentwise distances.

Read more in the *User Guide*.

Parameters

X [array_like] An array with shape (n_samples_X, n_features).

Y [array_like, optional] An array with shape (n_samples_Y, n_features).

sum_over_features [bool, default=True] If True the function returns the pairwise distance matrix else it returns the componentwise L1 pairwise-distances. Not supported for sparse matrix inputs.

Returns

D [array] If sum_over_features is False shape is (n_samples_X * n_samples_Y, n_features) and D contains the componentwise L1 pairwise-distances (ie. absolute difference), else shape is (n_samples_X, n_samples_Y) and D contains the pairwise L1 distances.

Examples

```
>>> from sklearn.metrics.pairwise import manhattan_distances
>>> manhattan_distances([[3]], [[3]])
array([[0.]])
>>> manhattan_distances([[3]], [[2]])
array([[1.]])
>>> manhattan_distances([[2]], [[3]])
array([[1.]])
>>> manhattan_distances([[1, 2], [3, 4]],         [[1, 2], [0, 3]])
array([[0., 2.],
       [4., 4.]])
>>> import numpy as np
>>> X = np.ones((1, 2))
>>> y = np.full((2, 2), 2.)
>>> manhattan_distances(X, y, sum_over_features=False)
array([[1., 1.],
       [1., 1.]])
```

sklearn.metrics.pairwise.pairwise_kernels

sklearn.metrics.pairwise.**pairwise_kernels**(*X*, *Y=None*, *metric='linear'*, *filter_params=False*, *n_jobs=None*, ***kwds*)

Compute the kernel between arrays X and optional array Y.

This method takes either a vector array or a kernel matrix, and returns a kernel matrix. If the input is a vector array, the kernels are computed. If the input is a kernel matrix, it is returned instead.

This method provides a safe way to take a kernel matrix as input, while preserving compatibility with many other algorithms that take a vector array.

If Y is given (default is None), then the returned matrix is the pairwise kernel between the arrays from both X and Y.

Valid values for metric are::

['additive_chi2', 'chi2', 'linear', 'poly', 'polynomial', 'rbf', 'laplacian', 'sigmoid', 'cosine']

Read more in the *User Guide*.

Parameters

X [array [n_samples_a, n_samples_a] if metric == "precomputed", or, [n_samples_a, n_features] otherwise] Array of pairwise kernels between samples, or a feature array.

Y [array [n_samples_b, n_features]] A second feature array only if X has shape [n_samples_a, n_features].

metric [string, or callable] The metric to use when calculating kernel between instances in a feature array. If metric is a string, it must be one of the metrics in pairwise.PAIRWISE_KERNEL_FUNCTIONS. If metric is "precomputed", X is assumed to be a kernel matrix. Alternatively, 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 from X as input and return a value indicating the distance between them.

filter_params [boolean] Whether to filter invalid parameters or not.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

****kwds** [optional keyword parameters] Any further parameters are passed directly to the kernel function.

Returns

K [array [n_samples_a, n_samples_a] or [n_samples_a, n_samples_b]] A kernel matrix K such that K_{i, j} is the kernel between the ith and jth vectors of the given matrix X, if Y is None. If Y is not None, then K_{i, j} is the kernel between the ith array from X and the jth array from Y.

Notes

If metric is 'precomputed', Y is ignored and X is returned.

sklearn.metrics.pairwise.polynomial_kernel

sklearn.metrics.pairwise.**polynomial_kernel**(*X*, *Y=None*, *degree=3*, *gamma=None*, *coef0=1*)

Compute the polynomial kernel between X and Y:

```
K(X, Y) = (gamma <X, Y> + coef0)^degree
```

Read more in the *User Guide*.

Parameters

X [ndarray of shape (n_samples_1, n_features)]

Y [ndarray of shape (n_samples_2, n_features)]

degree [int, default 3]

gamma [float, default None] if None, defaults to 1.0 / n_features

coef0 [float, default 1]

Returns

Gram matrix [array of shape (n_samples_1, n_samples_2)]

sklearn.metrics.pairwise.rbf_kernel

sklearn.metrics.pairwise.**rbf_kernel**(*X*, *Y=None*, *gamma=None*)

Compute the rbf (gaussian) kernel between X and Y:

```
K(x, y) = exp(-gamma ||x-y||^2)
```

for each pair of rows x in X and y in Y.

Read more in the *User Guide*.

Parameters

X [array of shape (n_samples_X, n_features)]

Y [array of shape (n_samples_Y, n_features)]

gamma [float, default None] If None, defaults to 1.0 / n_features

Returns

kernel_matrix [array of shape (n_samples_X, n_samples_Y)]

sklearn.metrics.pairwise.sigmoid_kernel

sklearn.metrics.pairwise.**sigmoid_kernel**(*X*, *Y=None*, *gamma=None*, *coef0=1*)

Compute the sigmoid kernel between X and Y:

```
K(X, Y) = tanh(gamma <X, Y> + coef0)
```

Read more in the *User Guide*.

Parameters

X [ndarray of shape (n_samples_1, n_features)]

Y [ndarray of shape (n_samples_2, n_features)]

gamma [float, default None] If None, defaults to 1.0 / n_features

coef0 [float, default 1]

Returns

Gram matrix [array of shape (n_samples_1, n_samples_2)]

sklearn.metrics.pairwise.paired_euclidean_distances

sklearn.metrics.pairwise.**paired_euclidean_distances**(*X*, *Y*)

Computes the paired euclidean distances between X and Y

Read more in the *User Guide*.

Parameters

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

Y [array-like, shape (n_samples, n_features)]

Returns

distances [ndarray (n_samples,)]

sklearn.metrics.pairwise.paired_manhattan_distances

sklearn.metrics.pairwise.**paired_manhattan_distances**(*X*, *Y*)

Compute the L1 distances between the vectors in X and Y.

Read more in the *User Guide*.

Parameters

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

Y [array-like, shape (n_samples, n_features)]

Returns

distances [ndarray (n_samples,)]

sklearn.metrics.pairwise.paired_cosine_distances

sklearn.metrics.pairwise.**paired_cosine_distances**(*X*, *Y*)

Computes the paired cosine distances between X and Y

Read more in the *User Guide*.

Parameters

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

Y [array-like, shape (n_samples, n_features)]

Returns

distances [ndarray, shape (n_samples,)]

Notes

The cosine distance is equivalent to the half the squared euclidean distance if each sample is normalized to unit norm

sklearn.metrics.pairwise.paired_distances

sklearn.metrics.pairwise.**paired_distances**(*X*, *Y*, *metric='euclidean'*, ***kwds*)

Computes the paired distances between X and Y.

Computes the distances between (X[0], Y[0]), (X[1], Y[1]), etc...

Read more in the *User Guide*.

Parameters

X [ndarray (n_samples, n_features)] Array 1 for distance computation.

Y [ndarray (n_samples, n_features)] Array 2 for distance computation.

metric [string or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options specified in PAIRED_DISTANCES, including "euclidean", "manhattan", or "cosine". Alternatively, 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 from X as input and return a value indicating the distance between them.

Returns

distances [ndarray (n_samples,)]

See also:

pairwise_distances Computes the distance between every pair of samples

Examples

```
>>> from sklearn.metrics.pairwise import paired_distances
>>> X = [[0, 1], [1, 1]]
>>> Y = [[0, 1], [2, 1]]
>>> paired_distances(X, Y)
array([0., 1.])
```

sklearn.metrics.pairwise_distances

sklearn.metrics.**pairwise_distances**(*X*, *Y=None*, *metric='euclidean'*, *n_jobs=None*, ***kwds*)

Compute the distance matrix from a vector array X and optional Y.

This method takes either a vector array or a distance matrix, and returns a distance matrix. If the input is a vector array, the distances are computed. If the input is a distances matrix, it is returned instead.

This method provides a safe way to take a distance matrix as input, while preserving compatibility with many other algorithms that take a vector array.

If Y is given (default is None), then the returned matrix is the pairwise distance between the arrays from both X and Y.

Valid values for metric are:

- From scikit-learn: [‘cityblock’, ‘cosine’, ‘euclidean’, ‘l1’, ‘l2’, ‘manhattan’]. These metrics support sparse matrix inputs.
- 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. These metrics do not support sparse matrix inputs.

Note that in the case of ‘cityblock’, ‘cosine’ and ‘euclidean’ (which are valid scipy.spatial.distance metrics), the scikit-learn implementation will be used, which is faster and has support for sparse matrices (except for ‘cityblock’). For a verbose description of the metrics from scikit-learn, see the __doc__ of the sklearn.pairwise.distance_metrics function.

Read more in the *User Guide*.

Parameters

X [array [n_samples_a, n_samples_a] if metric == “precomputed”, or, [n_samples_a, n_features] otherwise] Array of pairwise distances between samples, or a feature array.

Y [array [n_samples_b, n_features], optional] An optional second feature array. Only allowed if metric != “precomputed”.

metric [string, or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by scipy.spatial.distance.pdist for its metric parameter, or a metric listed in pairwise.PAIRWISE_DISTANCE_FUNCTIONS. If metric is “precomputed”, X is assumed to be a distance matrix. Alternatively, 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 from X as input and return a value indicating the distance between them.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

****kwds** [optional keyword parameters] Any further parameters are passed directly to the distance function. If using a scipy.spatial.distance metric, the parameters are still metric dependent. See the scipy docs for usage examples.

Returns

D [array [n_samples_a, n_samples_a] or [n_samples_a, n_samples_b]] A distance matrix D such that D_{i, j} is the distance between the ith and jth vectors of the given matrix X, if Y is None. If Y is not None, then D_{i, j} is the distance between the ith array from X and the jth array from Y.

See also:

pairwise_distances_chunked performs the same calculation as this function, but returns a generator of chunks of the distance matrix, in order to limit memory usage.

paired_distances Computes the distances between corresponding elements of two arrays

Examples using `sklearn.metrics.pairwise_distances`

- *Agglomerative clustering with different metrics*

`sklearn.metrics.pairwise_distances_argmin`

sklearn.metrics.**pairwise_distances_argmin**(*X*, *Y*, *axis=1*, *metric='euclidean'*, *batch_size=None*, *metric_kwargs=None*)

Compute minimum distances between one point and a set of points.

This function computes for each row in X, the index of the row of Y which is closest (according to the specified distance).

This is mostly equivalent to calling:

pairwise_distances(X, Y=Y, metric=metric).argmin(axis=axis)

but uses much less memory, and is faster for large arrays.

This function works with dense 2D arrays only.

Parameters

X [array-like] Arrays containing points. Respective shapes (n_samples1, n_features) and (n_samples2, n_features)

Y [array-like] Arrays containing points. Respective shapes (n_samples1, n_features) and (n_samples2, n_features)

axis [int, optional, default 1] Axis along which the argmin and distances are to be computed.

metric [string or callable] 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.

batch_size [integer] Deprecated since version 0.20: Deprecated for removal in 0.22. Use sklearn.set_config(working_memory=...) instead.

metric_kwargs [dict] keyword arguments to pass to specified metric function.

Returns

argmin [numpy.ndarray] Y[argmin[i], :] is the row in Y that is closest to X[i, :].

See also:

sklearn.metrics.pairwise_distances

sklearn.metrics.pairwise_distances_argmin_min

Examples using sklearn.metrics.pairwise_distances_argmin

- *Color Quantization using K-Means*
- *Comparison of the K-Means and MiniBatchKMeans clustering algorithms*

sklearn.metrics.pairwise_distances_argmin_min

sklearn.metrics.**pairwise_distances_argmin_min**(*X*, *Y*, *axis=1*, *metric='euclidean'*, *batch_size=None*, *metric_kwargs=None*)

Compute minimum distances between one point and a set of points.

This function computes for each row in X, the index of the row of Y which is closest (according to the specified distance). The minimal distances are also returned.

This is mostly equivalent to calling:

(pairwise_distances(X, Y=Y, metric=metric).argmin(axis=axis), pairwise_distances(X, Y=Y, metric=metric).min(axis=axis))

but uses much less memory, and is faster for large arrays.

Parameters

X [{array-like, sparse matrix}, shape (n_samples1, n_features)] Array containing points.

Y [{array-like, sparse matrix}, shape (n_samples2, n_features)] Arrays containing points.

axis [int, optional, default 1] Axis along which the argmin and distances are to be computed.

metric [string or callable, default 'euclidean'] 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.

batch_size [integer] Deprecated since version 0.20: Deprecated for removal in 0.22. Use sklearn.set_config(working_memory=...) instead.

metric_kwargs [dict, optional] Keyword arguments to pass to specified metric function.

Returns

argmin [numpy.ndarray] Y[argmin[i], :] is the row in Y that is closest to X[i, :].

distances [numpy.ndarray] distances[i] is the distance between the i-th row in X and the argmin[i]-th row in Y.

See also:

sklearn.metrics.pairwise_distances

sklearn.metrics.pairwise_distances_argmin

sklearn.metrics.pairwise_distances_chunked

sklearn.metrics.**pairwise_distances_chunked**(*X*, *Y=None*, *reduce_func=None*, *metric='euclidean'*, *n_jobs=None*, *working_memory=None*, ***kwds*)

Generate a distance matrix chunk by chunk with optional reduction

In cases where not all of a pairwise distance matrix needs to be stored at once, this is used to calculate pairwise distances in `working_memory`-sized chunks. If `reduce_func` is given, it is run on each chunk and its return values are concatenated into lists, arrays or sparse matrices.

Parameters

X [array [n_samples_a, n_samples_a] if metric == "precomputed", or,] [n_samples_a, n_features] otherwise Array of pairwise distances between samples, or a feature array.

Y [array [n_samples_b, n_features], optional] An optional second feature array. Only allowed if metric != "precomputed".

reduce_func [callable, optional] The function which is applied on each chunk of the distance matrix, reducing it to needed values. `reduce_func(D_chunk, start)` is called repeatedly, where `D_chunk` is a contiguous vertical slice of the pairwise distance matrix, starting at row `start`. It should return an array, a list, or a sparse matrix of length `D_chunk.shape[0]`, or a tuple of such objects.

If None, pairwise_distances_chunked returns a generator of vertical chunks of the distance matrix.

metric [string, or callable] The metric to use when calculating distance between instances in a feature array. If metric is a string, it must be one of the options allowed by scipy.spatial.distance.pdist for its metric parameter, or a metric listed in pairwise.PAIRWISE_DISTANCE_FUNCTIONS. If metric is "precomputed", X is assumed to be a distance matrix. Alternatively, 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 from X as input and return a value indicating the distance between them.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel.

`None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

working_memory [int, optional] The sought maximum memory for temporary distance matrix chunks. When None (default), the value of `sklearn.get_config()['working_memory']` is used.

'kwds'** [optional keyword parameters] Any further parameters are passed directly to the distance function. If using a scipy.spatial.distance metric, the parameters are still metric dependent. See the scipy docs for usage examples.

Yields

D_chunk [array or sparse matrix] A contiguous slice of distance matrix, optionally processed by `reduce_func`.

Examples

Without reduce_func:

```
>>> import numpy as np
>>> from sklearn.metrics import pairwise_distances_chunked
>>> X = np.random.RandomState(0).rand(5, 3)
>>> D_chunk = next(pairwise_distances_chunked(X))
>>> D_chunk
array([[0.  ..., 0.29..., 0.41..., 0.19..., 0.57...],
       [0.29..., 0.  ..., 0.57..., 0.41..., 0.76...],
       [0.41..., 0.57..., 0.  ..., 0.44..., 0.90...],
       [0.19..., 0.41..., 0.44..., 0.  ..., 0.51...],
       [0.57..., 0.76..., 0.90..., 0.51..., 0.  ...]])
```

Retrieve all neighbors and average distance within radius r:

```
>>> r = .2
>>> def reduce_func(D_chunk, start):
...     neigh = [np.flatnonzero(d < r) for d in D_chunk]
...     avg_dist = (D_chunk * (D_chunk < r)).mean(axis=1)
...     return neigh, avg_dist
>>> gen = pairwise_distances_chunked(X, reduce_func=reduce_func)
>>> neigh, avg_dist = next(gen)
>>> neigh
[array([0, 3]), array([1]), array([2]), array([0, 3]), array([4])]
>>> avg_dist
array([0.039..., 0.        , 0.        , 0.039..., 0.        ])
```

Where r is defined per sample, we need to make use of `start`:

```
>>> r = [.2, .4, .4, .3, .1]
>>> def reduce_func(D_chunk, start):
...     neigh = [np.flatnonzero(d < r[i])
...              for i, d in enumerate(D_chunk, start)]
...     return neigh
>>> neigh = next(pairwise_distances_chunked(X, reduce_func=reduce_func))
>>> neigh
[array([0, 3]), array([0, 1]), array([2]), array([0, 3]), array([4])]
```

Force row-by-row generation by reducing `working_memory`:

```
>>> gen = pairwise_distances_chunked(X, reduce_func=reduce_func,
...                                  working_memory=0)
>>> next(gen)
[array([0, 3])]
>>> next(gen)
[array([0, 1])]
```

6.25 `sklearn.mixture`: Gaussian Mixture Models

The *`sklearn.mixture`* module implements mixture modeling algorithms.

User guide: See the *Gaussian mixture models* section for further details.

`mixture.BayesianGaussianMixture`([...])	Variational Bayesian estimation of a Gaussian mixture.
`mixture.GaussianMixture`([n_components, ...])	Gaussian Mixture.

6.25.1 sklearn.mixture.BayesianGaussianMixture

class sklearn.mixture.**BayesianGaussianMixture**(*n_components=1*, *covariance_type='full'*, *tol=0.001*, *reg_covar=1e-06*, *max_iter=100*, *n_init=1*, *init_params='kmeans'*, *weight_concentration_prior_type='dirichlet_process'*, *weight_concentration_prior=None*, *mean_precision_prior=None*, *mean_prior=None*, *degrees_of_freedom_prior=None*, *covariance_prior=None*, *random_state=None*, *warm_start=False*, *verbose=0*, *verbose_interval=10*)

Variational Bayesian estimation of a Gaussian mixture.

This class allows to infer an approximate posterior distribution over the parameters of a Gaussian mixture distribution. The effective number of components can be inferred from the data.

This class implements two types of prior for the weights distribution: a finite mixture model with Dirichlet distribution and an infinite mixture model with the Dirichlet Process. In practice Dirichlet Process inference algorithm is approximated and uses a truncated distribution with a fixed maximum number of components (called the Stick-breaking representation). The number of components actually used almost always depends on the data.

New in version 0.18.

Read more in the *User Guide*.

Parameters

n_components [int, defaults to 1.] The number of mixture components. Depending on the data and the value of the `weight_concentration_prior` the model can decide to not use all the components by setting some component `weights_` to values very close to zero. The number of effective components is therefore smaller than n_components.

covariance_type [{'full', 'tied', 'diag', 'spherical'}, defaults to 'full'] String describing the type of covariance parameters to use. Must be one of:

```
'full' (each component has its own general covariance matrix),
'tied' (all components share the same general covariance matrix),
'diag' (each component has its own diagonal covariance matrix),
'spherical' (each component has its own single variance).
```

tol [float, defaults to 1e-3.] The convergence threshold. EM iterations will stop when the lower bound average gain on the likelihood (of the training data with respect to the model) is below this threshold.

reg_covar [float, defaults to 1e-6.] Non-negative regularization added to the diagonal of covariance. Allows to assure that the covariance matrices are all positive.

max_iter [int, defaults to 100.] The number of EM iterations to perform.

n_init [int, defaults to 1.] The number of initializations to perform. The result with the highest lower bound value on the likelihood is kept.

init_params [{'kmeans', 'random'}, defaults to 'kmeans'.] The method used to initialize the weights, the means and the covariances. Must be one of:

```
'kmeans' : responsibilities are initialized using kmeans.
'random' : responsibilities are initialized randomly.
```

weight_concentration_prior_type [str, defaults to 'dirichlet_process'.] String describing the type of the weight concentration prior. Must be one of:

```
'dirichlet_process' (using the Stick-breaking representation),
'dirichlet_distribution' (can favor more uniform weights).
```

weight_concentration_prior [float | None, optional.] The dirichlet concentration of each component on the weight distribution (Dirichlet). This is commonly called gamma in the literature. The higher concentration puts more mass in the center and will lead to more components being active, while a lower concentration parameter will lead to more mass at the edge of the mixture weights simplex. The value of the parameter must be greater than 0. If it is None, it's set to `1. / n_components`.

mean_precision_prior [float | None, optional.] The precision prior on the mean distribution (Gaussian). Controls the extend to where means can be placed. Larger values concentrate the means of each clusters around `mean_prior`. The value of the parameter must be greater than 0. If it is None, it's set to 1.

mean_prior [array-like, shape (n_features,), optional] The prior on the mean distribution (Gaussian). If it is None, it's set to the mean of X.

degrees_of_freedom_prior [float | None, optional.] The prior of the number of degrees of freedom on the covariance distributions (Wishart). If it is None, it's set to *n_features*.

covariance_prior [float or array-like, optional] The prior on the covariance distribution (Wishart). If it is None, the emiprical covariance prior is initialized using the covariance of X. The shape depends on `covariance_type`:

```
(n_features, n_features) if 'full',
(n_features, n_features) if 'tied',
(n_features)             if 'diag',
float                    if 'spherical'
```

random_state [int, RandomState instance or None, optional (default=None)] 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`.

warm_start [bool, default to False.] If 'warm_start' is True, the solution of the last fitting is used as initialization for the next call of fit(). This can speed up convergence when fit is called several times on similar problems. See *the Glossary*.

verbose [int, default to 0.] Enable verbose output. If 1 then it prints the current initialization and each iteration step. If greater than 1 then it prints also the log probability and the time needed for each step.

verbose_interval [int, default to 10.] Number of iteration done before the next print.

Attributes

weights_ [array-like, shape (n_components,)] The weights of each mixture components.

means_ [array-like, shape (n_components, n_features)] The mean of each mixture component.

covariances_ [array-like] The covariance of each mixture component. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

precisions_ [array-like] The precision matrices for each component in the mixture. A precision matrix is the inverse of a covariance matrix. A covariance matrix is symmetric positive definite so the mixture of Gaussian can be equivalently parameterized by the precision matrices. Storing the precision matrices instead of the covariance matrices makes it more efficient to compute the log-likelihood of new samples at test time. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

precisions_cholesky_ [array-like] The cholesky decomposition of the precision matrices of each mixture component. A precision matrix is the inverse of a covariance matrix. A covariance matrix is symmetric positive definite so the mixture of Gaussian can be equivalently parameterized by the precision matrices. Storing the precision matrices instead of the covariance matrices makes it more efficient to compute the log-likelihood of new samples at test time. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

converged_ [bool] True when convergence was reached in fit(), False otherwise.

n_iter_ [int] Number of step used by the best fit of inference to reach the convergence.

lower_bound_ [float] Lower bound value on the likelihood (of the training data with respect to the model) of the best fit of inference.

weight_concentration_prior_ [tuple or float] The dirichlet concentration of each component on the weight distribution (Dirichlet). The type depends on `weight_concentration_prior_type`:

```
(float, float) if 'dirichlet_process' (Beta parameters),
float          if 'dirichlet_distribution' (Dirichlet parameters).
```

The higher concentration puts more mass in the center and will lead to more components being active, while a lower concentration parameter will lead to more mass at the edge of the simplex.

weight_concentration_ [array-like, shape (n_components,)] The dirichlet concentration of each component on the weight distribution (Dirichlet).

mean_precision_prior [float] The precision prior on the mean distribution (Gaussian). Controls the extend to where means can be placed. Larger values concentrate the means of each clusters around `mean_prior`.

mean_precision_ [array-like, shape (n_components,)] The precision of each components on the mean distribution (Gaussian).

mean_prior_ [array-like, shape (n_features,)] The prior on the mean distribution (Gaussian).

degrees_of_freedom_prior_ [float] The prior of the number of degrees of freedom on the covariance distributions (Wishart).

degrees_of_freedom_ [array-like, shape (n_components,)] The number of degrees of freedom of each components in the model.

covariance_prior_ [float or array-like] The prior on the covariance distribution (Wishart). The shape depends on `covariance_type`:

```
(n_features, n_features) if 'full',
(n_features, n_features) if 'tied',
(n_features)             if 'diag',
float                    if 'spherical'
```

See also:

GaussianMixture Finite Gaussian mixture fit with EM.

References

[R16529824bff2-1], *[R16529824bff2-2]*, *[R16529824bff2-3]*

Methods

fit(self, X[, y])	Estimate model parameters with the EM algorithm.
fit_predict(self, X[, y])	Estimate model parameters using X and predict the labels for X.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the labels for the data samples in X using trained model.
predict_proba(self, X)	Predict posterior probability of each component given the data.
sample(self[, n_samples])	Generate random samples from the fitted Gaussian distribution.
score(self, X[, y])	Compute the per-sample average log-likelihood of the given data X.
score_samples(self, X)	Compute the weighted log probabilities for each sample.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_components=1*, *covariance_type='full'*, *tol=0.001*, *reg_covar=1e-06*, *max_iter=100*, *n_init=1*, *init_params='kmeans'*, *weight_concentration_prior_type='dirichlet_process'*, *weight_concentration_prior=None*, *mean_precision_prior=None*, *mean_prior=None*, *degrees_of_freedom_prior=None*, *covariance_prior=None*, *random_state=None*, *warm_start=False*, *verbose=0*, *verbose_interval=10*)

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

Estimate model parameters with the EM algorithm.

The method fits the model `n_init` times and sets the parameters with which the model has the largest likelihood or lower bound. Within each trial, the method iterates between E-step and M-step for `max_iter` times until the change of likelihood or lower bound is less than `tol`, otherwise, a `ConvergenceWarning` is raised. If `warm_start` is `True`, then `n_init` is ignored and a single initialization is performed upon the first call. Upon consecutive calls, training starts where it left off.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

self

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

Estimate model parameters using X and predict the labels for X.

The method fits the model n_init times and sets the parameters with which the model has the largest likelihood or lower bound. Within each trial, the method iterates between E-step and M-step for *max_iter* times until the change of likelihood or lower bound is less than `tol`, otherwise, a *ConvergenceWarning* is raised. After fitting, it predicts the most probable label for the input data points.

New in version 0.20.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

labels [array, shape (n_samples,)] Component labels.

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)

Predict the labels for the data samples in X using trained 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.

Returns

labels [array, shape (n_samples,)] Component labels.

predict_proba (*self*, *X*)

Predict posterior probability of each component given 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.

Returns

resp [array, shape (n_samples, n_components)] Returns the probability each Gaussian (state) in the model given each sample.

sample (*self*, *n_samples=1*)
Generate random samples from the fitted Gaussian distribution.

Parameters

n_samples [int, optional] Number of samples to generate. Defaults to 1.

Returns

X [array, shape (n_samples, n_features)] Randomly generated sample

y [array, shape (nsamples,)] Component labels

score (*self*, *X*, *y=None*)
Compute the per-sample average log-likelihood of the given data X.

Parameters

X [array-like, shape (n_samples, n_dimensions)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

log_likelihood [float] Log likelihood of the Gaussian mixture given X.

score_samples (*self*, *X*)
Compute the weighted log probabilities for each sample.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

log_prob [array, shape (n_samples,)] Log probabilities of each data point in X.

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.mixture.BayesianGaussianMixture`

- *Gaussian Mixture Model Ellipsoids*
- *Gaussian Mixture Model Sine Curve*
- *Concentration Prior Type Analysis of Variation Bayesian Gaussian Mixture*

6.25.2 sklearn.mixture.GaussianMixture

class sklearn.mixture.**GaussianMixture**(*n_components=1*, *covariance_type='full'*, *tol=0.001*, *reg_covar=1e-06*, *max_iter=100*, *n_init=1*, *init_params='kmeans'*, *weights_init=None*, *means_init=None*, *precisions_init=None*, *random_state=None*, *warm_start=False*, *verbose=0*, *verbose_interval=10*)

Gaussian Mixture.

Representation of a Gaussian mixture model probability distribution. This class allows to estimate the parameters of a Gaussian mixture distribution.

Read more in the *User Guide*.

New in version 0.18.

Parameters

n_components [int, defaults to 1.] The number of mixture components.

covariance_type [{'full' (default), 'tied', 'diag', 'spherical'}] String describing the type of covariance parameters to use. Must be one of:

'full' each component has its own general covariance matrix

'tied' all components share the same general covariance matrix

'diag' each component has its own diagonal covariance matrix

'spherical' each component has its own single variance

tol [float, defaults to 1e-3.] The convergence threshold. EM iterations will stop when the lower bound average gain is below this threshold.

reg_covar [float, defaults to 1e-6.] Non-negative regularization added to the diagonal of covariance. Allows to assure that the covariance matrices are all positive.

max_iter [int, defaults to 100.] The number of EM iterations to perform.

n_init [int, defaults to 1.] The number of initializations to perform. The best results are kept.

init_params [{'kmeans', 'random'}, defaults to 'kmeans'.] The method used to initialize the weights, the means and the precisions. Must be one of:

```
'kmeans' : responsibilities are initialized using kmeans.
'random' : responsibilities are initialized randomly.
```

weights_init [array-like, shape (n_components,), optional] The user-provided initial weights, defaults to None. If it None, weights are initialized using the `init_params` method.

means_init [array-like, shape (n_components, n_features), optional] The user-provided initial means, defaults to None, If it None, means are initialized using the `init_params` method.

precisions_init [array-like, optional.] The user-provided initial precisions (inverse of the covariance matrices), defaults to None. If it None, precisions are initialized using the 'init_params' method. The shape depends on 'covariance_type':

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

random_state [int, RandomState instance or None, optional (default=None)] 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`.

warm_start [bool, default to False.] If 'warm_start' is True, the solution of the last fitting is used as initialization for the next call of fit(). This can speed up convergence when fit is called several times on similar problems. In that case, 'n_init' is ignored and only a single initialization occurs upon the first call. See *the Glossary*.

verbose [int, default to 0.] Enable verbose output. If 1 then it prints the current initialization and each iteration step. If greater than 1 then it prints also the log probability and the time needed for each step.

verbose_interval [int, default to 10.] Number of iteration done before the next print.

Attributes

weights_ [array-like, shape (n_components,)] The weights of each mixture components.

means_ [array-like, shape (n_components, n_features)] The mean of each mixture component.

covariances_ [array-like] The covariance of each mixture component. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

precisions_ [array-like] The precision matrices for each component in the mixture. A precision matrix is the inverse of a covariance matrix. A covariance matrix is symmetric positive definite so the mixture of Gaussian can be equivalently parameterized by the precision matrices. Storing the precision matrices instead of the covariance matrices makes it more efficient to compute the log-likelihood of new samples at test time. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

precisions_cholesky_ [array-like] The cholesky decomposition of the precision matrices of each mixture component. A precision matrix is the inverse of a covariance matrix. A covariance matrix is symmetric positive definite so the mixture of Gaussian can be equivalently parameterized by the precision matrices. Storing the precision matrices instead of the covariance matrices makes it more efficient to compute the log-likelihood of new samples at test time. The shape depends on `covariance_type`:

```
(n_components,)                        if 'spherical',
(n_features, n_features)               if 'tied',
(n_components, n_features)             if 'diag',
(n_components, n_features, n_features) if 'full'
```

converged_ [bool] True when convergence was reached in fit(), False otherwise.

n_iter_ [int] Number of step used by the best fit of EM to reach the convergence.

lower_bound_ [float] Lower bound value on the log-likelihood (of the training data with respect to the model) of the best fit of EM.

See also:

BayesianGaussianMixture Gaussian mixture model fit with a variational inference.

Methods

aic(self, X)	Akaike information criterion for the current model on the input X.
bic(self, X)	Bayesian information criterion for the current model on the input X.
fit(self, X[, y])	Estimate model parameters with the EM algorithm.
fit_predict(self, X[, y])	Estimate model parameters using X and predict the labels for X.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the labels for the data samples in X using trained model.
predict_proba(self, X)	Predict posterior probability of each component given the data.
sample(self[, n_samples])	Generate random samples from the fitted Gaussian distribution.
score(self, X[, y])	Compute the per-sample average log-likelihood of the given data X.
score_samples(self, X)	Compute the weighted log probabilities for each sample.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_components=1*, *covariance_type='full'*, *tol=0.001*, *reg_covar=1e-06*, *max_iter=100*, *n_init=1*, *init_params='kmeans'*, *weights_init=None*, *means_init=None*, *precisions_init=None*, *random_state=None*, *warm_start=False*, *verbose=0*, *verbose_interval=10*)

aic(*self*, *X*)

Akaike information criterion for the current model on the input X.

Parameters

X [array of shape (n_samples, n_dimensions)]

Returns

aic [float] The lower the better.

bic(*self*, *X*)

Bayesian information criterion for the current model on the input X.

Parameters

X [array of shape (n_samples, n_dimensions)]

Returns

bic [float] The lower the better.

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

Estimate model parameters with the EM algorithm.

The method fits the model `n_init` times and sets the parameters with which the model has the largest likelihood or lower bound. Within each trial, the method iterates between E-step and M-step

for `max_iter` times until the change of likelihood or lower bound is less than `tol`, otherwise, a `ConvergenceWarning` is raised. If `warm_start` is `True`, then `n_init` is ignored and a single initialization is performed upon the first call. Upon consecutive calls, training starts where it left off.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

self

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

Estimate model parameters using X and predict the labels for X.

The method fits the model n_init times and sets the parameters with which the model has the largest likelihood or lower bound. Within each trial, the method iterates between E-step and M-step for *max_iter* times until the change of likelihood or lower bound is less than `tol`, otherwise, a *ConvergenceWarning* is raised. After fitting, it predicts the most probable label for the input data points.

New in version 0.20.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

labels [array, shape (n_samples,)] Component labels.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)

Predict the labels for the data samples in X using trained 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.

Returns

labels [array, shape (n_samples,)] Component labels.

predict_proba(*self*, *X*)

Predict posterior probability of each component given 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.

Returns

resp [array, shape (n_samples, n_components)] Returns the probability each Gaussian (state) in the model given each sample.

sample(*self*, *n_samples=1*)

Generate random samples from the fitted Gaussian distribution.

Parameters

n_samples [int, optional] Number of samples to generate. Defaults to 1.

Returns

X [array, shape (n_samples, n_features)] Randomly generated sample

y [array, shape (nsamples,)] Component labels

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

Compute the per-sample average log-likelihood of the given data X.

Parameters

X [array-like, shape (n_samples, n_dimensions)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

log_likelihood [float] Log likelihood of the Gaussian mixture given X.

score_samples(*self*, *X*)

Compute the weighted log probabilities for each sample.

Parameters

X [array-like, shape (n_samples, n_features)] List of n_features-dimensional data points. Each row corresponds to a single data point.

Returns

log_prob [array, shape (n_samples,)] Log probabilities of each data point in X.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.mixture.GaussianMixture`

- *Comparing different clustering algorithms on toy datasets*
- *Density Estimation for a Gaussian mixture*
- *Gaussian Mixture Model Ellipsoids*
- *Gaussian Mixture Model Selection*
- *GMM covariances*
- *Gaussian Mixture Model Sine Curve*

6.26 sklearn.model_selection: Model Selection

User guide: See the *Cross-validation: evaluating estimator performance*, *Tuning the hyper-parameters of an estimator* and *Learning curve* sections for further details.

6.26.1 Splitter Classes

model_selection.GroupKFold([n_splits])	K-fold iterator variant with non-overlapping groups.
model_selection.GroupShuffleSplit([...])	Shuffle-Group(s)-Out cross-validation iterator
model_selection.KFold([n_splits, shuffle, ...])	K-Folds cross-validator
model_selection.LeaveOneGroupOut	Leave One Group Out cross-validator
model_selection.LeavePGroupsOut(n_groups)	Leave P Group(s) Out cross-validator
model_selection.LeaveOneOut	Leave-One-Out cross-validator
model_selection.LeavePOut(p)	Leave-P-Out cross-validator
model_selection.PredefinedSplit(test_fold)	Predefined split cross-validator
model_selection.RepeatedKFold([n_splits, ...])	Repeated K-Fold cross validator.
model_selection.RepeatedStratifiedKFold([...])	Repeated Stratified K-Fold cross validator.
model_selection.ShuffleSplit([n_splits, ...])	Random permutation cross-validator
model_selection.StratifiedKFold([n_splits, ...])	Stratified K-Folds cross-validator
model_selection.StratifiedShuffleSplit([...])	Stratified ShuffleSplit cross-validator
model_selection.TimeSeriesSplit([n_splits, ...])	Time Series cross-validator

sklearn.model_selection.GroupKFold

class sklearn.model_selection.**GroupKFold**(*n_splits='warn'*)

K-fold iterator variant with non-overlapping groups.

The same group will not appear in two different folds (the number of distinct groups has to be at least equal to the number of folds).

The folds are approximately balanced in the sense that the number of distinct groups is approximately the same in each fold.

Parameters

n_splits [int, default=3] Number of folds. Must be at least 2.

Changed in version 0.20: `n_splits` default value will change from 3 to 5 in v0.22.

See also:

LeaveOneGroupOut For splitting the data according to explicit domain-specific stratification of the dataset.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import GroupKFold
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 3, 4])
>>> groups = np.array([0, 0, 2, 2])
```

```
>>> group_kfold = GroupKFold(n_splits=2)
>>> group_kfold.get_n_splits(X, y, groups)
2
>>> print(group_kfold)
GroupKFold(n_splits=2)
>>> for train_index, test_index in group_kfold.split(X, y, groups):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print(X_train, X_test, y_train, y_test)
...
TRAIN: [0 1] TEST: [2 3]
[[1 2]
 [3 4]] [[5 6]
 [7 8]] [1 2] [3 4]
TRAIN: [2 3] TEST: [0 1]
[[5 6]
 [7 8]] [[1 2]
 [3 4]] [3 4] [1 2]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits='warn'*)

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

y [array-like, shape (n_samples,), optional] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Examples using `sklearn.model_selection.GroupKFold`

- *Visualizing cross-validation behavior in scikit-learn*

`sklearn.model_selection.GroupShuffleSplit`

class `sklearn.model_selection.`**`GroupShuffleSplit`**(*n_splits=5*, *test_size=None*, *train_size=None*, *random_state=None*)

Shuffle-Group(s)-Out cross-validation iterator

Provides randomized train/test indices to split data according to a third-party provided group. This group information can be used to encode arbitrary domain specific stratifications of the samples as integers.

For instance the groups could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

The difference between LeavePGroupsOut and GroupShuffleSplit is that the former generates splits using all subsets of size `p` unique groups, whereas GroupShuffleSplit generates a user-determined number of random test splits, each with a user-determined fraction of unique groups.

For example, a less computationally intensive alternative to `LeavePGroupsOut(p=10)` would be `GroupShuffleSplit(test_size=10, n_splits=100)`.

Note: The parameters `test_size` and `train_size` refer to groups, and not to samples, as in ShuffleSplit.

Parameters

n_splits [int (default 5)] Number of re-shuffling & splitting iterations.

test_size [float, int, None, optional (default=None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test groups. If None, the value is set to the complement of the train size. If `train_size` is also None, it will be set to 0.2.

train_size [float, int, or None, default is None] If float, should be between 0.0 and 1.0 and represent the proportion of the groups to include in the train split. If int, represents the absolute number of train groups. If None, the value is automatically set to the complement of the test size.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

`__init__`(*self*, *n_splits=5*, *test_size=None*, *train_size=None*, *random_state=None*)

get_n_splits(*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split(*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

y [array-like, shape (n_samples,), optional] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples using `sklearn.model_selection.GroupShuffleSplit`

- *Visualizing cross-validation behavior in scikit-learn*

`sklearn.model_selection.KFold`

class `sklearn.model_selection.`**KFold**(*n_splits='warn'*, *shuffle=False*, *random_state=None*)
K-Folds cross-validator

Provides train/test indices to split data in train/test sets. Split dataset into k consecutive folds (without shuffling by default).

Each fold is then used once as a validation while the k - 1 remaining folds form the training set.

Read more in the *User Guide*.

Parameters

n_splits [int, default=3] Number of folds. Must be at least 2.

Changed in version 0.20: `n_splits` default value will change from 3 to 5 in v0.22.

shuffle [boolean, optional] Whether to shuffle the data before splitting into batches.

random_state [int, RandomState instance or None, optional, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `shuffle` == True.

See also:

StratifiedKFold Takes group information into account to avoid building folds with imbalanced class distributions (for binary or multiclass classification tasks).

GroupKFold K-fold iterator variant with non-overlapping groups.

RepeatedKFold Repeats K-Fold n times.

Notes

The first `n_samples % n_splits` folds have size `n_samples // n_splits + 1`, other folds have size `n_samples // n_splits`, where `n_samples` is the number of samples.

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import KFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4])
>>> kf = KFold(n_splits=2)
>>> kf.get_n_splits(X)
2
>>> print(kf)
KFold(n_splits=2, random_state=None, shuffle=False)
>>> for train_index, test_index in kf.split(X):
...    print("TRAIN:", train_index, "TEST:", test_index)
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
TRAIN: [2 3] TEST: [0 1]
TRAIN: [0 1] TEST: [2 3]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits='warn'*, *shuffle=False*, *random_state=None*)

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

y [array-like, shape (n_samples,)] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Examples using `sklearn.model_selection.KFold`

- *Feature agglomeration vs. univariate selection*
- *Gradient Boosting Out-of-Bag estimates*
- *Cross-validation on diabetes Dataset Exercise*
- *Nested versus non-nested cross-validation*
- *Visualizing cross-validation behavior in scikit-learn*

`sklearn.model_selection`.LeaveOneGroupOut

class `sklearn.model_selection.`**LeaveOneGroupOut**
Leave One Group Out cross-validator

Provides train/test indices to split data according to a third-party provided group. This group information can be used to encode arbitrary domain specific stratifications of the samples as integers.

For instance the groups could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

Read more in the *User Guide*.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import LeaveOneGroupOut
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 1, 2])
>>> groups = np.array([1, 1, 2, 2])
```

```
>>> logo = LeaveOneGroupOut()
>>> logo.get_n_splits(X, y, groups)
2
>>> logo.get_n_splits(groups=groups)  # 'groups' is always required
2
>>> print(logo)
LeaveOneGroupOut()
>>> for train_index, test_index in logo.split(X, y, groups):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print(X_train, X_test, y_train, y_test)
TRAIN: [2 3] TEST: [0 1]
[[5 6]
 [7 8]] [[1 2]
 [3 4]] [1 2] [1 2]
TRAIN: [0 1] TEST: [2 3]
[[1 2]
 [3 4]] [[5 6]
 [7 8]] [1 2] [1 2]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

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

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set. This 'groups' parameter must always be specified to calculate the number of splits, though the other parameters can be omitted.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

y [array-like, of length n_samples, optional] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.LeavePGroupsOut

class sklearn.model_selection.**LeavePGroupsOut**(*n_groups*)

Leave P Group(s) Out cross-validator

Provides train/test indices to split data according to a third-party provided group. This group information can be used to encode arbitrary domain specific stratifications of the samples as integers.

For instance the groups could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

The difference between LeavePGroupsOut and LeaveOneGroupOut is that the former builds the test sets with all the samples assigned to `p` different values of the groups while the latter uses samples all assigned the same groups.

Read more in the *User Guide*.

Parameters

n_groups [int] Number of groups (`p`) to leave out in the test split.

See also:

GroupKFold K-fold iterator variant with non-overlapping groups.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import LeavePGroupsOut
>>> X = np.array([[1, 2], [3, 4], [5, 6]])
>>> y = np.array([1, 2, 1])
>>> groups = np.array([1, 2, 3])
>>> lpgo = LeavePGroupsOut(n_groups=2)
>>> lpgo.get_n_splits(X, y, groups)
3
>>> lpgo.get_n_splits(groups=groups)  # 'groups' is always required
3
>>> print(lpgo)
LeavePGroupsOut(n_groups=2)
>>> for train_index, test_index in lpgo.split(X, y, groups):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print(X_train, X_test, y_train, y_test)
TRAIN: [2] TEST: [0 1]
[[5 6]] [[1 2]
 [3 4]] [1] [1 2]
TRAIN: [1] TEST: [0 2]
[[3 4]] [[1 2]
 [5 6]] [2] [1 1]
```

```
TRAIN: [0] TEST: [1 2]
[[1 2]] [[3 4]
 [5 6]] [1] [2 1]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_groups*)

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set. This 'groups' parameter must always be specified to calculate the number of splits, though the other parameters can be omitted.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

y [array-like, of length n_samples, optional] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,)] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.LeaveOneOut

class sklearn.model_selection.**LeaveOneOut**
Leave-One-Out cross-validator

Provides train/test indices to split data in train/test sets. Each sample is used once as a test set (singleton) while the remaining samples form the training set.

Note: `LeaveOneOut()` is equivalent to `KFold(n_splits=n)` and `LeavePOut(p=1)` where `n` is the number of samples.

Due to the high number of test sets (which is the same as the number of samples) this cross-validation method can be very costly. For large datasets one should favor *KFold*, *ShuffleSplit* or *StratifiedKFold*.

Read more in the *User Guide*.

See also:

LeaveOneGroupOut For splitting the data according to explicit, domain-specific stratification of the dataset.

GroupKFold K-fold iterator variant with non-overlapping groups.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import LeaveOneOut
>>> X = np.array([[1, 2], [3, 4]])
>>> y = np.array([1, 2])
>>> loo = LeaveOneOut()
>>> loo.get_n_splits(X)
2
>>> print(loo)
LeaveOneOut()
>>> for train_index, test_index in loo.split(X):
...    print("TRAIN:", train_index, "TEST:", test_index)
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
...    print(X_train, X_test, y_train, y_test)
TRAIN: [1] TEST: [0]
[[3 4]] [[1 2]] [2] [1]
TRAIN: [0] TEST: [1]
[[1 2]] [[3 4]] [1] [2]
```

Methods

get_n_splits(self, X[, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

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

get_n_splits (*self*, *X*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

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

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

y [array-like, of length n_samples] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.LeavePOut

class sklearn.model_selection.**LeavePOut**(*p*)
Leave-P-Out cross-validator

Provides train/test indices to split data in train/test sets. This results in testing on all distinct samples of size p, while the remaining n - p samples form the training set in each iteration.

Note: `LeavePOut(p)` is NOT equivalent to `KFold(n_splits=n_samples // p)` which creates non-overlapping test sets.

Due to the high number of iterations which grows combinatorically with the number of samples this cross-validation method can be very costly. For large datasets one should favor *KFold*, *StratifiedKFold* or *ShuffleSplit*.

Read more in the *User Guide*.

Parameters

p [int] Size of the test sets. Must be strictly greater than the number of samples.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import LeavePOut
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 3, 4])
>>> lpo = LeavePOut(2)
>>> lpo.get_n_splits(X)
6
>>> print(lpo)
LeavePOut(p=2)
>>> for train_index, test_index in lpo.split(X):
...    print("TRAIN:", train_index, "TEST:", test_index)
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
TRAIN: [2 3] TEST: [0 1]
TRAIN: [1 3] TEST: [0 2]
TRAIN: [1 2] TEST: [0 3]
TRAIN: [0 3] TEST: [1 2]
```

```
TRAIN: [0 2] TEST: [1 3]
TRAIN: [0 1] TEST: [2 3]
```

Methods

get_n_splits(self, X[, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *p*)

get_n_splits (*self*, *X*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

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

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

y [array-like, of length n_samples] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.PredefinedSplit

class sklearn.model_selection.**PredefinedSplit** (*test_fold*)
Predefined split cross-validator

Provides train/test indices to split data into train/test sets using a predefined scheme specified by the user with the `test_fold` parameter.

Read more in the *User Guide*.

Parameters

test_fold [array-like, shape (n_samples,)] The entry `test_fold[i]` represents the index of the test set that sample `i` belongs to. It is possible to exclude sample `i` from any test set (i.e. include sample `i` in every training set) by setting `test_fold[i]` equal to -1.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import PredefinedSplit
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> test_fold = [0, 1, -1, 1]
>>> ps = PredefinedSplit(test_fold)
>>> ps.get_n_splits()
2
>>> print(ps)
PredefinedSplit(test_fold=array([ 0,  1, -1,  1]))
>>> for train_index, test_index in ps.split():
...    print("TRAIN:", train_index, "TEST:", test_index)
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
TRAIN: [1 2 3] TEST: [0]
TRAIN: [0 2] TEST: [1 3]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self[, X, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *test_fold*)

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X=None*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.RepeatedKFold

class sklearn.model_selection.**RepeatedKFold**(*n_splits=5*, *n_repeats=10*, *random_state=None*)

Repeated K-Fold cross validator.

Repeats K-Fold n times with different randomization in each repetition.

Read more in the *User Guide*.

Parameters

n_splits [int, default=5] Number of folds. Must be at least 2.

n_repeats [int, default=10] Number of times cross-validator needs to be repeated.

random_state [int, RandomState instance or None, optional, default=None] 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`.

See also:

RepeatedStratifiedKFold Repeats Stratified K-Fold n times.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import RepeatedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> rkf = RepeatedKFold(n_splits=2, n_repeats=2, random_state=2652124)
>>> for train_index, test_index in rkf.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...
TRAIN: [0 1] TEST: [2 3]
TRAIN: [2 3] TEST: [0 1]
TRAIN: [1 2] TEST: [0 3]
TRAIN: [0 3] TEST: [1 2]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generates indices to split data into training and test set.

__init__(*self*, *n_splits=5*, *n_repeats=10*, *random_state=None*)

get_n_splits(*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility. `np.zeros(n_samples)` may be used as a placeholder.

y [object] Always ignored, exists for compatibility. `np.zeros(n_samples)` may be used as a placeholder.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split(*self*, *X*, *y=None*, *groups=None*)
Generates indices to split data into training and test set.

Parameters

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

y [array-like, of length n_samples] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.RepeatedStratifiedKFold

class sklearn.model_selection.**RepeatedStratifiedKFold**(*n_splits=5*, *n_repeats=10*, *random_state=None*)
Repeated Stratified K-Fold cross validator.

Repeats Stratified K-Fold n times with different randomization in each repetition.

Read more in the *User Guide*.

Parameters

n_splits [int, default=5] Number of folds. Must be at least 2.

n_repeats [int, default=10] Number of times cross-validator needs to be repeated.

random_state [None, int or RandomState, default=None] Random state to be used to generate random state for each repetition.

See also:

RepeatedKFold Repeats K-Fold n times.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import RepeatedStratifiedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> rskf = RepeatedStratifiedKFold(n_splits=2, n_repeats=2,
...     random_state=36851234)
>>> for train_index, test_index in rskf.split(X, y):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...
TRAIN: [1 2] TEST: [0 3]
TRAIN: [0 3] TEST: [1 2]
TRAIN: [1 3] TEST: [0 2]
TRAIN: [0 2] TEST: [1 3]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generates indices to split data into training and test set.

__init__ (*self*, *n_splits=5*, *n_repeats=10*, *random_state=None*)

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility. `np.zeros(n_samples)` may be used as a placeholder.

y [object] Always ignored, exists for compatibility. `np.zeros(n_samples)` may be used as a placeholder.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generates indices to split data into training and test set.

Parameters

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

y [array-like, of length n_samples] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

sklearn.model_selection.ShuffleSplit

class sklearn.model_selection.**ShuffleSplit**(*n_splits=10*, *test_size=None*, *train_size=None*, *random_state=None*)

Random permutation cross-validator

Yields indices to split data into training and test sets.

Note: contrary to other cross-validation strategies, random splits do not guarantee that all folds will be different, although this is still very likely for sizeable datasets.

Read more in the *User Guide*.

Parameters

n_splits [int, default 10] Number of re-shuffling & splitting iterations.

test_size [float, int, None, default=None] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If `train_size` is also None, it will be set to 0.1.

train_size [float, int, or None, default=None] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import ShuffleSplit
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [3, 4], [5, 6]])
>>> y = np.array([1, 2, 1, 2, 1, 2])
>>> rs = ShuffleSplit(n_splits=5, test_size=.25, random_state=0)
>>> rs.get_n_splits(X)
5
>>> print(rs)
ShuffleSplit(n_splits=5, random_state=0, test_size=0.25, train_size=None)
>>> for train_index, test_index in rs.split(X):
...    print("TRAIN:", train_index, "TEST:", test_index)
...
TRAIN: [1 3 0 4] TEST: [5 2]
TRAIN: [4 0 2 5] TEST: [1 3]
```

```
TRAIN: [1 2 4 0] TEST: [3 5]
TRAIN: [3 4 1 0] TEST: [5 2]
TRAIN: [3 5 1 0] TEST: [2 4]
>>> rs = ShuffleSplit(n_splits=5, train_size=0.5, test_size=.25,
...                   random_state=0)
>>> for train_index, test_index in rs.split(X):
...    print("TRAIN:", train_index, "TEST:", test_index)
...
TRAIN: [1 3 0] TEST: [5 2]
TRAIN: [4 0 2] TEST: [1 3]
TRAIN: [1 2 4] TEST: [3 5]
TRAIN: [3 4 1] TEST: [5 2]
TRAIN: [3 5 1] TEST: [2 4]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits=10*, *test_size=None*, *train_size=None*, *random_state=None*)

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

y [array-like, shape (n_samples,)] The target variable for supervised learning problems.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples using `sklearn.model_selection.ShuffleSplit`

- *Visualizing cross-validation behavior in scikit-learn*
- *Plotting Learning Curves*
- *Scaling the regularization parameter for SVCs*

`sklearn.model_selection.StratifiedKFold`

class `sklearn.model_selection.`**`StratifiedKFold`**(*n_splits='warn'*, *shuffle=False*, *random_state=None*)

Stratified K-Folds cross-validator

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of KFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

Read more in the *User Guide*.

Parameters

n_splits [int, default=3] Number of folds. Must be at least 2.

Changed in version 0.20: `n_splits` default value will change from 3 to 5 in v0.22.

shuffle [boolean, optional] Whether to shuffle each class's samples before splitting into batches.

random_state [int, RandomState instance or None, optional, default=None] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `shuffle` == True.

See also:

`RepeatedStratifiedKFold` Repeats Stratified K-Fold n times.

Notes

Train and test sizes may be different in each fold, with a difference of at most `n_classes`.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import StratifiedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> skf = StratifiedKFold(n_splits=2)
>>> skf.get_n_splits(X, y)
2
```

```
>>> print(skf)
StratifiedKFold(n_splits=2, random_state=None, shuffle=False)
>>> for train_index, test_index in skf.split(X, y):
...    print("TRAIN:", train_index, "TEST:", test_index)
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
TRAIN: [1 3] TEST: [0 2]
TRAIN: [0 2] TEST: [1 3]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X, y[, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits='warn'*, *shuffle=False*, *random_state=None*)

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

Note that providing `y` is sufficient to generate the splits and hence `np.zeros(n_samples)` may be used as a placeholder for `X` instead of actual training data.

y [array-like, shape (n_samples,)] The target variable for supervised learning problems. Stratification is done based on the y labels.

groups [object] Always ignored, exists for compatibility.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples using `sklearn.model_selection.StratifiedKFold`

- *Recursive feature elimination with cross-validation*
- *Test with permutations the significance of a classification score*
- *GMM covariances*
- *Receiver Operating Characteristic (ROC) with cross validation*
- *Visualizing cross-validation behavior in scikit-learn*

`sklearn.model_selection`.StratifiedShuffleSplit

class `sklearn.model_selection.`**`StratifiedShuffleSplit`**(*n_splits=10*, *test_size=None*, *train_size=None*, *random_state=None*)

Stratified ShuffleSplit cross-validator

Provides train/test indices to split data in train/test sets.

This cross-validation object is a merge of StratifiedKFold and ShuffleSplit, which returns stratified randomized folds. The folds are made by preserving the percentage of samples for each class.

Note: like the ShuffleSplit strategy, stratified random splits do not guarantee that all folds will be different, although this is still very likely for sizeable datasets.

Read more in the *User Guide*.

Parameters

n_splits [int, default 10] Number of re-shuffling & splitting iterations.

test_size [float, int, None, optional (default=None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If `train_size` is also None, it will be set to 0.1.

train_size [float, int, or None, default is None] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import StratifiedShuffleSplit
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 0, 1, 1, 1])
>>> sss = StratifiedShuffleSplit(n_splits=5, test_size=0.5, random_state=0)
>>> sss.get_n_splits(X, y)
5
>>> print(sss)
StratifiedShuffleSplit(n_splits=5, random_state=0, ...)
```

```
>>> for train_index, test_index in sss.split(X, y):
...    print("TRAIN:", train_index, "TEST:", test_index)
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
TRAIN: [5 2 3] TEST: [4 1 0]
TRAIN: [5 1 4] TEST: [0 2 3]
TRAIN: [5 0 2] TEST: [4 3 1]
TRAIN: [4 1 0] TEST: [2 3 5]
TRAIN: [0 5 1] TEST: [3 4 2]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X, y[, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits=10*, *test_size=None*, *train_size=None*, *random_state=None*)

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

Note that providing `y` is sufficient to generate the splits and hence `np.zeros(n_samples)` may be used as a placeholder for `X` instead of actual training data.

y [array-like, shape (n_samples,)] The target variable for supervised learning problems. Stratification is done based on the y labels.

groups [object] Always ignored, exists for compatibility.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Notes

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

Examples using sklearn.model_selection.StratifiedShuffleSplit

- *Visualizing cross-validation behavior in scikit-learn*
- *RBF SVM parameters*

sklearn.model_selection.TimeSeriesSplit

class sklearn.model_selection.**TimeSeriesSplit**(*n_splits='warn'*, *max_train_size=None*)

Time Series cross-validator

Provides train/test indices to split time series data samples that are observed at fixed time intervals, in train/test sets. In each split, test indices must be higher than before, and thus shuffling in cross validator is inappropriate.

This cross-validation object is a variation of *KFold*. In the kth split, it returns first k folds as train set and the (k+1)th fold as test set.

Note that unlike standard cross-validation methods, successive training sets are supersets of those that come before them.

Read more in the *User Guide*.

Parameters

n_splits [int, default=3] Number of splits. Must be at least 2.

Changed in version 0.20: `n_splits` default value will change from 3 to 5 in v0.22.

max_train_size [int, optional] Maximum size for a single training set.

Notes

The training set has size `i * n_samples // (n_splits + 1) + n_samples % (n_splits + 1)` in the `i``th split, with a test set of size ``n_samples//(n_splits + 1)`, where `n_samples` is the number of samples.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import TimeSeriesSplit
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> tscv = TimeSeriesSplit(n_splits=5)
>>> print(tscv)
TimeSeriesSplit(max_train_size=None, n_splits=5)
>>> for train_index, test_index in tscv.split(X):
...    print("TRAIN:", train_index, "TEST:", test_index)
...    X_train, X_test = X[train_index], X[test_index]
...    y_train, y_test = y[train_index], y[test_index]
TRAIN: [0] TEST: [1]
TRAIN: [0 1] TEST: [2]
TRAIN: [0 1 2] TEST: [3]
TRAIN: [0 1 2 3] TEST: [4]
TRAIN: [0 1 2 3 4] TEST: [5]
```

Methods

get_n_splits(self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split(self, X[, y, groups])	Generate indices to split data into training and test set.

__init__ (*self*, *n_splits='warn'*, *max_train_size=None*)

get_n_splits (*self*, *X=None*, *y=None*, *groups=None*)
Returns the number of splitting iterations in the cross-validator

Parameters

X [object] Always ignored, exists for compatibility.

y [object] Always ignored, exists for compatibility.

groups [object] Always ignored, exists for compatibility.

Returns

n_splits [int] Returns the number of splitting iterations in the cross-validator.

split (*self*, *X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

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

y [array-like, shape (n_samples,)] Always ignored, exists for compatibility.

groups [array-like, with shape (n_samples,)] Always ignored, exists for compatibility.

Yields

train [ndarray] The training set indices for that split.

test [ndarray] The testing set indices for that split.

Examples using `sklearn.model_selection.TimeSeriesSplit`

- *Visualizing cross-validation behavior in scikit-learn*

6.26.2 Splitter Functions

model_selection.check_cv([cv, y, classifier])	Input checker utility for building a cross-validator
model_selection.train_test_split(*arrays, ...)	Split arrays or matrices into random train and test subsets

sklearn.model_selection.check_cv

sklearn.model_selection.**check_cv** (*cv='warn'*, *y=None*, *classifier=False*)
Input checker utility for building a cross-validator

Parameters

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if classifier is True and `y` is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value will change from 3-fold to 5-fold in v0.22.

y [array-like, optional] The target variable for supervised learning problems.

classifier [boolean, optional, default False] Whether the task is a classification task, in which case stratified KFold will be used.

Returns

checked_cv [a cross-validator instance.] The return value is a cross-validator which generates the train/test splits via the `split` method.

sklearn.model_selection.train_test_split

sklearn.model_selection.**train_test_split**(**arrays*, ***options*)

Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and `next(ShuffleSplit().split(X, y))` and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.

Read more in the *User Guide*.

Parameters

***arrays** [sequence of indexables with same length / shape[0]] Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.

test_size [float, int or None, optional (default=None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If `train_size` is also None, it will be set to 0.25.

train_size [float, int, or None, (default=None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state [int, RandomState instance or None, optional (default=None)] 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 [boolean, optional (default=True)] Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None.

stratify [array-like or None (default=None)] If not None, data is split in a stratified fashion, using this as the class labels.

Returns

splitting [list, length=2 * len(arrays)] List containing train-test split of inputs.

New in version 0.16: If the input is sparse, the output will be a `scipy.sparse.csr_matrix`. Else, output type is the same as the input type.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> X, y = np.arange(10).reshape((5, 2)), range(5)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> list(y)
[0, 1, 2, 3, 4]
```

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.33, random_state=42)
...
>>> X_train
array([[4, 5],
       [0, 1],
       [6, 7]])
>>> y_train
[2, 0, 3]
>>> X_test
array([[2, 3],
       [8, 9]])
>>> y_test
[1, 4]
```

```
>>> train_test_split(y, shuffle=False)
[[0, 1, 2], [3, 4]]
```

Examples using `sklearn.model_selection.train_test_split`

- *Faces recognition example using eigenfaces and SVMs*
- *Prediction Latency*
- *Probability Calibration curves*
- *Probability calibration of classifiers*
- *Classifier comparison*
- *Column Transformer with Mixed Types*
- *Effect of transforming the targets in regression model*
- *Comparing random forests and the multi-output meta estimator*
- *Early stopping of Gradient Boosting*

- *Feature transformations with ensembles of trees*
- *Gradient Boosting Out-of-Bag estimates*
- *Pipeline Anova SVM*
- *Comparing various online solvers*
- *MNIST classfification using multinomial logistic + L1*
- *Multiclass sparse logisitic regression on newgroups20*
- *Early stopping of Stochastic Gradient Descent*
- *Parameter estimation using grid search with cross-validation*
- *Confusion matrix*
- *Receiver Operating Characteristic (ROC)*
- *Precision-Recall*
- *Classifier Chain*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Restricted Boltzmann Machine features for digit classification*
- *Varying regularization in Multi-layer Perceptron*
- *Using FunctionTransformer to select columns*
- *Importance of Feature Scaling*
- *Map data to a normal distribution*
- *Feature discretization*
- *Understanding the decision tree structure*

6.26.3 Hyper-parameter optimizers

`model_selection.GridSearchCV`(estimator, ...)	Exhaustive search over specified parameter values for an estimator.
`model_selection.ParameterGrid`(param_grid)	Grid of parameters with a discrete number of values for each.
`model_selection.ParameterSampler`(..., [, ...])	Generator on parameters sampled from given distributions.
`model_selection.RandomizedSearchCV`(..., [, ...])	Randomized search on hyper parameters.

sklearn.model_selection.GridSearchCV

class sklearn.model_selection.**GridSearchCV**(*estimator*, *param_grid*, *scoring=None*, *n_jobs=None*, *iid='warn'*, *refit=True*, *cv='warn'*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score='raise-deprecating'*, *return_train_score=False*)

Exhaustive search over specified parameter values for an estimator.

Important members are fit, predict.

GridSearchCV implements a "fit" and a "score" method. It also implements "predict", "predict_proba", "decision_function", "transform" and "inverse_transform" if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

Read more in the *User Guide*.

Parameters

estimator [estimator object.] This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

param_grid [dict or list of dictionaries] Dictionary with parameters names (string) as keys and lists of parameter settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored. This enables searching over any sequence of parameter settings.

scoring [string, callable, list/tuple, dict or None, default: None] A single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*) to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

See *Specifying multiple metrics for evaluation* for an example.

If None, the estimator's score method is used.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

iid [boolean, default='warn'] If True, return the average score across folds, weighted by the number of samples in each test set. In this case, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample, and not the mean loss across the folds. If False, return the average score across folds. Default is True, but will change to False in version 0.22, to correspond to the standard definition of cross-validation.

Changed in version 0.20: Parameter `iid` will change from True to False by default in version 0.22, and will be removed in 0.24.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,

- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

refit [boolean, string, or callable, default=True] Refit an estimator using the best found parameters on the whole dataset.

For multiple metric evaluation, this needs to be a string denoting the scorer that would be used to find the best parameters for refitting the estimator at the end.

Where there are considerations other than maximum score in choosing a best estimator, `refit` can be set to a function which returns the selected `best_index_` given `cv_results_`.

The refitted estimator is made available at the `best_estimator_` attribute and permits using `predict` directly on this `GridSearchCV` instance.

Also for multiple metric evaluation, the attributes `best_index_`, `best_score_` and `best_params_` will only be available if `refit` is set and all of them will be determined w.r.t this specific scorer. `best_score_` is not returned if refit is callable.

See `scoring` parameter to know more about multiple metric evaluation.

Changed in version 0.20: Support for callable added.

verbose [integer] Controls the verbosity: the higher, the more messages.

error_score ['raise' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error. Default is 'raise' but from version 0.22 it will change to np.nan.

return_train_score [boolean, default=False] If `False`, the `cv_results_` attribute will not include training scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance.

Attributes

cv_results_ [dict of numpy (masked) ndarrays] A dict with keys as column headers and values as columns, that can be imported into a pandas `DataFrame`.

For instance the below given table

param_kernel	param_gamma	param_degree	split0_test_score	...	rank_t...
'poly'	–	2	0.80	...	2
'poly'	–	3	0.70	...	4
'rbf'	0.1	–	0.80	...	3
'rbf'	0.2	–	0.93	...	1

will be represented by a `cv_results_` dict of:

```
{
'param_kernel': masked_array(data = ['poly', 'poly', 'rbf', 'rbf'],
                             mask = [False False False False]...)
'param_gamma': masked_array(data = [-- -- 0.1 0.2],
                            mask = [ True  True False False]...),
'param_degree': masked_array(data = [2.0 3.0 -- --],
                             mask = [False False  True  True]...),
'split0_test_score'  : [0.80, 0.70, 0.80, 0.93],
'split1_test_score'  : [0.82, 0.50, 0.70, 0.78],
'mean_test_score'    : [0.81, 0.60, 0.75, 0.85],
'std_test_score'     : [0.01, 0.10, 0.05, 0.08],
'rank_test_score'    : [2, 4, 3, 1],
'split0_train_score' : [0.80, 0.92, 0.70, 0.93],
'split1_train_score' : [0.82, 0.55, 0.70, 0.87],
'mean_train_score'   : [0.81, 0.74, 0.70, 0.90],
'std_train_score'    : [0.01, 0.19, 0.00, 0.03],
'mean_fit_time'      : [0.73, 0.63, 0.43, 0.49],
'std_fit_time'       : [0.01, 0.02, 0.01, 0.01],
'mean_score_time'    : [0.01, 0.06, 0.04, 0.04],
'std_score_time'     : [0.00, 0.00, 0.00, 0.01],
'params'             : [{'kernel': 'poly', 'degree': 2}, ...],
}
```

NOTE

The key `'params'` is used to store a list of parameter settings dicts for all the parameter candidates.

The `mean_fit_time`, `std_fit_time`, `mean_score_time` and `std_score_time` are all in seconds.

For multi-metric evaluation, the scores for all the scorers are available in the `cv_results_` dict at the keys ending with that scorer's name (`'_<scorer_name>'`) instead of `'_score'` shown above. ('split0_test_precision', 'mean_train_precision' etc.)

best_estimator_ [estimator or dict] Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if `refit=False`.

See `refit` parameter for more information on allowed values.

best_score_ [float] Mean cross-validated score of the best_estimator

For multi-metric evaluation, this is present only if `refit` is specified.

best_params_ [dict] Parameter setting that gave the best results on the hold out data.

For multi-metric evaluation, this is present only if `refit` is specified.

best_index_ [int] The index (of the `cv_results_` arrays) which corresponds to the best candidate parameter setting.

The dict at `search.cv_results_['params'][search.best_index_]` gives the parameter setting for the best model, that gives the highest mean score (`search.best_score_`).

For multi-metric evaluation, this is present only if `refit` is specified.

scorer_ [function or a dict] Scorer function used on the held out data to choose the best parameters for the model.

For multi-metric evaluation, this attribute holds the validated `scoring` dict which maps the scorer key to the scorer callable.

n_splits_ [int] The number of cross-validation splits (folds/iterations).

refit_time_ [float] Seconds used for refitting the best model on the whole dataset.

This is present only if `refit` is not False.

See also:

ParameterGrid generates all the combinations of a hyperparameter grid.

sklearn.model_selection.train_test_split utility function to split the data into a development set usable for fitting a GridSearchCV instance and an evaluation set for its final evaluation.

sklearn.metrics.make_scorer Make a scorer from a performance metric or loss function.

Notes

The parameters selected are those that maximize the score of the left out data, unless an explicit score is passed in which case it is used instead.

If *n_jobs* was set to a value higher than one, the data is copied for each point in the grid (and not *n_jobs* times). This is done for efficiency reasons if individual jobs take very little time, but may raise errors if the dataset is large and not enough memory is available. A workaround in this case is to set `pre_dispatch`. Then, the memory is copied only `pre_dispatch` many times. A reasonable value for `pre_dispatch` is `2 * n_jobs`.

Examples

```
>>> from sklearn import svm, datasets
>>> from sklearn.model_selection import GridSearchCV
>>> iris = datasets.load_iris()
>>> parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
>>> svc = svm.SVC(gamma="scale")
>>> clf = GridSearchCV(svc, parameters, cv=5)
>>> clf.fit(iris.data, iris.target)
...
GridSearchCV(cv=5, error_score=...,
       estimator=SVC(C=1.0, cache_size=..., class_weight=..., coef0=...,
                     decision_function_shape='ovr', degree=..., gamma=...,
                     kernel='rbf', max_iter=-1, probability=False,
                     random_state=None, shrinking=True, tol=...,
                     verbose=False),
       iid=..., n_jobs=None,
       param_grid=..., pre_dispatch=..., refit=..., return_train_score=...,
       scoring=..., verbose=...)
>>> sorted(clf.cv_results_.keys())
...
['mean_fit_time', 'mean_score_time', 'mean_test_score',...
 'param_C', 'param_kernel', 'params',...
 'rank_test_score', 'split0_test_score',...
 'split2_test_score', ...
 'std_fit_time', 'std_score_time', 'std_test_score']
```

Methods

decision_function(self, X)	Call decision_function on the estimator with the best found parameters.
fit(self, X[, y, groups])	Run fit with all sets of parameters.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Xt)	Call inverse_transform on the estimator with the best found params.
predict(self, X)	Call predict on the estimator with the best found parameters.
predict_log_proba(self, X)	Call predict_log_proba on the estimator with the best found parameters.
predict_proba(self, X)	Call predict_proba on the estimator with the best found parameters.
score(self, X[, y])	Returns the score on the given data, if the estimator has been refit.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Call transform on the estimator with the best found parameters.

__init__(*self*, *estimator*, *param_grid*, *scoring=None*, *n_jobs=None*, *iid='warn'*, *refit=True*, *cv='warn'*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score='raise-deprecating'*, *return_train_score=False*)

decision_function(*self*, *X*)

Call decision_function on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `decision_function`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

fit(*self*, *X*, *y=None*, *groups=None*, ***fit_params*)

Run fit with all sets of parameters.

Parameters

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

y [array-like, shape = [n_samples] or [n_samples, n_output], optional] Target relative to X for classification or regression; None for unsupervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of the estimator

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *Xt*)

Call inverse_transform on the estimator with the best found params.

Only available if the underlying estimator implements `inverse_transform` and `refit=True`.

Parameters

Xt [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict(*self*, *X*)

Call predict on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_log_proba(*self*, *X*)

Call predict_log_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_log_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_proba(*self*, *X*)

Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

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

Returns the score on the given data, if the estimator has been refit.

This uses the score defined by `scoring` where provided, and the `best_estimator_.score` method otherwise.

Parameters

X [array-like, shape = [n_samples, n_features]] Input data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples] or [n_samples, n_output], optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score [float]

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)

Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Examples using `sklearn.model_selection.GridSearchCV`

- *Comparison of kernel ridge regression and SVR*
- *Faces recognition example using eigenfaces and SVMs*
- *Feature agglomeration vs. univariate selection*
- *Concatenating multiple feature extraction methods*
- *Pipelining: chaining a PCA and a logistic regression*
- *Column Transformer with Mixed Types*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *Shrinkage covariance estimation: LedoitWolf vs OAS and max-likelihood*
- *Model selection with Probabilistic PCA and Factor Analysis (FA)*
- *Cross-validation on diabetes Dataset Exercise*
- *Comparison of kernel ridge and Gaussian process regression*
- *Parameter estimation using grid search with cross-validation*
- *Comparing randomized search and grid search for hyperparameter estimation*
- *Nested versus non-nested cross-validation*
- *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV*
- *Balance model complexity and cross-validated score*
- *Sample pipeline for text feature extraction and evaluation*
- *Kernel Density Estimation*
- *Feature discretization*
- *Scaling the regularization parameter for SVCs*
- *RBF SVM parameters*

sklearn.model_selection.ParameterGrid

class sklearn.model_selection.**ParameterGrid**(*param_grid*)

Grid of parameters with a discrete number of values for each.

Can be used to iterate over parameter value combinations with the Python built-in function iter.

Read more in the *User Guide*.

Parameters

param_grid [dict of string to sequence, or sequence of such] The parameter grid to explore, as a dictionary mapping estimator parameters to sequences of allowed values.

An empty dict signifies default parameters.

A sequence of dicts signifies a sequence of grids to search, and is useful to avoid exploring parameter combinations that make no sense or have no effect. See the examples below.

See also:

GridSearchCV Uses *ParameterGrid* to perform a full parallelized parameter search.

Examples

```
>>> from sklearn.model_selection import ParameterGrid
>>> param_grid = {'a': [1, 2], 'b': [True, False]}
>>> list(ParameterGrid(param_grid)) == (
...    [{'a': 1, 'b': True}, {'a': 1, 'b': False},
...     {'a': 2, 'b': True}, {'a': 2, 'b': False}])
True
```

```
>>> grid = [{'kernel': ['linear']}, {'kernel': ['rbf'], 'gamma': [1, 10]}]
>>> list(ParameterGrid(grid)) == [{'kernel': 'linear'},
...                               {'kernel': 'rbf', 'gamma': 1},
...                               {'kernel': 'rbf', 'gamma': 10}]
True
>>> ParameterGrid(grid)[1] == {'kernel': 'rbf', 'gamma': 1}
True
```

__init__(*self*, *param_grid*)

sklearn.model_selection.ParameterSampler

class sklearn.model_selection.**ParameterSampler**(*param_distributions*, *n_iter*, *random_state=None*)

Generator on parameters sampled from given distributions.

Non-deterministic iterable over random candidate combinations for hyper- parameter search. If all parameters are presented as a list, sampling without replacement is performed. If at least one parameter is given as a distribution, sampling with replacement is used. It is highly recommended to use continuous distributions for continuous parameters.

Note that before SciPy 0.16, the `scipy.stats.distributions` do not accept a custom RNG instance and always use the singleton RNG from `numpy.random`. Hence setting `random_state` will not guarantee a deterministic iteration whenever `scipy.stats` distributions are used to define the parameter search space. Deterministic behavior is however guaranteed from SciPy 0.16 onwards.

Read more in the User Guide.

Parameters

param_distributions [dict] Dictionary where the keys are parameters and values are distributions from which a parameter is to be sampled. Distributions either have to provide a `rvs` function to sample from them, or can be given as a list of values, where a uniform distribution is assumed.

n_iter [integer] Number of parameter settings that are produced.

random_state [int, RandomState instance or None, optional (default=None)] Pseudo random number generator state used for random uniform sampling from lists of possible values instead of scipy.stats distributions. 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`.

Returns

params [dict of string to any] **Yields** dictionaries mapping each estimator parameter to as sampled value.

Examples

```
>>> from sklearn.model_selection import ParameterSampler
>>> from scipy.stats.distributions import expon
>>> import numpy as np
>>> rng = np.random.RandomState(0)
>>> param_grid = {'a':[1, 2], 'b': expon()}
>>> param_list = list(ParameterSampler(param_grid, n_iter=4,
...                                    random_state=rng))
>>> rounded_list = [dict((k, round(v, 6)) for (k, v) in d.items())
...                 for d in param_list]
>>> rounded_list == [{'b': 0.89856, 'a': 1},
...                  {'b': 0.923223, 'a': 1},
...                  {'b': 1.878964, 'a': 2},
...                  {'b': 1.038159, 'a': 2}]
True
```

__init__(*self*, *param_distributions*, *n_iter*, *random_state=None*)

sklearn.model_selection.RandomizedSearchCV

class sklearn.model_selection.**RandomizedSearchCV**(*estimator*, *param_distributions*, *n_iter=10*, *scoring=None*, *n_jobs=None*, *iid='warn'*, *refit=True*, *cv='warn'*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *random_state=None*, *error_score='raise-deprecating'*, *return_train_score=False*)

Randomized search on hyper parameters.

RandomizedSearchCV implements a "fit" and a "score" method. It also implements "predict", "predict_proba", "decision_function", "transform" and "inverse_transform" if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings.

In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by n_iter.

If all parameters are presented as a list, sampling without replacement is performed. If at least one parameter is given as a distribution, sampling with replacement is used. It is highly recommended to use continuous distributions for continuous parameters.

Note that before SciPy 0.16, the `scipy.stats.distributions` do not accept a custom RNG instance and always use the singleton RNG from `numpy.random`. Hence setting `random_state` will not guarantee a deterministic iteration whenever `scipy.stats` distributions are used to define the parameter search space.

Read more in the *User Guide*.

Parameters

estimator [estimator object.] A object of that type is instantiated for each grid point. This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

param_distributions [dict] Dictionary with parameters names (string) as keys and distributions or lists of parameters to try. Distributions must provide a `rvs` method for sampling (such as those from scipy.stats.distributions). If a list is given, it is sampled uniformly.

n_iter [int, default=10] Number of parameter settings that are sampled. n_iter trades off runtime vs quality of the solution.

scoring [string, callable, list/tuple, dict or None, default: None] A single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*) to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

See *Specifying multiple metrics for evaluation* for an example.

If None, the estimator's score method is used.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

iid [boolean, default='warn'] If True, return the average score across folds, weighted by the number of samples in each test set. In this case, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample, and not the mean loss across the folds. If False, return the average score across folds. Default is True, but will change to False in version 0.22, to correspond to the standard definition of cross-validation.

Changed in version 0.20: Parameter `iid` will change from True to False by default in version 0.22, and will be removed in 0.24.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

refit [boolean, string, or callable, default=True] Refit an estimator using the best found parameters on the whole dataset.

For multiple metric evaluation, this needs to be a string denoting the scorer that would be used to find the best parameters for refitting the estimator at the end.

Where there are considerations other than maximum score in choosing a best estimator, `refit` can be set to a function which returns the selected `best_index_` given the `cv_results`.

The refitted estimator is made available at the `best_estimator_` attribute and permits using `predict` directly on this `RandomizedSearchCV` instance.

Also for multiple metric evaluation, the attributes `best_index_`, `best_score_` and `best_params_` will only be available if `refit` is set and all of them will be determined w.r.t this specific scorer. When refit is callable, `best_score_` is disabled.

See `scoring` parameter to know more about multiple metric evaluation.

Changed in version 0.20: Support for callable added.

verbose [integer] Controls the verbosity: the higher, the more messages.

random_state [int, RandomState instance or None, optional, default=None] Pseudo random number generator state used for random uniform sampling from lists of possible values instead of scipy.stats distributions. 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`.

error_score ['raise' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error. Default is 'raise' but from version 0.22 it will change to np.nan.

return_train_score [boolean, default=False] If `False`, the `cv_results_` attribute will not include training scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance.

Attributes

cv_results_ [dict of numpy (masked) ndarrays] A dict with keys as column headers and values as columns, that can be imported into a pandas `DataFrame`.

For instance the below given table

param_kernel	param_gamma	split0_test_score	...	rank_test_score
'rbf'	0.1	0.80	...	2
'rbf'	0.2	0.90	...	1
'rbf'	0.3	0.70	...	1

will be represented by a `cv_results_` dict of:

```
{
'param_kernel' : masked_array(data = ['rbf', 'rbf', 'rbf'],
                              mask = False),
'param_gamma'  : masked_array(data = [0.1 0.2 0.3], mask = False),
'split0_test_score'  : [0.80, 0.90, 0.70],
'split1_test_score'  : [0.82, 0.50, 0.70],
'mean_test_score'    : [0.81, 0.70, 0.70],
'std_test_score'     : [0.01, 0.20, 0.00],
'rank_test_score'    : [3, 1, 1],
'split0_train_score' : [0.80, 0.92, 0.70],
'split1_train_score' : [0.82, 0.55, 0.70],
'mean_train_score'   : [0.81, 0.74, 0.70],
'std_train_score'    : [0.01, 0.19, 0.00],
'mean_fit_time'      : [0.73, 0.63, 0.43],
'std_fit_time'       : [0.01, 0.02, 0.01],
'mean_score_time'    : [0.01, 0.06, 0.04],
'std_score_time'     : [0.00, 0.00, 0.00],
'params'             : [{'kernel' : 'rbf', 'gamma' : 0.1}, ...],
}
```

NOTE

The key `'params'` is used to store a list of parameter settings dicts for all the parameter candidates.

The `mean_fit_time`, `std_fit_time`, `mean_score_time` and `std_score_time` are all in seconds.

For multi-metric evaluation, the scores for all the scorers are available in the `cv_results_` dict at the keys ending with that scorer's name (`'_<scorer_name>'`) instead of `'_score'` shown above. ('split0_test_precision', 'mean_train_precision' etc.)

best_estimator_ [estimator or dict] Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if `refit=False`.

For multi-metric evaluation, this attribute is present only if `refit` is specified.

See `refit` parameter for more information on allowed values.

best_score_ [float] Mean cross-validated score of the best_estimator.

For multi-metric evaluation, this is not available if `refit` is `False`. See `refit` parameter for more information.

best_params_ [dict] Parameter setting that gave the best results on the hold out data.

For multi-metric evaluation, this is not available if `refit` is `False`. See `refit` parameter for more information.

best_index_ [int] The index (of the `cv_results_` arrays) which corresponds to the best candidate parameter setting.

The dict at `search.cv_results_['params'][search.best_index_]` gives the parameter setting for the best model, that gives the highest mean score (`search.best_score_`).

For multi-metric evaluation, this is not available if `refit` is `False`. See `refit` parameter for more information.

scorer_ [function or a dict] Scorer function used on the held out data to choose the best parameters for the model.

For multi-metric evaluation, this attribute holds the validated `scoring` dict which maps the scorer key to the scorer callable.

n_splits_ [int] The number of cross-validation splits (folds/iterations).

refit_time_ [float] Seconds used for refitting the best model on the whole dataset.

This is present only if `refit` is not False.

See also:

GridSearchCV Does exhaustive search over a grid of parameters.

ParameterSampler A generator over parameter settings, constructed from param_distributions.

Notes

The parameters selected are those that maximize the score of the held-out data, according to the scoring parameter.

If *n_jobs* was set to a value higher than one, the data is copied for each parameter setting(and not *n_jobs* times). This is done for efficiency reasons if individual jobs take very little time, but may raise errors if the dataset is large and not enough memory is available. A workaround in this case is to set `pre_dispatch`. Then, the memory is copied only `pre_dispatch` many times. A reasonable value for `pre_dispatch` is `2 * n_jobs`.

Methods

decision_function(self, X)	Call decision_function on the estimator with the best found parameters.
fit(self, X[, y, groups])	Run fit with all sets of parameters.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Xt)	Call inverse_transform on the estimator with the best found params.

Continued on next page

Table 6.206 – continued from previous page

predict(self, X)	Call predict on the estimator with the best found parameters.
predict_log_proba(self, X)	Call predict_log_proba on the estimator with the best found parameters.
predict_proba(self, X)	Call predict_proba on the estimator with the best found parameters.
score(self, X[, y])	Returns the score on the given data, if the estimator has been refit.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Call transform on the estimator with the best found parameters.

__init__ (*self*, *estimator*, *param_distributions*, *n_iter=10*, *scoring=None*, *n_jobs=None*, *iid='warn'*, *refit=True*, *cv='warn'*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *random_state=None*, *error_score='raise-deprecating'*, *return_train_score=False*)

decision_function (*self*, *X*)
Call decision_function on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `decision_function`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

fit (*self*, *X*, *y=None*, *groups=None*, ***fit_params*)
Run fit with all sets of parameters.

Parameters

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

y [array-like, shape = [n_samples] or [n_samples, n_output], optional] Target relative to X for classification or regression; None for unsupervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of the estimator

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform (*self*, *Xt*)
Call inverse_transform on the estimator with the best found params.

Only available if the underlying estimator implements `inverse_transform` and `refit=True`.

Parameters

Xt [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict(*self*, *X*)

Call predict on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_log_proba(*self*, *X*)

Call predict_log_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_log_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

predict_proba(*self*, *X*)

Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_proba`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

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

Returns the score on the given data, if the estimator has been refit.

This uses the score defined by `scoring` where provided, and the `best_estimator_.score` method otherwise.

Parameters

X [array-like, shape = [n_samples, n_features]] Input data, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples] or [n_samples, n_output], optional] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score [float]

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)

Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

Parameters

X [indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Examples using `sklearn.model_selection.RandomizedSearchCV`

- *Comparing randomized search and grid search for hyperparameter estimation*

model_selection.fit_grid_point(X, y, ...[, ...])	Run fit on one set of parameters.

sklearn.model_selection.fit_grid_point

sklearn.model_selection.**fit_grid_point**(*X*, *y*, *estimator*, *parameters*, *train*, *test*, *scorer*, *verbose*, *error_score='raise-deprecating'*, ***fit_params*)

Run fit on one set of parameters.

Parameters

X [array-like, sparse matrix or list] Input data.

y [array-like or None] Targets for input data.

estimator [estimator object] A object of that type is instantiated for each grid point. This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

parameters [dict] Parameters to be set on estimator for this grid point.

train [ndarray, dtype int or bool] Boolean mask or indices for training set.

test [ndarray, dtype int or bool] Boolean mask or indices for test set.

scorer [callable or None] The scorer callable object / function must have its signature as `scorer(estimator, X, y)`.

If `None` the estimator's score method is used.

verbose [int] Verbosity level.

****fit_params** [kwargs] Additional parameter passed to the fit function of the estimator.

error_score ['raise' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error. Default is 'raise' but from version 0.22 it will change to np.nan.

Returns

score [float] Score of this parameter setting on given test split.

parameters [dict] The parameters that have been evaluated.

n_samples_test [int] Number of test samples in this split.

6.26.4 Model validation

model_selection.cross_validate(estimator, X)	Evaluate metric(s) by cross-validation and also record fit/score times.
model_selection.cross_val_predict(estimator, X)	Generate cross-validated estimates for each input data point
model_selection.cross_val_score(estimator, X)	Evaluate a score by cross-validation
model_selection.learning_curve(estimator, X, y)	Learning curve.
model_selection.permutation_test_score(...)	Evaluate the significance of a cross-validated score with permutations
model_selection.validation_curve(estimator, ...)	Validation curve.

sklearn.model_selection.cross_validate

sklearn.model_selection.**cross_validate**(*estimator*, *X*, *y=None*, *groups=None*, *scoring=None*, *cv='warn'*, *n_jobs=None*, *verbose=0*, *fit_params=None*, *pre_dispatch='2*n_jobs'*, *return_train_score=False*, *return_estimator=False*, *error_score='raise-deprecating'*)

Evaluate metric(s) by cross-validation and also record fit/score times.

Read more in the *User Guide*.

Parameters

estimator [estimator object implementing 'fit'] The object to use to fit the data.

X [array-like] The data to fit. Can be for example a list, or an array.

y [array-like, optional, default: None] The target variable to try to predict in the case of supervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

scoring [string, callable, list/tuple, dict or None, default: None] A single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*) to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

See *Specifying multiple metrics for evaluation* for an example.

If None, the estimator's score method is used.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,

- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [integer, optional] The verbosity level.

fit_params [dict, optional] Parameters to pass to the fit method of the estimator.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

return_train_score [boolean, default=False] Whether to include train scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance.

return_estimator [boolean, default False] Whether to return the estimators fitted on each split.

error_score ['raise' | 'raise-deprecating' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If set to 'raise-deprecating', a FutureWarning is printed before the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error. Default is 'raise-deprecating' but from version 0.22 it will change to np.nan.

Returns

scores [dict of float arrays of shape=(n_splits,)] Array of scores of the estimator for each run of the cross validation.

A dict of arrays containing the score/time arrays for each scorer is returned. The possible keys for this `dict` are:

`test_score` The score array for test scores on each cv split.

`train_score` The score array for train scores on each cv split. This is available only if `return_train_score` parameter is `True`.

`fit_time` The time for fitting the estimator on the train set for each cv split.

`score_time` The time for scoring the estimator on the test set for each cv split. (Note time for scoring on the train set is not included even if `return_train_score` is set to `True`

estimator The estimator objects for each cv split. This is available only if `return_estimator` parameter is set to `True`.

See also:

sklearn.model_selection.cross_val_score Run cross-validation for single metric evaluation.

sklearn.model_selection.cross_val_predict Get predictions from each split of cross-validation for diagnostic purposes.

sklearn.metrics.make_scorer Make a scorer from a performance metric or loss function.

Examples

```
>>> from sklearn import datasets, linear_model
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics.scorer import make_scorer
>>> from sklearn.metrics import confusion_matrix
>>> from sklearn.svm import LinearSVC
>>> diabetes = datasets.load_diabetes()
>>> X = diabetes.data[:150]
>>> y = diabetes.target[:150]
>>> lasso = linear_model.Lasso()
```

Single metric evaluation using `cross_validate`

```
>>> cv_results = cross_validate(lasso, X, y, cv=3)
>>> sorted(cv_results.keys())
['fit_time', 'score_time', 'test_score']
>>> cv_results['test_score']
array([0.33150734, 0.08022311, 0.03531764])
```

Multiple metric evaluation using `cross_validate` (please refer the `scoring` parameter doc for more information)

```
>>> scores = cross_validate(lasso, X, y, cv=3,
...                         scoring=('r2', 'neg_mean_squared_error'),
...                         return_train_score=True)
>>> print(scores['test_neg_mean_squared_error'])
[-3635.5... -3573.3... -6114.7...]
>>> print(scores['train_r2'])
[0.28010158 0.39088426 0.22784852]
```

sklearn.model_selection.cross_val_predict

sklearn.model_selection.**cross_val_predict**(*estimator*, *X*, *y=None*, *groups=None*, *cv='warn'*, *n_jobs=None*, *verbose=0*, *fit_params=None*, *pre_dispatch='2*n_jobs'*, *method='predict'*)

Generate cross-validated estimates for each input data point

The data is split according to the cv parameter. Each sample belongs to exactly one test set, and its prediction is computed with an estimator fitted on the corresponding training set.

Passing these predictions into an evaluation metric may not be a valid way to measure generalization performance. Results can differ from *cross_validate* and *cross_val_score* unless all tests sets have equal size and the metric decomposes over samples.

Read more in the *User Guide*.

Parameters

estimator [estimator object implementing 'fit' and 'predict'] The object to use to fit the data.

X [array-like] The data to fit. Can be, for example a list, or an array at least 2d.

y [array-like, optional, default: None] The target variable to try to predict in the case of supervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., `GroupKFold`).

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [integer, optional] The verbosity level.

fit_params [dict, optional] Parameters to pass to the fit method of the estimator.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

method [string, optional, default: 'predict'] Invokes the passed method name of the passed estimator. For method='predict_proba', the columns correspond to the classes in sorted order.

Returns

predictions [ndarray] This is the result of calling `method`

See also:

`cross_val_score` calculate score for each CV split

cross_validate calculate one or more scores and timings for each CV split

Notes

In the case that one or more classes are absent in a training portion, a default score needs to be assigned to all instances for that class if `method` produces columns per class, as in {'decision_function', 'predict_proba', 'predict_log_proba'}. For `predict_proba` this value is 0. In order to ensure finite output, we approximate negative infinity by the minimum finite float value for the dtype in other cases.

Examples

```
>>> from sklearn import datasets, linear_model
>>> from sklearn.model_selection import cross_val_predict
>>> diabetes = datasets.load_diabetes()
>>> X = diabetes.data[:150]
>>> y = diabetes.target[:150]
>>> lasso = linear_model.Lasso()
>>> y_pred = cross_val_predict(lasso, X, y, cv=3)
```

Examples using `sklearn.model_selection.cross_val_predict`

- *Plotting Cross-Validated Predictions*

sklearn.model_selection.cross_val_score

sklearn.model_selection.**cross_val_score**(*estimator*, *X*, *y=None*, *groups=None*, *scoring=None*, *cv='warn'*, *n_jobs=None*, *verbose=0*, *fit_params=None*, *pre_dispatch='2*n_jobs'*, *error_score='raise-deprecating'*)

Evaluate a score by cross-validation

Read more in the *User Guide*.

Parameters

estimator [estimator object implementing 'fit'] The object to use to fit the data.

X [array-like] The data to fit. Can be for example a list, or an array.

y [array-like, optional, default: None] The target variable to try to predict in the case of supervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

scoring [string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)` which should return only a single value.

Similar to *cross_validate* but only a single metric is permitted.

If None, the estimator's default scorer (if available) is used.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [integer, optional] The verbosity level.

fit_params [dict, optional] Parameters to pass to the fit method of the estimator.

pre_dispatch [int, or string, optional] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A string, giving an expression as a function of n_jobs, as in '2*n_jobs'

error_score ['raise' | 'raise-deprecating' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If set to 'raise-deprecating', a FutureWarning is printed before the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error. Default is 'raise-deprecating' but from version 0.22 it will change to np.nan.

Returns

scores [array of float, shape=(len(list(cv)),)] Array of scores of the estimator for each run of the cross validation.

See also:

sklearn.model_selection.cross_validate To run cross-validation on multiple metrics and also to return train scores, fit times and score times.

sklearn.model_selection.cross_val_predict Get predictions from each split of cross-validation for diagnostic purposes.

sklearn.metrics.make_scorer Make a scorer from a performance metric or loss function.

Examples

```
>>> from sklearn import datasets, linear_model
>>> from sklearn.model_selection import cross_val_score
>>> diabetes = datasets.load_diabetes()
>>> X = diabetes.data[:150]
>>> y = diabetes.target[:150]
>>> lasso = linear_model.Lasso()
>>> print(cross_val_score(lasso, X, y, cv=3))
[0.33150734 0.08022311 0.03531764]
```

Examples using `sklearn.model_selection.cross_val_score`

- *Model selection with Probabilistic PCA and Factor Analysis (FA)*
- *Cross-validation on Digits Dataset Exercise*
- *Imputing missing values with variants of IterativeImputer*
- *Imputing missing values before building an estimator*
- *Underfitting vs. Overfitting*
- *Nested versus non-nested cross-validation*
- *SVM-Anova: SVM with univariate feature selection*

`sklearn.model_selection.learning_curve`

sklearn.model_selection.**learning_curve**(*estimator*, *X*, *y*, *groups=None*, *train_sizes=array([0.1, 0.325, 0.55, 0.775, 1.])*, *cv='warn'*, *scoring=None*, *exploit_incremental_learning=False*, *n_jobs=None*, *pre_dispatch='all'*, *verbose=0*, *shuffle=False*, *random_state=None*, *error_score='raise-deprecating'*)

Learning curve.

Determines cross-validated training and test scores for different training set sizes.

A cross-validation generator splits the whole dataset k times in training and test data. Subsets of the training set with varying sizes will be used to train the estimator and a score for each training subset size and the test set will be computed. Afterwards, the scores will be averaged over all k runs for each training subset size.

Read more in the *User Guide*.

Parameters

estimator [object type that implements the "fit" and "predict" methods] An object of that type which is cloned for each validation.

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

y [array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

train_sizes [array-like, shape (n_ticks,), dtype float or int] Relative or absolute numbers of training examples that will be used to generate the learning curve. If the dtype is float, it is regarded as a fraction of the maximum size of the training set (that is determined by the selected validation method), i.e. it has to be within (0, 1]. Otherwise it is interpreted as absolute sizes of the training sets. Note that for classification the number of samples usually have to be big enough to contain at least one sample from each class. (default: np.linspace(0.1, 1.0, 5))

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

scoring [string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

exploit_incremental_learning [boolean, optional, default: False] If the estimator supports incremental learning, this will be used to speed up fitting for different training set sizes.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

pre_dispatch [integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like '2*n_jobs'.

verbose [integer, optional] Controls the verbosity: the higher, the more messages.

shuffle [boolean, optional] Whether to shuffle training data before taking prefixes of it based on''train_sizes''.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `shuffle` is True.

error_score ['raise' | 'raise-deprecating' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If set to 'raise-deprecating', a FutureWarning is printed before the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error. Default is 'raise-deprecating' but from version 0.22 it will change to np.nan.

Returns

train_sizes_abs [array, shape (n_unique_ticks,), dtype int] Numbers of training examples that has been used to generate the learning curve. Note that the number of ticks might be less than n_ticks because duplicate entries will be removed.

train_scores [array, shape (n_ticks, n_cv_folds)] Scores on training sets.

test_scores [array, shape (n_ticks, n_cv_folds)] Scores on test set.

Notes

See *examples/model_selection/plot_learning_curve.py*

Examples using `sklearn.model_selection.learning_curve`

- *Comparison of kernel ridge regression and SVR*
- *Plotting Learning Curves*

`sklearn.model_selection.`permutation_test_score

`sklearn.model_selection.`**`permutation_test_score`**(*estimator*, *X*, *y*, *groups=None*, *cv='warn'*, *n_permutations=100*, *n_jobs=None*, *random_state=0*, *verbose=0*, *scoring=None*)

Evaluate the significance of a cross-validated score with permutations

Read more in the *User Guide*.

Parameters

estimator [estimator object implementing 'fit'] The object to use to fit the data.

X [array-like of shape at least 2D] The data to fit.

y [array-like] The target variable to try to predict in the case of supervised learning.

groups [array-like, with shape (n_samples,), optional] Labels to constrain permutation within groups, i.e. `y` values are permuted among samples with the same group identifier. When not specified, `y` values are permuted among all samples.

When a grouped cross-validator is used, the group labels are also passed on to the `split` method of the cross-validator. The cross-validator uses them for grouping the samples while splitting the dataset into train/test set.

scoring [string, callable or None, optional, default: None] A single string (see *The scoring parameter: defining model evaluation rules*) or a callable (see *Defining your scoring strategy from metric functions*) to evaluate the predictions on the test set.

If None the estimator's score method is used.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *`StratifiedKFold`* is used. In all other cases, *`KFold`* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

n_permutations [integer, optional] Number of times to permute `y`.

n_jobs [int or None, optional (default=None)] The number of CPUs to use to do the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

random_state [int, RandomState instance or None, optional (default=0)] 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`.

verbose [integer, optional] The verbosity level.

Returns

score [float] The true score without permuting targets.

permutation_scores [array, shape (n_permutations,)] The scores obtained for each permutations.

pvalue [float] The p-value, which approximates the probability that the score would be obtained by chance. This is calculated as:

```
(C + 1) / (n_permutations + 1)
```

Where C is the number of permutations whose score >= the true score.

The best possible p-value is 1/(n_permutations + 1), the worst is 1.0.

Notes

This function implements Test 1 in:

Ojala and Garriga. Permutation Tests for Studying Classifier Performance. The Journal of Machine Learning Research (2010) vol. 11

Examples using `sklearn.model_selection.permutation_test_score`

- *Test with permutations the significance of a classification score*

`sklearn.model_selection.validation_curve`

`sklearn.model_selection.`**`validation_curve`**(*estimator*, *X*, *y*, *param_name*, *param_range*, *groups=None*, *cv='warn'*, *scoring=None*, *n_jobs=None*, *pre_dispatch='all'*, *verbose=0*, *error_score='raise-deprecating'*)

Validation curve.

Determine training and test scores for varying parameter values.

Compute scores for an estimator with different values of a specified parameter. This is similar to grid search with one parameter. However, this will also compute training scores and is merely a utility for plotting the results.

Read more in the *User Guide*.

Parameters

estimator [object type that implements the "fit" and "predict" methods] An object of that type which is cloned for each validation.

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

y [array-like, shape (n_samples) or (n_samples, n_features), optional] Target relative to X for classification or regression; None for unsupervised learning.

param_name [string] Name of the parameter that will be varied.

param_range [array-like, shape (n_values,)] The values of the parameter that will be evaluated.

groups [array-like, with shape (n_samples,), optional] Group labels for the samples used while splitting the dataset into train/test set. Only used in conjunction with a "Group" *cv* instance (e.g., *GroupKFold*).

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross validation,
- integer, to specify the number of folds in a `(Stratified)KFold`,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, *StratifiedKFold* is used. In all other cases, *KFold* is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value if None will change from 3-fold to 5-fold in v0.22.

scoring [string, callable or None, optional, default: None] A string (see model evaluation documentation) or a scorer callable object / function with signature `scorer(estimator, X, y)`.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

pre_dispatch [integer or string, optional] Number of predispatched jobs for parallel execution (default is all). The option can reduce the allocated memory. The string can be an expression like '2*n_jobs'.

verbose [integer, optional] Controls the verbosity: the higher, the more messages.

error_score ['raise' | 'raise-deprecating' or numeric] Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If set to 'raise-deprecating', a FutureWarning is printed before the error is raised. If a numeric value is given, FitFailedWarning is raised. This parameter does not affect the refit step, which will always raise the error. Default is 'raise-deprecating' but from version 0.22 it will change to np.nan.

Returns

train_scores [array, shape (n_ticks, n_cv_folds)] Scores on training sets.

test_scores [array, shape (n_ticks, n_cv_folds)] Scores on test set.

Notes

See *Plotting Validation Curves*

Examples using `sklearn.model_selection.validation_curve`

- *Plotting Validation Curves*

6.27 `sklearn.multiclass`: Multiclass and multilabel classification

6.27.1 Multiclass and multilabel classification strategies

This module implements multiclass learning algorithms:

- one-vs-the-rest / one-vs-all
- one-vs-one
- error correcting output codes

The estimators provided in this module are meta-estimators: they require a base estimator to be provided in their constructor. For example, it is possible to use these estimators to turn a binary classifier or a regressor into a multiclass classifier. It is also possible to use these estimators with multiclass estimators in the hope that their accuracy or runtime performance improves.

All classifiers in scikit-learn implement multiclass classification; you only need to use this module if you want to experiment with custom multiclass strategies.

The one-vs-the-rest meta-classifier also implements a *predict_proba* method, so long as such a method is implemented by the base classifier. This method returns probabilities of class membership in both the single label and multilabel case. Note that in the multilabel case, probabilities are the marginal probability that a given sample falls in the given class. As such, in the multilabel case the sum of these probabilities over all possible labels for a given sample *will not* sum to unity, as they do in the single label case.

User guide: See the *Multiclass and multilabel algorithms* section for further details.

multiclass.OneVsRestClassifier(estimator[, ...])	One-vs-the-rest (OvR) multiclass/multilabel strategy
multiclass.OneVsOneClassifier(estimator[, ...])	One-vs-one multiclass strategy
multiclass.OutputCodeClassifier(estimator[, ...])	(Error-Correcting) Output-Code multiclass strategy

6.27.2 `sklearn.multiclass.OneVsRestClassifier`

class `sklearn.multiclass.`**`OneVsRestClassifier`**(*estimator*, *n_jobs=None*)

One-vs-the-rest (OvR) multiclass/multilabel strategy

Also known as one-vs-all, this strategy consists in fitting one classifier per class. For each classifier, the class is

fitted against all the other classes. In addition to its computational efficiency (only `n_classes` classifiers are needed), one advantage of this approach is its interpretability. Since each class is represented by one and one classifier only, it is possible to gain knowledge about the class by inspecting its corresponding classifier. This is the most commonly used strategy for multiclass classification and is a fair default choice.

This strategy can also be used for multilabel learning, where a classifier is used to predict multiple labels for instance, by fitting on a 2-d matrix in which cell [i, j] is 1 if sample i has label j and 0 otherwise.

In the multilabel learning literature, OvR is also known as the binary relevance method.

Read more in the *User Guide*.

Parameters

estimator [estimator object] An estimator object implementing *fit* and one of *decision_function* or *predict_proba*.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

estimators_ [list of `n_classes` estimators] Estimators used for predictions.

classes_ [array, shape = [`n_classes`]] Class labels.

label_binarizer_ [LabelBinarizer object] Object used to transform multiclass labels to binary labels and vice-versa.

multilabel_ [boolean] Whether this is a multilabel classifier

Methods

decision_function(self, X)	Returns the distance of each sample from the decision boundary for each class.
fit(self, X, y)	Fit underlying estimators.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes])	Partially fit underlying estimators
predict(self, X)	Predict multi-class targets using underlying estimators.
predict_proba(self, X)	Probability estimates.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *estimator*, *n_jobs=None*)

decision_function (*self*, *X*)

Returns the distance of each sample from the decision boundary for each class. This can only be used with estimators which implement the decision_function method.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

T [array-like, shape = [n_samples, n_classes]]

fit (*self*, *X*, *y*)

Fit underlying estimators.

Parameters

X [(sparse) array-like, shape = [n_samples, n_features]] Data.

y [(sparse) array-like, shape = [n_samples,], [n_samples, n_classes]] Multi-class targets. An indicator matrix turns on multilabel classification.

Returns

self

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

multilabel_
Whether this is a multilabel classifier

partial_fit (*self*, *X*, *y*, *classes=None*)
Partially fit underlying estimators

Should be used when memory is inefficient to train all data. Chunks of data can be passed in several iteration.

Parameters

X [(sparse) array-like, shape = [n_samples, n_features]] Data.

y [(sparse) array-like, shape = [n_samples,], [n_samples, n_classes]] Multi-class targets. An indicator matrix turns on multilabel classification.

classes [array, shape (n_classes,)] Classes across all calls to partial_fit. Can be obtained via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is only required in the first call of partial_fit and can be omitted in the subsequent calls.

Returns

self

predict (*self*, *X*)
Predict multi-class targets using underlying estimators.

Parameters

X [(sparse) array-like, shape = [n_samples, n_features]] Data.

Returns

y [(sparse) array-like, shape = [n_samples,], [n_samples, n_classes].] Predicted multi-class targets.

predict_proba (*self*, *X*)
Probability estimates.

The returned estimates for all classes are ordered by label of classes.

Note that in the multilabel case, each sample can have any number of labels. This returns the marginal probability that the given sample has the label in question. For example, it is entirely consistent that two labels both have a 90% probability of applying to a given sample.

In the single label multiclass case, the rows of the returned matrix sum to 1.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

T [(sparse) array-like, shape = [n_samples, n_classes]] Returns the probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

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

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.multiclass.OneVsRestClassifier`

- *Multilabel classification*
- *Receiver Operating Characteristic (ROC)*
- *Precision-Recall*
- *Classifier Chain*

6.27.3 `sklearn.multiclass`.OneVsOneClassifier

class `sklearn.multiclass.`**OneVsOneClassifier** (*estimator*, *n_jobs=None*)

One-vs-one multiclass strategy

This strategy consists in fitting one classifier per class pair. At prediction time, the class which received the most votes is selected. Since it requires to fit `n_classes * (n_classes - 1) / 2` classifiers, this method is usually slower than one-vs-the-rest, due to its O(n_classes^2) complexity. However, this method may be advantageous for algorithms such as kernel algorithms which don't scale well with *n_samples*. This is because

each individual learning problem only involves a small subset of the data whereas, with one-vs-the-rest, the complete dataset is used `n_classes` times.

Read more in the *User Guide*.

Parameters

estimator [estimator object] An estimator object implementing *fit* and one of *decision_function* or *predict_proba*.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

estimators_ [list of `n_classes * (n_classes - 1) / 2` estimators] Estimators used for predictions.

classes_ [numpy array of shape [n_classes]] Array containing labels.

pairwise_indices_ [list, length = `len(estimators_)`, or `None`] Indices of samples used when training the estimators. `None` when `estimator` does not have `_pairwise` attribute.

Methods

decision_function(self, X)	Decision function for the OneVsOneClassifier.
fit(self, X, y)	Fit underlying estimators.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes])	Partially fit underlying estimators
predict(self, X)	Estimate the best class label for each sample in X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *estimator*, *n_jobs=None*)

decision_function(*self*, *X*)

Decision function for the OneVsOneClassifier.

The decision values for the samples are computed by adding the normalized sum of pair-wise classification confidence levels to the votes in order to disambiguate between the decision values when the votes for all the classes are equal leading to a tie.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

Y [array-like, shape = [n_samples, n_classes]]

fit(*self*, *X*, *y*)

Fit underlying estimators.

Parameters

X [(sparse) array-like, shape = [n_samples, n_features]] Data.

y [array-like, shape = [n_samples]] Multi-class targets.

Returns

self

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit (*self*, *X*, *y*, *classes=None*)
Partially fit underlying estimators

Should be used when memory is inefficient to train all data. Chunks of data can be passed in several iteration, where the first call should have an array of all target variables.

Parameters

X [(sparse) array-like, shape = [n_samples, n_features]] Data.

y [array-like, shape = [n_samples]] Multi-class targets.

classes [array, shape (n_classes,)] Classes across all calls to partial_fit. Can be obtained via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is only required in the first call of partial_fit and can be omitted in the subsequent calls.

Returns

self

predict (*self*, *X*)
Estimate the best class label for each sample in X.

This is implemented as `argmax(decision_function(X), axis=1)` which will return the label of the class with most votes by estimators predicting the outcome of a decision for each possible class pair.

Parameters

X [(sparse) array-like, shape = [n_samples, n_features]] Data.

Returns

y [numpy array of shape [n_samples]] Predicted multi-class targets.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

6.27.4 sklearn.multiclass.OutputCodeClassifier

class sklearn.multiclass.**OutputCodeClassifier**(*estimator*, *code_size=1.5*, *random_state=None*, *n_jobs=None*)
(Error-Correcting) Output-Code multiclass strategy

Output-code based strategies consist in representing each class with a binary code (an array of 0s and 1s). At fitting time, one binary classifier per bit in the code book is fitted. At prediction time, the classifiers are used to project new points in the class space and the class closest to the points is chosen. The main advantage of these strategies is that the number of classifiers used can be controlled by the user, either for compressing the model (0 < code_size < 1) or for making the model more robust to errors (code_size > 1). See the documentation for more details.

Read more in the *User Guide*.

Parameters

estimator [estimator object] An estimator object implementing *fit* and one of *decision_function* or *predict_proba*.

code_size [float] Percentage of the number of classes to be used to create the code book. A number between 0 and 1 will require fewer classifiers than one-vs-the-rest. A number greater than 1 will require more classifiers than one-vs-the-rest.

random_state [int, RandomState instance or None, optional, default: None] The generator used to initialize the codebook. 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`.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

estimators_ [list of `int(n_classes * code_size)` estimators] Estimators used for predictions.

classes_ [numpy array of shape [n_classes]] Array containing labels.

code_book_ [numpy array of shape [n_classes, code_size]] Binary array containing the code of each class.

References

[R2eddaeec0849-1], *[R2eddaeec0849-2]*, *[R2eddaeec0849-3]*

Methods

fit(self, X, y)	Fit underlying estimators.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict multi-class targets using underlying estimators.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *estimator*, *code_size=1.5*, *random_state=None*, *n_jobs=None*)

fit (*self*, *X*, *y*)
Fit underlying estimators.

Parameters

X [(sparse) array-like, shape = [n_samples, n_features]] Data.

y [numpy array of shape [n_samples]] Multi-class targets.

Returns

self

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict multi-class targets using underlying estimators.

Parameters

X [(sparse) array-like, shape = [n_samples, n_features]] Data.

Returns

y [numpy array of shape [n_samples]] Predicted multi-class targets.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

6.28 sklearn.multioutput: Multioutput regression and classification

This module implements multioutput regression and classification.

The estimators provided in this module are meta-estimators: they require a base estimator to be provided in their constructor. The meta-estimator extends single output estimators to multioutput estimators.

User guide: See the *Multiclass and multilabel algorithms* section for further details.

multioutput.ClassifierChain(base_estimator)	A multi-label model that arranges binary classifiers into a chain.
multioutput.MultiOutputRegressor(estimator)	Multi target regression
multioutput.MultiOutputClassifier(estimator)	Multi target classification
multioutput.RegressorChain(base_estimator[, ...])	A multi-label model that arranges regressions into a chain.

6.28.1 sklearn.multioutput.ClassifierChain

class sklearn.multioutput.**ClassifierChain**(*base_estimator*, *order=None*, *cv=None*, *random_state=None*)
A multi-label model that arranges binary classifiers into a chain.

Each model makes a prediction in the order specified by the chain using all of the available features provided to the model plus the predictions of models that are earlier in the chain.

Read more in the *User Guide*.

Parameters

base_estimator [estimator] The base estimator from which the classifier chain is built.

order [array-like, shape=[n_outputs] or 'random', optional] By default the order will be determined by the order of columns in the label matrix Y.:

```
order = [0, 1, 2, ..., Y.shape[1] - 1]
```

The order of the chain can be explicitly set by providing a list of integers. For example, for a chain of length 5.:

```
order = [1, 3, 2, 4, 0]
```

means that the first model in the chain will make predictions for column 1 in the Y matrix, the second model will make predictions for column 3, etc.

If order is 'random' a random ordering will be used.

cv [int, cross-validation generator or an iterable, optional (default=None)] Determines whether to use cross validated predictions or true labels for the results of previous estimators in the chain. If cv is None the true labels are used when fitting. Otherwise possible inputs for cv are:

- integer, to specify the number of folds in a (Stratified)KFold,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

The random number generator is used to generate random chain orders.

Attributes

classes_ [list] A list of arrays of length `len(estimators_)` containing the class labels for each estimator in the chain.

estimators_ [list] A list of clones of base_estimator.

order_ [list] The order of labels in the classifier chain.

See also:

RegressorChain Equivalent for regression

MultioutputClassifier Classifies each output independently rather than chaining.

References

Jesse Read, Bernhard Pfahringer, Geoff Holmes, Eibe Frank, "Classifier Chains for Multi-label Classification", 2009.

Methods

decision_function(self, X)	Evaluate the decision_function of the models in the chain.
fit(self, X, Y)	Fit the model to data matrix X and targets Y.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict on the data matrix X using the ClassifierChain model.
predict_proba(self, X)	Predict probability estimates.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *base_estimator*, *order=None*, *cv=None*, *random_state=None*)

decision_function(*self*, *X*)
Evaluate the decision_function of the models in the chain.

Parameters

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

Returns

Y_decision [array-like, shape (n_samples, n_classes)] Returns the decision function of the sample for each model in the chain.

fit (*self*, *X*, *Y*)
Fit the model to data matrix X and targets Y.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Y [array-like, shape (n_samples, n_classes)] The target values.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict on the data matrix X using the ClassifierChain model.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Returns

Y_pred [array-like, shape (n_samples, n_classes)] The predicted values.

predict_proba (*self*, *X*)
Predict probability estimates.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)]

Returns

Y_prob [array-like, shape (n_samples, n_classes)]

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.multioutput.ClassifierChain`

- *Classifier Chain*

6.28.2 `sklearn.multioutput`.MultiOutputRegressor

class `sklearn.multioutput`.**MultiOutputRegressor**(*estimator*, *n_jobs=None*)
Multi target regression

This strategy consists of fitting one regressor per target. This is a simple strategy for extending regressors that do not natively support multi-target regression.

Parameters

estimator [estimator object] An estimator object implementing *fit* and *predict*.

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

When individual estimators are fast to train or predict using `n_jobs>1` can result in slower performance due to the overhead of spawning processes.

Attributes

estimators_ [list of `n_output` estimators] Estimators used for predictions.

Methods

fit(self, X, y[, sample_weight])	Fit the model to data.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, sample_weight])	Incrementally fit the model to data.
predict(self, X)	Predict multi-output variable using a model trained for each target variable.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *estimator*, *n_jobs=None*)

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the model to data. Fit a separate model for each output variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets. An indicator matrix turns on multilabel estimation.

sample_weight [array-like, shape = (n_samples) or None] Sample weights. If None, then samples are equally weighted. Only supported if the underlying regressor supports sample weights.

Returns

self [object]

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit (*self*, *X*, *y*, *sample_weight=None*)

Incrementally fit the model to data. Fit a separate model for each output variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets.

sample_weight [array-like, shape = (n_samples) or None] Sample weights. If None, then samples are equally weighted. Only supported if the underlying regressor supports sample weights.

Returns

self [object]

predict (*self*, *X*)

Predict multi-output variable using a model trained for each target variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

Returns

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets predicted across multiple predictors. Note: Separate models are generated for each predictor.

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

Returns 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 regression sum of squares ((y_true - y_true.mean()) ** 2).sum(). 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, shape (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape [n_samples], optional] Sample weights.

Returns

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

Notes

R^2 is calculated by weighting all the targets equally using `multioutput='uniform_average'`.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.multioutput.MultiOutputRegressor`

- *Comparing random forests and the multi-output meta estimator*

6.28.3 `sklearn.multioutput`.MultiOutputClassifier

class `sklearn.multioutput.`**MultiOutputClassifier**(*estimator*, *n_jobs=None*)
Multi target classification

This strategy consists of fitting one classifier per target. This is a simple strategy for extending classifiers that do not natively support multi-target classification

Parameters

estimator [estimator object] An estimator object implementing *fit*, *score* and *predict_proba*.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. It does each target variable in y in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

estimators_ [list of `n_output` estimators] Estimators used for predictions.

Methods

fit(self, X, y[, sample_weight])	Fit the model to data.
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incrementally fit the model to data.

Continued on next page

Table 6.216 – continued from previous page

predict(self, X)	Predict multi-output variable using a model trained for each target variable.
predict_proba(self, X)	Probability estimates.
score(self, X, y)	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *estimator*, *n_jobs=None*)

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

Fit the model to data. Fit a separate model for each output variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets. An indicator matrix turns on multilabel estimation.

sample_weight [array-like, shape = (n_samples) or None] Sample weights. If None, then samples are equally weighted. Only supported if the underlying regressor supports sample weights.

Returns

self [object]

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit (*self*, *X*, *y*, *classes=None*, *sample_weight=None*)

Incrementally fit the model to data. Fit a separate model for each output variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets.

classes [list of numpy arrays, shape (n_outputs)] Each array is unique classes for one output in str/int Can be obtained by via `[np.unique(y[:, i]) for i in range(y.shape[1])]`, where y is the target matrix of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in `classes`.

sample_weight [array-like, shape = (n_samples) or None] Sample weights. If None, then samples are equally weighted. Only supported if the underlying regressor supports sample weights.

Returns

self [object]

predict (*self*, *X*)

Predict multi-output variable using a model trained for each target variable.

Parameters

X [(sparse) array-like, shape (n_samples, n_features)] Data.

Returns

y [(sparse) array-like, shape (n_samples, n_outputs)] Multi-output targets predicted across multiple predictors. Note: Separate models are generated for each predictor.

predict_proba(*self*, *X*)

Probability estimates. Returns prediction probabilities for each class of each output.

This method will raise a `ValueError` if any of the estimators do not have `predict_proba`.

Parameters

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

Returns

p [array of shape = [n_samples, n_classes], or a list of n_outputs such arrays if n_outputs > 1.] The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

score(*self*, *X*, *y*)

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

Parameters

X [array-like, shape [n_samples, n_features]] Test samples

y [array-like, shape [n_samples, n_outputs]] True values for X

Returns

scores [float] accuracy_score of self.predict(X) versus y

set_params(*self*, ***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.

Returns

self

6.28.4 sklearn.multioutput.RegressorChain

class sklearn.multioutput.**RegressorChain**(*base_estimator*, *order=None*, *cv=None*, *random_state=None*)

A multi-label model that arranges regressions into a chain.

Each model makes a prediction in the order specified by the chain using all of the available features provided to the model plus the predictions of models that are earlier in the chain.

Read more in the *User Guide*.

Parameters

base_estimator [estimator] The base estimator from which the classifier chain is built.

order [array-like, shape=[n_outputs] or 'random', optional] By default the order will be determined by the order of columns in the label matrix Y.:

```
order = [0, 1, 2, ..., Y.shape[1] - 1]
```

The order of the chain can be explicitly set by providing a list of integers. For example, for a chain of length 5.:

```
order = [1, 3, 2, 4, 0]
```

means that the first model in the chain will make predictions for column 1 in the Y matrix, the second model will make predictions for column 3, etc.

If order is 'random' a random ordering will be used.

cv [int, cross-validation generator or an iterable, optional (default=None)] Determines whether to use cross validated predictions or true labels for the results of previous estimators in the chain. If cv is None the true labels are used when fitting. Otherwise possible inputs for cv are:

- integer, to specify the number of folds in a (Stratified)KFold,
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

The random number generator is used to generate random chain orders.

Attributes

estimators_ [list] A list of clones of base_estimator.

order_ [list] The order of labels in the classifier chain.

See also:

`ClassifierChain` Equivalent for classification

`MultioutputRegressor` Learns each output independently rather than chaining.

Methods

fit(self, X, Y)	Fit the model to data matrix X and targets Y.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict on the data matrix X using the ClassifierChain model.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *base_estimator*, *order=None*, *cv=None*, *random_state=None*)

fit (*self*, *X*, *Y*)
Fit the model to data matrix X and targets Y.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Y [array-like, shape (n_samples, n_classes)] The target values.

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Predict on the data matrix X using the ClassifierChain model.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Returns

Y_pred [array-like, shape (n_samples, n_classes)] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

6.29 sklearn.naive_bayes: Naive Bayes

The *sklearn.naive_bayes* module implements Naive Bayes algorithms. These are supervised learning methods based on applying Bayes' theorem with strong (naive) feature independence assumptions.

User guide: See the *Naive Bayes* section for further details.

naive_bayes.BernoulliNB([alpha, binarize, ...])	Naive Bayes classifier for multivariate Bernoulli models.
naive_bayes.GaussianNB([priors, var_smoothing])	Gaussian Naive Bayes (GaussianNB)
naive_bayes.MultinomialNB([alpha, ...])	Naive Bayes classifier for multinomial models
naive_bayes.ComplementNB([alpha, fit_prior, ...])	The Complement Naive Bayes classifier described in Rennie et al.

6.29.1 sklearn.naive_bayes.BernoulliNB

class sklearn.naive_bayes.**BernoulliNB**(*alpha=1.0*, *binarize=0.0*, *fit_prior=True*, *class_prior=None*)
Naive Bayes classifier for multivariate Bernoulli models.

Like MultinomialNB, this classifier is suitable for discrete data. The difference is that while MultinomialNB works with occurrence counts, BernoulliNB is designed for binary/boolean features.

Read more in the *User Guide*.

Parameters

alpha [float, optional (default=1.0)] Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

binarize [float or None, optional (default=0.0)] Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.

fit_prior [boolean, optional (default=True)] Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

class_prior [array-like, size=[n_classes,], optional (default=None)] Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

Attributes

class_log_prior_ [array, shape = [n_classes]] Log probability of each class (smoothed).

feature_log_prob_ [array, shape = [n_classes, n_features]] Empirical log probability of features given a class, P(x_i|y).

class_count_ [array, shape = [n_classes]] Number of samples encountered for each class during fitting. This value is weighted by the sample weight when provided.

feature_count_ [array, shape = [n_classes, n_features]] Number of samples encountered for each (class, feature) during fitting. This value is weighted by the sample weight when provided.

References

C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234-265. https://nlp.stanford.edu/IR-book/html/htmledition/the-bernoulli-model-1.html

A. McCallum and K. Nigam (1998). A comparison of event models for naive Bayes text classification. Proc. AAAI/ICML-98 Workshop on Learning for Text Categorization, pp. 41-48.

V. Metsis, I. Androutsopoulos and G. Paliouras (2006). Spam filtering with naive Bayes – Which naive Bayes? 3rd Conf. on Email and Anti-Spam (CEAS).

Examples

```
>>> import numpy as np
>>> X = np.random.randint(2, size=(6, 100))
>>> Y = np.array([1, 2, 3, 4, 4, 5])
>>> from sklearn.naive_bayes import BernoulliNB
>>> clf = BernoulliNB()
>>> clf.fit(X, Y)
BernoulliNB(alpha=1.0, binarize=0.0, class_prior=None, fit_prior=True)
>>> print(clf.predict(X[2:3]))
[3]
```

Methods

fit(self, X, y[, sample_weight])	Fit Naive Bayes classifier according to X, y
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return log-probability estimates for the test vector X.
predict_proba(self, X)	Return probability estimates for the test vector X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *binarize=0.0*, *fit_prior=True*, *class_prior=None*)

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit Naive Bayes classifier according to X, y

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target values.

sample_weight [array-like, shape = [n_samples], (default=None)] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance overhead hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target values.

classes [array-like, shape = [n_classes] (default=None)] List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

sample_weight [array-like, shape = [n_samples] (default=None)] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

predict(*self*, *X*)
Perform classification on an array of test vectors X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array, shape = [n_samples]] Predicted target values for X

predict_log_proba(*self*, *X*)
Return log-probability estimates for the test vector X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array-like, shape = [n_samples, n_classes]] Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

predict_proba (*self*, *X*)
Return probability estimates for the test vector X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array-like, shape = [n_samples, n_classes]] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.naive_bayes.BernoulliNB`

- *Hashing feature transformation using Totally Random Trees*
- *Classification of text documents using sparse features*

6.29.2 `sklearn.naive_bayes`.GaussianNB

class `sklearn.naive_bayes.`**GaussianNB** (*priors=None*, *var_smoothing=1e-09*)
Gaussian Naive Bayes (GaussianNB)

Can perform online updates to model parameters via `partial_fit` method. For details on algorithm used to update feature means and variance online, see Stanford CS tech report STAN-CS-79-773 by Chan, Golub, and LeVeque:

http://i.stanford.edu/pub/cstr/reports/cs/tr/79/773/CS-TR-79-773.pdf

Read more in the *User Guide*.

Parameters

priors [array-like, shape (n_classes,)] Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

var_smoothing [float, optional (default=1e-9)] Portion of the largest variance of all features that is added to variances for calculation stability.

Attributes

class_prior_ [array, shape (n_classes,)] probability of each class.

class_count_ [array, shape (n_classes,)] number of training samples observed in each class.

theta_ [array, shape (n_classes, n_features)] mean of each feature per class

sigma_ [array, shape (n_classes, n_features)] variance of each feature per class

epsilon_ [float] absolute additive value to variances

Examples

```
>>> 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])
>>> from sklearn.naive_bayes import GaussianNB
>>> clf = GaussianNB()
>>> clf.fit(X, Y)
GaussianNB(priors=None, var_smoothing=1e-09)
>>> print(clf.predict([[-0.8, -1]]))
[1]
>>> clf_pf = GaussianNB()
>>> clf_pf.partial_fit(X, Y, np.unique(Y))
GaussianNB(priors=None, var_smoothing=1e-09)
>>> print(clf_pf.predict([[-0.8, -1]]))
[1]
```

Methods

fit(self, X, y[, sample_weight])	Fit Gaussian Naive Bayes according to X, y
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return log-probability estimates for the test vector X.
predict_proba(self, X)	Return probability estimates for the test vector X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *priors=None*, *var_smoothing=1e-09*)

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit Gaussian Naive Bayes according to X, y

Parameters

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

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

sample_weight [array-like, shape (n_samples,), optional (default=None)] Weights applied to individual samples (1. for unweighted).

New in version 0.17: Gaussian Naive Bayes supports fitting with *sample_weight*.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance and numerical stability overhead, hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.

Parameters

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

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

classes [array-like, shape (n_classes,), optional (default=None)] List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

sample_weight [array-like, shape (n_samples,), optional (default=None)] Weights applied to individual samples (1. for unweighted).

New in version 0.17.

Returns

self [object]

predict(*self*, *X*)
Perform classification on an array of test vectors X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array, shape = [n_samples]] Predicted target values for X

predict_log_proba(*self*, *X*)
Return log-probability estimates for the test vector X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array-like, shape = [n_samples, n_classes]] Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

predict_proba(*self*, *X*)
Return probability estimates for the test vector X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array-like, shape = [n_samples, n_classes]] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.naive_bayes.GaussianNB`

- *Comparison of Calibration of Classifiers*
- *Probability Calibration curves*
- *Probability calibration of classifiers*
- *Classifier comparison*
- *Plot class probabilities calculated by the VotingClassifier*

- *Plotting Learning Curves*
- *Importance of Feature Scaling*

6.29.3 sklearn.naive_bayes.MultinomialNB

class sklearn.naive_bayes.**MultinomialNB**(*alpha=1.0*, *fit_prior=True*, *class_prior=None*)
Naive Bayes classifier for multinomial models

The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

Read more in the *User Guide*.

Parameters

alpha [float, optional (default=1.0)] Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

fit_prior [boolean, optional (default=True)] Whether to learn class prior probabilities or not. If false, a uniform prior will be used.

class_prior [array-like, size (n_classes,), optional (default=None)] Prior probabilities of the classes. If specified the priors are not adjusted according to the data.

Attributes

class_log_prior_ [array, shape (n_classes,)] Smoothed empirical log probability for each class.

intercept_ [array, shape (n_classes,)] Mirrors `class_log_prior_` for interpreting MultinomialNB as a linear model.

feature_log_prob_ [array, shape (n_classes, n_features)] Empirical log probability of features given a class, `P(x_i|y)`.

coef_ [array, shape (n_classes, n_features)] Mirrors `feature_log_prob_` for interpreting MultinomialNB as a linear model.

class_count_ [array, shape (n_classes,)] Number of samples encountered for each class during fitting. This value is weighted by the sample weight when provided.

feature_count_ [array, shape (n_classes, n_features)] Number of samples encountered for each (class, feature) during fitting. This value is weighted by the sample weight when provided.

Notes

For the rationale behind the names *coef_* and `intercept_`, i.e. naive Bayes as a linear classifier, see J. Rennie et al. (2003), Tackling the poor assumptions of naive Bayes text classifiers, ICML.

References

C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234-265. https://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html

Examples

```
>>> import numpy as np
>>> X = np.random.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB()
>>> clf.fit(X, y)
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
>>> print(clf.predict(X[2:3]))
[3]
```

Methods

fit(self, X, y[, sample_weight])	Fit Naive Bayes classifier according to X, y
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return log-probability estimates for the test vector X.
predict_proba(self, X)	Return probability estimates for the test vector X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *alpha=1.0*, *fit_prior=True*, *class_prior=None*)

fit (*self*, *X*, *y*, *sample_weight=None*)
Fit Naive Bayes classifier according to X, y

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target values.

sample_weight [array-like, shape = [n_samples], (default=None)] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit (*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance overhead hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target values.

classes [array-like, shape = [n_classes] (default=None)] List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

sample_weight [array-like, shape = [n_samples] (default=None)] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

predict(*self*, *X*)
Perform classification on an array of test vectors X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array, shape = [n_samples]] Predicted target values for X

predict_log_proba(*self*, *X*)
Return log-probability estimates for the test vector X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array-like, shape = [n_samples, n_classes]] Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

predict_proba(*self*, *X*)
Return probability estimates for the test vector X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array-like, shape = [n_samples, n_classes]] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.naive_bayes.MultinomialNB`

- *Out-of-core classification of text documents*
- *Classification of text documents using sparse features*

6.29.4 `sklearn.naive_bayes`.ComplementNB

class `sklearn.naive_bayes.`**ComplementNB**(*alpha=1.0*, *fit_prior=True*, *class_prior=None*, *norm=False*)
The Complement Naive Bayes classifier described in Rennie et al. (2003).

The Complement Naive Bayes classifier was designed to correct the "severe assumptions" made by the standard Multinomial Naive Bayes classifier. It is particularly suited for imbalanced data sets.

Read more in the *User Guide*.

Parameters

alpha [float, optional (default=1.0)] Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

fit_prior [boolean, optional (default=True)] Only used in edge case with a single class in the training set.

class_prior [array-like, size (n_classes,), optional (default=None)] Prior probabilities of the classes. Not used.

norm [boolean, optional (default=False)] Whether or not a second normalization of the weights is performed. The default behavior mirrors the implementations found in Mahout and Weka, which do not follow the full algorithm described in Table 9 of the paper.

Attributes

class_log_prior_ [array, shape (n_classes,)] Smoothed empirical log probability for each class. Only used in edge case with a single class in the training set.

feature_log_prob_ [array, shape (n_classes, n_features)] Empirical weights for class complements.

class_count_ [array, shape (n_classes,)] Number of samples encountered for each class during fitting. This value is weighted by the sample weight when provided.

feature_count_ [array, shape (n_classes, n_features)] Number of samples encountered for each (class, feature) during fitting. This value is weighted by the sample weight when provided.

feature_all_ [array, shape (n_features,)] Number of samples encountered for each feature during fitting. This value is weighted by the sample weight when provided.

References

Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). Tackling the poor assumptions of naive bayes text classifiers. In ICML (Vol. 3, pp. 616-623). https://people.csail.mit.edu/jrennie/papers/icml03-nb.pdf

Examples

```
>>> import numpy as np
>>> X = np.random.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import ComplementNB
>>> clf = ComplementNB()
>>> clf.fit(X, y)
ComplementNB(alpha=1.0, class_prior=None, fit_prior=True, norm=False)
>>> print(clf.predict(X[2:3]))
[3]
```

Methods

fit(self, X, y[, sample_weight])	Fit Naive Bayes classifier according to X, y
get_params(self[, deep])	Get parameters for this estimator.
partial_fit(self, X, y[, classes, sample_weight])	Incremental fit on a batch of samples.
predict(self, X)	Perform classification on an array of test vectors X.
predict_log_proba(self, X)	Return log-probability estimates for the test vector X.
predict_proba(self, X)	Return probability estimates for the test vector X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *alpha=1.0*, *fit_prior=True*, *class_prior=None*, *norm=False*)

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit Naive Bayes classifier according to X, y

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target values.

sample_weight [array-like, shape = [n_samples], (default=None)] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit(*self*, *X*, *y*, *classes=None*, *sample_weight=None*)
Incremental fit on a batch of samples.

This method is expected to be called several times consecutively on different chunks of a dataset so as to implement out-of-core or online learning.

This is especially useful when the whole dataset is too big to fit in memory at once.

This method has some performance overhead hence it is better to call partial_fit on chunks of data that are as large as possible (as long as fitting in the memory budget) to hide the overhead.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target values.

classes [array-like, shape = [n_classes] (default=None)] List of all the classes that can possibly appear in the y vector.

Must be provided at the first call to partial_fit, can be omitted in subsequent calls.

sample_weight [array-like, shape = [n_samples] (default=None)] Weights applied to individual samples (1. for unweighted).

Returns

self [object]

predict(*self*, *X*)
Perform classification on an array of test vectors X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array, shape = [n_samples]] Predicted target values for X

predict_log_proba(*self*, *X*)
Return log-probability estimates for the test vector X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array-like, shape = [n_samples, n_classes]] Returns the log-probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

predict_proba (*self*, *X*)
Return probability estimates for the test vector X.

Parameters

X [array-like, shape = [n_samples, n_features]]

Returns

C [array-like, shape = [n_samples, n_classes]] Returns the probability of the samples for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.naive_bayes.ComplementNB`

- *Classification of text documents using sparse features*

6.30 `sklearn.neighbors`: Nearest Neighbors

The *`sklearn.neighbors`* module implements the k-nearest neighbors algorithm.

User guide: See the *Nearest Neighbors* section for further details.

`neighbors.BallTree`	BallTree for fast generalized N-point problems
`neighbors.DistanceMetric`	DistanceMetric class
`neighbors.KDTree`	KDTree for fast generalized N-point problems

Continued on next page

Table 6.223 – continued from previous page

neighbors.KernelDensity([bandwidth, ...])	Kernel Density Estimation
neighbors.KNeighborsClassifier([...])	Classifier implementing the k-nearest neighbors vote.
neighbors.KNeighborsRegressor([n_neighbors, ...])	Regression based on k-nearest neighbors.
neighbors.LocalOutlierFactor([n_neighbors, ...])	Unsupervised Outlier Detection using Local Outlier Factor (LOF)
neighbors.RadiusNeighborsClassifier([...])	Classifier implementing a vote among neighbors within a given radius
neighbors.RadiusNeighborsRegressor([radius, ...])	Regression based on neighbors within a fixed radius.
neighbors.NearestCentroid([metric, ...])	Nearest centroid classifier.
neighbors.NearestNeighbors([n_neighbors, ...])	Unsupervised learner for implementing neighbor searches.
neighbors.NeighborhoodComponentsAnalysis([...])	Neighborhood Components Analysis

6.30.1 sklearn.neighbors.BallTree

class sklearn.neighbors.**BallTree**

BallTree for fast generalized N-point problems

BallTree(X, leaf_size=40, metric='minkowski', **kwargs)

Parameters

X [array-like, 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 integer (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 [string 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.

Attributes

data [memory view] The training data

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

Methods

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()	query_radius(self, X, r, count_only = False):
two_point_correlation()	Compute the two-point correlation function

get_arrays	
get_n_calls	
get_tree_stats	
reset_n_calls	

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

kernel_density (*self*, *X*, *h*, *kernel='gaussian'*, *atol=0*, *rtol=1E-8*, *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, 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 [string] specify the kernel to use. Options are - 'gaussian' - 'tophat' - 'epanechnikov' - 'exponential' - 'linear' - 'cosine' Default is kernel = 'gaussian'

atol, rtol [float (default = 0)] 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 [boolean (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 [boolean (default = False)] return the logarithm of the result. This can be more accurate than returning the result itself for narrow kernels.

Returns

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

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

Parameters

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

k [integer (default = 1)] The number of nearest neighbors to return

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

dualtree [boolean (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 [boolean (default = False)] if True, then query the nodes in a breadth-first manner. Otherwise, query the nodes in a depth-first manner.

sort_results [boolean (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 [array of doubles - shape: x.shape[:-1] + (k,)] each entry gives the list of distances to the neighbors of the corresponding point

i [array of integers - shape: x.shape[:-1] + (k,)] each entry gives the list of indices of neighbors of the corresponding point

query_radius()

query_radius(self, X, r, count_only = False):

query the tree for neighbors within a radius r

Parameters

X [array-like, 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 [boolean (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 [boolean (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 [boolean (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 [array of integers, shape = X.shape[:-1]] each entry gives the number of neighbors within a distance r of the corresponding point.

ind [array of objects, shape = X.shape[:-1]] 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 [array of objects, shape = X.shape[:-1]] each element is a numpy double array listing the distances corresponding to indices in i.

two_point_correlation()
Compute the two-point correlation function

Parameters

X [array-like, 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 [boolean (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]

6.30.2 sklearn.neighbors.DistanceMetric

class sklearn.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). For example, to use the Euclidean distance:

```
>>> 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.

Methods

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.

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

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.

6.30.3 sklearn.neighbors.KDTree

class sklearn.neighbors.**KDTree**
KDTree for fast generalized N-point problems

KDTree(X, leaf_size=40, metric='minkowski', **kwargs)

Parameters

X [array-like, 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 integer (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 [string 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.

Attributes

data [memory view] The training data

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])
```

Methods

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()	query_radius(self, X, r, count_only = False):
two_point_correlation()	Compute the two-point correlation function

get_arrays	
get_n_calls	
get_tree_stats	
reset_n_calls	

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

kernel_density (*self*, *X*, *h*, *kernel='gaussian'*, *atol=0*, *rtol=1E-8*, *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, 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 [string] specify the kernel to use. Options are - 'gaussian' - 'tophat' - 'epanechnikov' - 'exponential' - 'linear' - 'cosine' Default is kernel = 'gaussian'

atol, rtol [float (default = 0)] 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 [boolean (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 [boolean (default = False)] return the logarithm of the result. This can be more accurate than returning the result itself for narrow kernels.

Returns

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

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

Parameters

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

k [integer (default = 1)] The number of nearest neighbors to return

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

dualtree [boolean (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 [boolean (default = False)] if True, then query the nodes in a breadth-first manner. Otherwise, query the nodes in a depth-first manner.

sort_results [boolean (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 [array of doubles - shape: x.shape[:-1] + (k,)] each entry gives the list of distances to the neighbors of the corresponding point

i [array of integers - shape: x.shape[:-1] + (k,)] each entry gives the list of indices of neighbors of the corresponding point

query_radius()

query_radius(self, X, r, count_only = False):

query the tree for neighbors within a radius r

Parameters

X [array-like, 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 [boolean (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 [boolean (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 [boolean (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 [array of integers, shape = X.shape[:-1]] each entry gives the number of neighbors within a distance r of the corresponding point.

ind [array of objects, shape = X.shape[:-1]] 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 [array of objects, shape = X.shape[:-1]] each element is a numpy double array listing the distances corresponding to indices in i.

two_point_correlation()
Compute the two-point correlation function

Parameters

X [array-like, 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 [boolean (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]

6.30.4 sklearn.neighbors.KernelDensity

class sklearn.neighbors.**KernelDensity**(*bandwidth=1.0*, *algorithm='auto'*, *kernel='gaussian'*, *metric='euclidean'*, *atol=0*, *rtol=0*, *breadth_first=True*, *leaf_size=40*, *metric_params=None*)

Kernel Density Estimation

Read more in the *User Guide*.

Parameters

bandwidth [float] The bandwidth of the kernel.

algorithm [string] The tree algorithm to use. Valid options are ['kd_tree'|'ball_tree'|'auto']. Default is 'auto'.

kernel [string] The kernel to use. Valid kernels are ['gaussian'|'tophat'|'epanechnikov'|'exponential'|'linear'|'cosine'] Default is 'gaussian'.

metric [string] 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 [boolean] 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*.

Methods

fit(self, X[, y, sample_weight])	Fit the Kernel Density model on the data.
get_params(self[, deep])	Get parameters for this estimator.
sample(self[, n_samples, random_state])	Generate random samples from the model.
score(self, X[, y])	Compute the total log probability density under the model.
score_samples(self, X)	Evaluate the density model on the data.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *bandwidth=1.0*, *algorithm='auto'*, *kernel='gaussian'*, *metric='euclidean'*, *atol=0*, *rtol=0*, *breadth_first=True*, *leaf_size=40*, *metric_params=None*)

fit(*self*, *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.

sample_weight [array_like, shape (n_samples,), optional] List of sample weights attached to the data X.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *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 or None. default to None] 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`.

Returns

X [array_like, shape (n_samples, n_features)] List of samples.

score(*self*, *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.

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(*self*, *X*)
Evaluate the 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(*self*, ***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.

Returns

self

Examples using `sklearn.neighbors.KernelDensity`

- *Kernel Density Estimation*
- *Kernel Density Estimate of Species Distributions*
- *Simple 1D Kernel Density Estimation*

6.30.5 `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*, ***kwargs*)
Classifier implementing the k-nearest neighbors vote.

Read more in the *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', 'ball_tree', 'kd_tree', 'brute'}, optional] 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, 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 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]]))
[[0.66666667 0.33333333]]
```

Methods

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

__init__(*self*, *n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

fit(*self*, *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 of shape = [n_samples] or [n_samples, n_outputs]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X=None*, *n_neighbors=None*, *return_distance=True*)

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 sklearn.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_graph(*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)

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

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 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 matrix in CSR format, shape = [n_samples, 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(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

predict (*self*, *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 (*self*, *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 (*self*, *X*, *y*, *sample_weight=None*)

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.neighbors.KNeighborsClassifier`

- *Classifier comparison*
- *Plot the decision boundaries of a VotingClassifier*
- *Digits Classification Exercise*
- *Nearest Neighbors Classification*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Classification of text documents using sparse features*

6.30.6 `sklearn.neighbors`.KNeighborsRegressor

class `sklearn.neighbors.`**`KNeighborsRegressor`**(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *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 *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', 'ball_tree', 'kd_tree', 'brute'}, optional] 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, 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:

NearestNeighbors

RadiusNeighborsRegressor

KNeighborsClassifier

RadiusNeighborsClassifier

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 KNeighborsRegressor
>>> neigh = KNeighborsRegressor(n_neighbors=2)
>>> neigh.fit(X, y)
KNeighborsRegressor(...)
>>> print(neigh.predict([[1.5]]))
[0.5]
```

Methods

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

__init__(*self*, *n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

fit(*self*, *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(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X=None*, *n_neighbors=None*, *return_distance=True*)

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 sklearn.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_graph (*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)

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

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 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 matrix in CSR format, shape = [n_samples, 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(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
```

```
       [0., 1., 1.],
       [1., 0., 1.]])
```

predict (*self*, *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 (*self*, *X*, *y*, *sample_weight=None*)

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.neighbors.KNeighborsRegressor`

- *Face completion with a multi-output estimators*

- *Imputing missing values with variants of IterativeImputer*
- *Nearest Neighbors regression*

6.30.7 sklearn.neighbors.LocalOutlierFactor

class sklearn.neighbors.**LocalOutlierFactor**(*n_neighbors=20*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *contamination='legacy'*, *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', 'ball_tree', 'kd_tree', 'brute'}, optional] 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, 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 [float in (0., 0.5), optional (default=0.1)] 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 decision function. If "auto", the decision function threshold is determined as in the original paper.

Changed in version 0.20: The default value of `contamination` will change from 0.1 in 0.20 to `'auto'` in 0.22.

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.

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.

References

[Rca479bb49841-1]

Methods

`fit`(self, X[, y])	Fit the model using X as training data.
`get_params`(self[, deep])	Get parameters for this estimator.

Continued on next page

Table 6.230 – continued from previous page

kneighbors(self[, X, n_neighbors, ...])	Finds the K-neighbors of a point.
kneighbors_graph(self[, X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *n_neighbors=20*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *contamination='legacy'*, *novelty=False*, *n_jobs=None*)

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 (*self*, *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'.

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

Returns

self [object]

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 (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X=None*, *n_neighbors=None*, *return_distance=True*)
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 sklearn.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_graph (*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)
Computes the (weighted) graph of k-Neighbors for points in X

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 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 matrix in CSR format, shape = [n_samples, 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(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

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.

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(*self*, ***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.

Returns

self

Examples using `sklearn.neighbors.LocalOutlierFactor`

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *Outlier detection with Local Outlier Factor (LOF)*
- *Novelty detection with Local Outlier Factor (LOF)*

6.30.8 `sklearn.neighbors`.RadiusNeighborsClassifier

class `sklearn.neighbors.`**RadiusNeighborsClassifier**(*radius=1.0*, *weights='uniform'*, *algorithm='auto'*, *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 *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', 'ball_tree', 'kd_tree', 'brute'}, optional] 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, 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 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 sklearn.neighbors import RadiusNeighborsClassifier
>>> neigh = RadiusNeighborsClassifier(radius=1.0)
>>> neigh.fit(X, y)
RadiusNeighborsClassifier(...)
>>> print(neigh.predict([[1.5]]))
[0]
```

Methods

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

__init__ (*self*, *radius=1.0*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *outlier_label=None*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

fit (*self*, *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 of shape = [n_samples] or [n_samples, n_outputs]

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *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 (*self*, *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 sklearn.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 (*self*, *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 (*self*, *X*, *y*, *sample_weight=None*)

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

6.30.9 sklearn.neighbors.RadiusNeighborsRegressor

class sklearn.neighbors.**RadiusNeighborsRegressor**(*radius=1.0*, *weights='uniform'*, *algorithm='auto'*, *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 *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', 'ball_tree', 'kd_tree', 'brute'}, optional] 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, 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.

See also:

NearestNeighbors

KNeighborsRegressor

KNeighborsClassifier

RadiusNeighborsClassifier

Notes

See *Nearest Neighbors* in the 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 sklearn.neighbors import RadiusNeighborsRegressor
>>> neigh = RadiusNeighborsRegressor(radius=1.0)
>>> neigh.fit(X, y)
RadiusNeighborsRegressor(...)
>>> print(neigh.predict([[1.5]]))
[0.5]
```

Methods

fit(self, X, y)	Fit the model using X as training data and y as target values
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict the target for the provided data
radius_neighbors(self[, X, radius, ...])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph(self[, X, radius, mode])	Computes the (weighted) graph of Neighbors for points in X
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *radius=1.0*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

fit(*self*, *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(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *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(*self*, *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 sklearn.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 (*self*, *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(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

6.30.10 sklearn.neighbors.NearestCentroid

class sklearn.neighbors.**NearestCentroid**(*metric='euclidean'*, *shrink_threshold=None*)
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 *User Guide*.

Parameters

metric [string, 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.

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

Attributes

centroids_ [array-like, shape = [n_classes, n_features]] Centroid of each class

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.nearest_centroid 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(metric='euclidean', shrink_threshold=None)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

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

__init__ (*self*, *metric='euclidean'*, *shrink_threshold=None*)

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

Parameters

X [{array-like, sparse matrix}, 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, shape = [n_samples]] Target values (integers)

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *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, shape = [n_samples, n_features]]

Returns

C [array, 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 (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.neighbors.NearestCentroid`

- *Nearest Centroid Classification*
- *Classification of text documents using sparse features*

6.30.11 `sklearn.neighbors`.NearestNeighbors

class `sklearn.neighbors.`**`NearestNeighbors`**(*n_neighbors=5*, *radius=1.0*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

Unsupervised learner for implementing neighbor searches.

Read more in the *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', 'ball_tree', 'kd_tree', 'brute'}, optional] 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, 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 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 sklearn.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)
```

Methods

fit(self, X[, y])	Fit the model using X as training data

Continued on next page

Table 6.234 – continued from previous page

get_params(self[, deep])	Get parameters for this estimator.
kneighbors(self[, X, n_neighbors, ...])	Finds the K-neighbors of a point.
kneighbors_graph(self[, X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbors for points in X
radius_neighbors(self[, X, radius, ...])	Finds the neighbors within a given radius of a point or points.
radius_neighbors_graph(self[, X, radius, mode])	Computes the (weighted) graph of Neighbors for points in X
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *n_neighbors=5*, *radius=1.0*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=None*, ***kwargs*)

fit(*self*, *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(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X=None*, *n_neighbors=None*, *return_distance=True*)

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 sklearn.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_graph (*self*, *X=None*, *n_neighbors=None*, *mode='connectivity'*)

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

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 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 matrix in CSR format, shape = [n_samples, 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(algorithm='auto', leaf_size=30, ...)
>>> A = neigh.kneighbors_graph(X)
>>> A.toarray()
array([[1., 0., 1.],
       [0., 1., 1.],
       [1., 0., 1.]])
```

radius_neighbors (*self*, *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 sklearn.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 (*self*, *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(*self*, ***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.

Returns

self

6.30.12 sklearn.neighbors.NeighborhoodComponentsAnalysis

class sklearn.neighbors.**NeighborhoodComponentsAnalysis**(*n_components=None*, *init='auto'*, *warm_start=False*, *max_iter=50*, *tol=1e-05*, *callback=None*, *verbose=0*, *random_state=None*)

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 *User Guide*.

Parameters

n_components [int, optional (default=None)] Preferred dimensionality of the projected space. If None it will be set to `n_features`.

init [string or numpy array, optional (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 *decomposition.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 *discriminant_analysis.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, optional, (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, optional (default=50)] Maximum number of iterations in the optimization.

tol [float, optional (default=1e-5)] Convergence tolerance for the optimization.

callback [callable, optional (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, optional (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 or None, optional (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.

Attributes

components_ [array, shape (n_components, n_features)] The linear transformation learned during fitting.

n_iter_ [int] Counts the number of iterations performed by the optimizer.

References

[Rf9b6baee8229-1], *[Rf9b6baee8229-2]*

Examples

```
>>> from sklearn.neighbors.nca 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...
```

Methods

fit(self, X, y)	Fit the model according to the given training data.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Applies the learned transformation to the given data.

__init__(*self*, *n_components=None*, *init='auto'*, *warm_start=False*, *max_iter=50*, *tol=1e-05*, *callback=None*, *verbose=0*, *random_state=None*)

fit(*self*, *X*, *y*)

Fit the model according to the given training data.

Parameters

X [array-like, shape (n_samples, n_features)] The training samples.

y [array-like, shape (n_samples,)] The corresponding training labels.

Returns

self [object] returns a trained NeighborhoodComponentsAnalysis model.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Applies the learned transformation to the given data.

Parameters

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

Returns

X_embedded: array, shape (n_samples, n_components) The data samples transformed.

Raises

NotFittedError If *fit* has not been called before.

Examples using `sklearn.neighbors.NeighborhoodComponentsAnalysis`

- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Neighborhood Components Analysis Illustration*

neighbors.kneighbors_graph(X, n_neighbors[, ...])	Computes the (weighted) graph of k-Neighbors for points in X
neighbors.radius_neighbors_graph(X, radius)	Computes the (weighted) graph of Neighbors for points in X

6.30.13 sklearn.neighbors.kneighbors_graph

sklearn.neighbors.**kneighbors_graph**(*X*, *n_neighbors*, *mode='connectivity'*, *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 *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.

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 sklearn.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.]])
```

Examples using `sklearn.neighbors.kneighbors_graph`

- *Agglomerative clustering with and without structure*
- *Hierarchical clustering: structured vs unstructured ward*
- *Comparing different clustering algorithms on toy datasets*

6.30.14 `sklearn.neighbors`.radius_neighbors_graph

`sklearn.neighbors.`**`radius_neighbors_graph`**(*X*, *radius*, *mode='connectivity'*, *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 *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.

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 sklearn.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.]])
```

6.31 sklearn.neural_network: Neural network models

The *sklearn.neural_network* module includes models based on neural networks.

User guide: See the *Neural network models (supervised)* and *Neural network models (unsupervised)* sections for further details.

neural_network.BernoulliRBM([n_components, ...])	Bernoulli Restricted Boltzmann Machine (RBM).
neural_network.MLPClassifier([...])	Multi-layer Perceptron classifier.
neural_network.MLPRegressor([...])	Multi-layer Perceptron regressor.

6.31.1 sklearn.neural_network.BernoulliRBM

class sklearn.neural_network.**BernoulliRBM**(*n_components=256*, *learning_rate=0.1*, *batch_size=10*, *n_iter=10*, *verbose=0*, *random_state=None*)

Bernoulli Restricted Boltzmann Machine (RBM).

A Restricted Boltzmann Machine with binary visible units and binary hidden units. Parameters are estimated using Stochastic Maximum Likelihood (SML), also known as Persistent Contrastive Divergence (PCD) [2].

The time complexity of this implementation is `O(d ** 2)` assuming d ~ n_features ~ n_components.

Read more in the *User Guide*.

Parameters

n_components [int, optional] Number of binary hidden units.

learning_rate [float, optional] The learning rate for weight updates. It is *highly* recommended to tune this hyper-parameter. Reasonable values are in the 10**[0., -3.] range.

batch_size [int, optional] Number of examples per minibatch.

n_iter [int, optional] Number of iterations/sweeps over the training dataset to perform during training.

verbose [int, optional] The verbosity level. The default, zero, means silent mode.

random_state [integer or RandomState, optional] A random number generator instance to define the state of the random permutations generator. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

Attributes

intercept_hidden_ [array-like, shape (n_components,)] Biases of the hidden units.

intercept_visible_ [array-like, shape (n_features,)] Biases of the visible units.

components_ [array-like, shape (n_components, n_features)] Weight matrix, where n_features in the number of visible units and n_components is the number of hidden units.

References

[1] Hinton, G. E., Osindero, S. and Teh, Y. A fast learning algorithm for deep belief nets. Neural Computation 18, pp 1527-1554. https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf

[2] Tieleman, T. Training Restricted Boltzmann Machines using Approximations to the Likelihood Gradient. International Conference on Machine Learning (ICML) 2008

Examples

```
>>> import numpy as np
>>> from sklearn.neural_network import BernoulliRBM
>>> X = np.array([[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]])
>>> model = BernoulliRBM(n_components=2)
>>> model.fit(X)
BernoulliRBM(batch_size=10, learning_rate=0.1, n_components=2, n_iter=10,
       random_state=None, verbose=0)
```

Methods

fit(self, X[, y])	Fit the model to the data X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
gibbs(self, v)	Perform one Gibbs sampling step.
partial_fit(self, X[, y])	Fit the model to the data X which should contain a partial segment of the data.
score_samples(self, X)	Compute the pseudo-likelihood of X.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Compute the hidden layer activation probabilities, P(h=1|v=X).

__init__(*self*, *n_components=256*, *learning_rate=0.1*, *batch_size=10*, *n_iter=10*, *verbose=0*, *random_state=None*)

fit(*self*, *X*, *y=None*)
Fit the model to the data X.

Parameters

X [{array-like, sparse matrix} shape (n_samples, n_features)] Training data.

Returns

self [BernoulliRBM] The fitted model.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

gibbs(*self*, *v*)
Perform one Gibbs sampling step.

Parameters

v [array-like, shape (n_samples, n_features)] Values of the visible layer to start from.

Returns

v_new [array-like, shape (n_samples, n_features)] Values of the visible layer after one Gibbs step.

partial_fit(*self*, *X*, *y=None*)
Fit the model to the data X which should contain a partial segment of the data.

Parameters

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

Returns

self [BernoulliRBM] The fitted model.

score_samples(*self*, *X*)
Compute the pseudo-likelihood of X.

Parameters

X [{array-like, sparse matrix} shape (n_samples, n_features)] Values of the visible layer. Must be all-boolean (not checked).

Returns

pseudo_likelihood [array-like, shape (n_samples,)] Value of the pseudo-likelihood (proxy for likelihood).

Notes

This method is not deterministic: it computes a quantity called the free energy on X, then on a randomly corrupted version of X, and returns the log of the logistic function of the difference.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Compute the hidden layer activation probabilities, P(h=1|v=X).

Parameters

X [{array-like, sparse matrix} shape (n_samples, n_features)] The data to be transformed.

Returns

h [array, shape (n_samples, n_components)] Latent representations of the data.

Examples using `sklearn.neural_network.BernoulliRBM`

- *Restricted Boltzmann Machine features for digit classification*

6.31.2 sklearn.neural_network.MLPClassifier

class sklearn.neural_network.**MLPClassifier**(*hidden_layer_sizes=(100,)*, *activation='relu'*, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*)
Multi-layer Perceptron classifier.

This model optimizes the log-loss function using LBFGS or stochastic gradient descent.

New in version 0.18.

Parameters

hidden_layer_sizes [tuple, length = n_layers - 2, default (100,)] The ith element represents the number of neurons in the ith hidden layer.

activation [{'identity', 'logistic', 'tanh', 'relu'}, default 'relu'] Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

solver [{'lbfgs', 'sgd', 'adam'}, default 'adam'] The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

alpha [float, optional, default 0.0001] L2 penalty (regularization term) parameter.

batch_size [int, optional, default 'auto'] Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the classifier will not use minibatch. When set to "auto", `batch_size=min(200, n_samples)`

learning_rate [{'constant', 'invscaling', 'adaptive'}, default 'constant'] Learning rate schedule for weight updates.

- 'constant' is a constant learning rate given by 'learning_rate_init'.
- 'invscaling' gradually decreases the learning rate at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)
- 'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.

Only used when `solver='sgd'`.

learning_rate_init [double, optional, default 0.001] The initial learning rate used. It controls the step-size in updating the weights. Only used when solver='sgd' or 'adam'.

power_t [double, optional, default 0.5] The exponent for inverse scaling learning rate. It is used in updating effective learning rate when the learning_rate is set to 'invscaling'. Only used when solver='sgd'.

max_iter [int, optional, default 200] Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. For stochastic solvers ('sgd', 'adam'), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps.

shuffle [bool, optional, default True] Whether to shuffle samples in each iteration. Only used when solver='sgd' or 'adam'.

random_state [int, RandomState instance or None, optional, default None] 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`.

tol [float, optional, default 1e-4] Tolerance for the optimization. When the loss or score is not improving by at least `tol` for `n_iter_no_change` consecutive iterations, unless `learning_rate` is set to 'adaptive', convergence is considered to be reached and training stops.

verbose [bool, optional, default False] Whether to print progress messages to stdout.

warm_start [bool, optional, default False] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

momentum [float, default 0.9] Momentum for gradient descent update. Should be between 0 and 1. Only used when solver='sgd'.

nesterovs_momentum [boolean, default True] Whether to use Nesterov's momentum. Only used when solver='sgd' and momentum > 0.

early_stopping [bool, default False] Whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as validation and terminate training when validation score is not improving by at least tol for `n_iter_no_change` consecutive epochs. The split is stratified, except in a multilabel setting. Only effective when solver='sgd' or 'adam'

validation_fraction [float, optional, default 0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True

beta_1 [float, optional, default 0.9] Exponential decay rate for estimates of first moment vector in adam, should be in [0, 1). Only used when solver='adam'

beta_2 [float, optional, default 0.999] Exponential decay rate for estimates of second moment vector in adam, should be in [0, 1). Only used when solver='adam'

epsilon [float, optional, default 1e-8] Value for numerical stability in adam. Only used when solver='adam'

n_iter_no_change [int, optional, default 10] Maximum number of epochs to not meet `tol` improvement. Only effective when solver='sgd' or 'adam'

New in version 0.20.

Attributes

classes_ [array or list of array of shape (n_classes,)] Class labels for each output.

loss_ [float] The current loss computed with the loss function.

coefs_ [list, length n_layers - 1] The ith element in the list represents the weight matrix corresponding to layer i.

intercepts_ [list, length n_layers - 1] The ith element in the list represents the bias vector corresponding to layer i + 1.

n_iter_ [int,] The number of iterations the solver has ran.

n_layers_ [int] Number of layers.

n_outputs_ [int] Number of outputs.

out_activation_ [string] Name of the output activation function.

Notes

MLPClassifier trains iteratively since at each time step the partial derivatives of the loss function with respect to the model parameters are computed to update the parameters.

It can also have a regularization term added to the loss function that shrinks model parameters to prevent overfitting.

This implementation works with data represented as dense numpy arrays or sparse scipy arrays of floating point values.

References

Hinton, Geoffrey E. "Connectionist learning procedures." Artificial intelligence 40.1 (1989): 185-234.

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." International Conference on Artificial Intelligence and Statistics. 2010.

He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." arXiv preprint arXiv:1502.01852 (2015).

Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

Methods

fit(self, X, y)	Fit the model to data matrix X and target(s) y.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the multi-layer perceptron classifier
predict_log_proba(self, X)	Return the log of probability estimates.
predict_proba(self, X)	Probability estimates.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *hidden_layer_sizes=(100,)*, *activation='relu'*, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*)

fit(*self*, *X*, *y*)

Fit the model to data matrix X and target(s) y.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The input data.

y [array-like, shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels in classification, real numbers in regression).

Returns

self [returns a trained MLP model.]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit

Update the model with a single iteration over the given data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

y [array-like, shape (n_samples,)] The target values.

classes [array, shape (n_classes), default None] Classes across all calls to partial_fit. Can be obtained via `np.unique(y_all)`, where y_all is the target vector of the entire dataset. This argument is required for the first call to partial_fit and can be omitted in the subsequent calls. Note that y doesn't need to contain all labels in `classes`.

Returns

self [returns a trained MLP model.]

predict (*self*, *X*)

Predict using the multi-layer perceptron classifier

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Returns

y [array-like, shape (n_samples,) or (n_samples, n_classes)] The predicted classes.

predict_log_proba (*self*, *X*)

Return the log of probability estimates.

Parameters

X [array-like, shape (n_samples, n_features)] The input data.

Returns

log_y_prob [array-like, shape (n_samples, n_classes)] The predicted log-probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`. Equivalent to log(predict_proba(X))

predict_proba (*self*, *X*)

Probability estimates.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Returns

y_prob [array-like, shape (n_samples, n_classes)] The predicted probability of the sample for each class in the model, where classes are ordered as they are in `self.classes_`.

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

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.neural_network.MLPClassifier`

- *Classifier comparison*
- *Visualization of MLP weights on MNIST*
- *Varying regularization in Multi-layer Perceptron*
- *Compare Stochastic learning strategies for MLPClassifier*

6.31.3 `sklearn.neural_network`.MLPRegressor

class `sklearn.neural_network.`**MLPRegressor**(*hidden_layer_sizes=(100,)*, *activation='relu'*, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*)

Multi-layer Perceptron regressor.

This model optimizes the squared-loss using LBFGS or stochastic gradient descent.

New in version 0.18.

Parameters

hidden_layer_sizes [tuple, length = n_layers - 2, default (100,)] The ith element represents the number of neurons in the ith hidden layer.

activation [{'identity', 'logistic', 'tanh', 'relu'}, default 'relu'] Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

solver [{'lbfgs', 'sgd', 'adam'}, default 'adam'] The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.

- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

alpha [float, optional, default 0.0001] L2 penalty (regularization term) parameter.

batch_size [int, optional, default 'auto'] Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the classifier will not use minibatch. When set to "auto", `batch_size=min(200, n_samples)`

learning_rate [{'constant', 'invscaling', 'adaptive'}, default 'constant'] Learning rate schedule for weight updates.

- 'constant' is a constant learning rate given by 'learning_rate_init'.
- 'invscaling' gradually decreases the learning rate `learning_rate_` at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)
- 'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.

Only used when solver='sgd'.

learning_rate_init [double, optional, default 0.001] The initial learning rate used. It controls the step-size in updating the weights. Only used when solver='sgd' or 'adam'.

power_t [double, optional, default 0.5] The exponent for inverse scaling learning rate. It is used in updating effective learning rate when the learning_rate is set to 'invscaling'. Only used when solver='sgd'.

max_iter [int, optional, default 200] Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. For stochastic solvers ('sgd', 'adam'), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps.

shuffle [bool, optional, default True] Whether to shuffle samples in each iteration. Only used when solver='sgd' or 'adam'.

random_state [int, RandomState instance or None, optional, default None] 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`.

tol [float, optional, default 1e-4] Tolerance for the optimization. When the loss or score is not improving by at least `tol` for `n_iter_no_change` consecutive iterations, unless `learning_rate` is set to 'adaptive', convergence is considered to be reached and training stops.

verbose [bool, optional, default False] Whether to print progress messages to stdout.

warm_start [bool, optional, default False] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

momentum [float, default 0.9] Momentum for gradient descent update. Should be between 0 and 1. Only used when solver='sgd'.

nesterovs_momentum [boolean, default True] Whether to use Nesterov's momentum. Only used when solver='sgd' and momentum > 0.

early_stopping [bool, default False] Whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as validation and terminate training when validation score is not improving by at least `tol` for `n_iter_no_change` consecutive epochs. Only effective when solver='sgd' or 'adam'

validation_fraction [float, optional, default 0.1] The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True

beta_1 [float, optional, default 0.9] Exponential decay rate for estimates of first moment vector in adam, should be in [0, 1). Only used when solver='adam'

beta_2 [float, optional, default 0.999] Exponential decay rate for estimates of second moment vector in adam, should be in [0, 1). Only used when solver='adam'

epsilon [float, optional, default 1e-8] Value for numerical stability in adam. Only used when solver='adam'

n_iter_no_change [int, optional, default 10] Maximum number of epochs to not meet `tol` improvement. Only effective when solver='sgd' or 'adam'

New in version 0.20.

Attributes

loss_ [float] The current loss computed with the loss function.

coefs_ [list, length n_layers - 1] The ith element in the list represents the weight matrix corresponding to layer i.

intercepts_ [list, length n_layers - 1] The ith element in the list represents the bias vector corresponding to layer i + 1.

n_iter_ [int,] The number of iterations the solver has ran.

n_layers_ [int] Number of layers.

n_outputs_ [int] Number of outputs.

out_activation_ [string] Name of the output activation function.

Notes

MLPRegressor trains iteratively since at each time step the partial derivatives of the loss function with respect to the model parameters are computed to update the parameters.

It can also have a regularization term added to the loss function that shrinks model parameters to prevent overfitting.

This implementation works with data represented as dense and sparse numpy arrays of floating point values.

References

Hinton, Geoffrey E. "Connectionist learning procedures." Artificial intelligence 40.1 (1989): 185-234.

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." International Conference on Artificial Intelligence and Statistics. 2010.

He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." arXiv preprint arXiv:1502.01852 (2015).

Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

Methods

fit(self, X, y)	Fit the model to data matrix X and target(s) y.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the multi-layer perceptron model.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *hidden_layer_sizes=(100,)*, *activation='relu'*, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*)

fit(*self*, *X*, *y*)

Fit the model to data matrix X and target(s) y.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)] The input data.

y [array-like, shape (n_samples,) or (n_samples, n_outputs)] The target values (class labels in classification, real numbers in regression).

Returns

self [returns a trained MLP model.]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

partial_fit

Update the model with a single iteration over the given data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

y [array-like, shape (n_samples,)] The target values.

Returns

self [returns a trained MLP model.]

predict (*self*, *X*)
Predict using the multi-layer perceptron model.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] The input data.

Returns

y [array-like, shape (n_samples, n_outputs)] The predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.neural_network.MLPRegressor`

- *Partial Dependence Plots*

6.32 sklearn.pipeline: Pipeline

The *sklearn.pipeline* module implements utilities to build a composite estimator, as a chain of transforms and estimators.

pipeline.FeatureUnion(transformer_list[, ...])	Concatenates results of multiple transformer objects.
pipeline.Pipeline(steps[, memory, verbose])	Pipeline of transforms with a final estimator.

6.32.1 sklearn.pipeline.FeatureUnion

class sklearn.pipeline.**FeatureUnion**(*transformer_list*, *n_jobs=None*, *transformer_weights=None*, *verbose=False*)

Concatenates results of multiple transformer objects.

This estimator applies a list of transformer objects in parallel to the input data, then concatenates the results. This is useful to combine several feature extraction mechanisms into a single transformer.

Parameters of the transformers may be set using its name and the parameter name separated by a '__'. A transformer may be replaced entirely by setting the parameter with its name to another transformer, or removed by setting to 'drop' or `None`.

Read more in the *User Guide*.

Parameters

transformer_list [list of (string, transformer) tuples] List of transformer objects to be applied to the data. The first half of each tuple is the name of the transformer.

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

transformer_weights [dict, optional] Multiplicative weights for features per transformer. Keys are transformer names, values the weights.

verbose [boolean, optional(default=False)] If True, the time elapsed while fitting each transformer will be printed as it is completed.

See also:

sklearn.pipeline.make_union convenience function for simplified feature union construction.

Examples

```
>>> from sklearn.pipeline import FeatureUnion
>>> from sklearn.decomposition import PCA, TruncatedSVD
>>> union = FeatureUnion([("pca", PCA(n_components=1)),
...                       ("svd", TruncatedSVD(n_components=2))])
>>> X = [[0., 1., 3], [2., 2., 5]]
>>> union.fit_transform(X)
array([[ 1.5       ,  3.0...,  0.8...],
       [-1.5       ,  5.7..., -0.4...]])
```

Methods

fit(self, X[, y])	Fit all transformers using X.
fit_transform(self, X[, y])	Fit all transformers, transform the data and concatenate results.
get_feature_names(self)	Get feature names from all transformers.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **kwargs)	Set the parameters of this estimator.
transform(self, X)	Transform X separately by each transformer, concatenate results.

__init__ (*self*, *transformer_list*, *n_jobs=None*, *transformer_weights=None*, *verbose=False*)

fit (*self*, *X*, *y=None*)
Fit all transformers using X.

Parameters

X [iterable or array-like, depending on transformers] Input data, used to fit transformers.

y [array-like, shape (n_samples, ...), optional] Targets for supervised learning.

Returns

self [FeatureUnion] This estimator

fit_transform (*self*, *X*, *y=None*, ***fit_params*)
Fit all transformers, transform the data and concatenate results.

Parameters

X [iterable or array-like, depending on transformers] Input data to be transformed.

y [array-like, shape (n_samples, ...), optional] Targets for supervised learning.

Returns

X_t [array-like or sparse matrix, shape (n_samples, sum_n_components)] hstack of results of transformers. sum_n_components is the sum of n_components (output dimension) over transformers.

get_feature_names (*self*)
Get feature names from all transformers.

Returns

feature_names [list of strings] Names of the features produced by transform.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, ***kwargs*)
Set the parameters of this estimator.

Valid parameter keys can be listed with `get_params()`.

Returns

self

transform(*self*, *X*)
Transform X separately by each transformer, concatenate results.

Parameters

X [iterable or array-like, depending on transformers] Input data to be transformed.

Returns

X_t [array-like or sparse matrix, shape (n_samples, sum_n_components)] hstack of results of transformers. sum_n_components is the sum of n_components (output dimension) over transformers.

Examples using `sklearn.pipeline.FeatureUnion`

- *Concatenating multiple feature extraction methods*

6.32.2 sklearn.pipeline.Pipeline

class sklearn.pipeline.**Pipeline**(*steps*, *memory=None*, *verbose=False*)
Pipeline of transforms with a final estimator.

Sequentially apply a list of transforms and a final estimator. Intermediate steps of the pipeline must be 'transforms', that is, they must implement fit and transform methods. The final estimator only needs to implement fit. The transformers in the pipeline can be cached using `memory` argument.

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps using their names and the parameter name separated by a '__', as in the example below. A step's estimator may be replaced entirely by setting the parameter with its name to another estimator, or a transformer removed by setting it to 'passthrough' or `None`.

Read more in the *User Guide*.

Parameters

steps [list] List of (name, transform) tuples (implementing fit/transform) that are chained, in the order in which they are chained, with the last object an estimator.

memory [None, str or object with the joblib.Memory interface, optional] Used to cache the fitted transformers of the pipeline. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute `named_steps` or `steps` to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming.

verbose [boolean, optional] If True, the time elapsed while fitting each step will be printed as it is completed.

Attributes

named_steps [bunch object, a dictionary with attribute access] Read-only attribute to access any step parameter by user given name. Keys are step names and values are steps parameters.

See also:

sklearn.pipeline.make_pipeline convenience function for simplified pipeline construction.

Examples

```
>>> from sklearn import svm
>>> from sklearn.datasets import samples_generator
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import f_regression
>>> from sklearn.pipeline import Pipeline
>>> # generate some data to play with
>>> X, y = samples_generator.make_classification(
...     n_informative=5, n_redundant=0, random_state=42)
>>> # ANOVA SVM-C
>>> anova_filter = SelectKBest(f_regression, k=5)
>>> clf = svm.SVC(kernel='linear')
>>> anova_svm = Pipeline([('anova', anova_filter), ('svc', clf)])
>>> # You can set the parameters using the names issued
>>> # For instance, fit using a k of 10 in the SelectKBest
>>> # and a parameter 'C' of the svm
>>> anova_svm.set_params(anova__k=10, svc__C=.1).fit(X, y)
...
Pipeline(memory=None,
         steps=[('anova', SelectKBest(...)),
                ('svc', SVC(...))], verbose=False)
>>> prediction = anova_svm.predict(X)
>>> anova_svm.score(X, y)
0.83
>>> # getting the selected features chosen by anova_filter
>>> anova_svm['anova'].get_support()
...
array([False, False,  True,  True, False, False,  True,  True, False,
       True, False,  True,  True, False,  True, False,  True,  True,
       False, False])
>>> # Another way to get selected features chosen by anova_filter
>>> anova_svm.named_steps.anova.get_support()
...
array([False, False,  True,  True, False, False,  True,  True, False,
       True, False,  True,  True, False,  True, False,  True,  True,
       False, False])
>>> # Indexing can also be used to extract a sub-pipeline.
>>> sub_pipeline = anova_svm[:1]
>>> sub_pipeline
Pipeline(memory=None, steps=[('anova', ...)], verbose=False)
>>> coef = anova_svm[-1].coef_
>>> anova_svm['svc'] is anova_svm[-1]
True
>>> coef.shape
(1, 10)
>>> sub_pipeline.inverse_transform(coef).shape
(1, 20)
```

Methods

decision_function(self, X)	Apply transforms, and decision_function of the final estimator
fit(self, X[, y])	Fit the model
fit_predict(self, X[, y])	Applies fit_predict of last step in pipeline after transforms.
fit_transform(self, X[, y])	Fit the model and transform with the final estimator
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X, **predict_params)	Apply transforms to the data, and predict with the final estimator
predict_log_proba(self, X)	Apply transforms, and predict_log_proba of the final estimator
predict_proba(self, X)	Apply transforms, and predict_proba of the final estimator
score(self, X[, y, sample_weight])	Apply transforms, and score with the final estimator
set_params(self, **kwargs)	Set the parameters of this estimator.

__init__(*self*, *steps*, *memory=None*, *verbose=False*)

decision_function(*self*, *X*)

Apply transforms, and decision_function of the final estimator

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns

y_score [array-like, shape = [n_samples, n_classes]]

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

Fit the model

Fit all the transforms one after the other and transform the data, then fit the transformed data using the final estimator.

Parameters

X [iterable] Training data. Must fulfill input requirements of first step of the pipeline.

y [iterable, default=None] Training targets. Must fulfill label requirements for all steps of the pipeline.

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

Returns

self [Pipeline] This estimator

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

Applies fit_predict of last step in pipeline after transforms.

Applies fit_transforms of a pipeline to the data, followed by the fit_predict method of the final estimator in the pipeline. Valid only if the final estimator implements fit_predict.

Parameters

X [iterable] Training data. Must fulfill input requirements of first step of the pipeline.

y [iterable, default=None] Training targets. Must fulfill label requirements for all steps of the pipeline.

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

Returns

y_pred [array-like]

fit_transform(*self*, *X*, *y=None*, ***fit_params*)
Fit the model and transform with the final estimator

Fits all the transforms one after the other and transforms the data, then uses fit_transform on transformed data with the final estimator.

Parameters

X [iterable] Training data. Must fulfill input requirements of first step of the pipeline.

y [iterable, default=None] Training targets. Must fulfill label requirements for all steps of the pipeline.

****fit_params** [dict of string -> object] Parameters passed to the `fit` method of each step, where each parameter name is prefixed such that parameter `p` for step `s` has key `s__p`.

Returns

Xt [array-like, shape = [n_samples, n_transformed_features]] Transformed samples

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform
Apply inverse transformations in reverse order

All estimators in the pipeline must support `inverse_transform`.

Parameters

Xt [array-like, shape = [n_samples, n_transformed_features]] Data samples, where `n_samples` is the number of samples and `n_features` is the number of features. Must fulfill input requirements of last step of pipeline's `inverse_transform` method.

Returns

Xt [array-like, shape = [n_samples, n_features]]

predict(*self*, *X*, ***predict_params*)
Apply transforms to the data, and predict with the final estimator

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

****predict_params** [dict of string -> object] Parameters to the `predict` called at the end of all transformations in the pipeline. Note that while this may be used to return uncertainties from some models with return_std or return_cov, uncertainties that are generated by the transformations in the pipeline are not propagated to the final estimator.

Returns

y_pred [array-like]

predict_log_proba(*self*, *X*)
Apply transforms, and predict_log_proba of the final estimator

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns

y_score [array-like, shape = [n_samples, n_classes]]

predict_proba(*self*, *X*)
Apply transforms, and predict_proba of the final estimator

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

Returns

y_proba [array-like, shape = [n_samples, n_classes]]

score(*self*, *X*, *y=None*, *sample_weight=None*)
Apply transforms, and score with the final estimator

Parameters

X [iterable] Data to predict on. Must fulfill input requirements of first step of the pipeline.

y [iterable, default=None] Targets used for scoring. Must fulfill label requirements for all steps of the pipeline.

sample_weight [array-like, default=None] If not None, this argument is passed as `sample_weight` keyword argument to the `score` method of the final estimator.

Returns

score [float]

set_params(*self*, ***kwargs*)
Set the parameters of this estimator.

Valid parameter keys can be listed with `get_params()`.

Returns

self

transform
Apply transforms, and transform with the final estimator

This also works where final estimator is `None`: all prior transformations are applied.

Parameters

X [iterable] Data to transform. Must fulfill input requirements of first step of the pipeline.

Returns

Xt [array-like, shape = [n_samples, n_transformed_features]]

Examples using `sklearn.pipeline.Pipeline`

- *Explicit feature map approximation for RBF kernels*
- *Feature agglomeration vs. univariate selection*
- *Concatenating multiple feature extraction methods*
- *Pipelining: chaining a PCA and a logistic regression*
- *Column Transformer with Mixed Types*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *Column Transformer with Heterogeneous Data Sources*
- *Underfitting vs. Overfitting*
- *Balance model complexity and cross-validated score*
- *Sample pipeline for text feature extraction and evaluation*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Restricted Boltzmann Machine features for digit classification*
- *SVM-Anova: SVM with univariate feature selection*
- *Classification of text documents using sparse features*

pipeline.make_pipeline(*steps, **kwargs)	Construct a Pipeline from the given estimators.
pipeline.make_union(*transformers, **kwargs)	Construct a FeatureUnion from the given transformers.

6.32.3 `sklearn.pipeline`.make_pipeline

sklearn.pipeline.**make_pipeline**(**steps*, ***kwargs*)

Construct a Pipeline from the given estimators.

This is a shorthand for the Pipeline constructor; it does not require, and does not permit, naming the estimators. Instead, their names will be set to the lowercase of their types automatically.

Parameters

***steps** [list of estimators.]

memory [None, str or object with the joblib.Memory interface, optional] Used to cache the fitted transformers of the pipeline. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute `named_steps` or `steps` to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming.

verbose [boolean, optional] If True, the time elapsed while fitting each step will be printed as it is completed.

Returns

p [Pipeline]

See also:

sklearn.pipeline.Pipeline Class for creating a pipeline of transforms with a final estimator.

Examples

```
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.preprocessing import StandardScaler
>>> make_pipeline(StandardScaler(), GaussianNB(priors=None))
...
Pipeline(memory=None,
         steps=[('standardscaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('gaussiannb',
                 GaussianNB(priors=None, var_smoothing=1e-09))],
         verbose=False)
```

Examples using `sklearn.pipeline.make_pipeline`

- *Feature transformations with ensembles of trees*
- *Pipeline Anova SVM*
- *Imputing missing values with variants of IterativeImputer*
- *Imputing missing values before building an estimator*
- *Polynomial interpolation*
- *Robust linear estimator fitting*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Using FunctionTransformer to select columns*
- *Importance of Feature Scaling*
- *Feature discretization*
- *Clustering text documents using k-means*

6.32.4 `sklearn.pipeline`.make_union

`sklearn.pipeline.`**`make_union`**(**transformers*, ***kwargs*)

Construct a FeatureUnion from the given transformers.

This is a shorthand for the FeatureUnion constructor; it does not require, and does not permit, naming the transformers. Instead, they will be given names automatically based on their types. It also does not allow weighting.

Parameters

***transformers** [list of estimators]

n_jobs [int or None, optional (default=None)] Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [boolean, optional(default=False)] If True, the time elapsed while fitting each transformer will be printed as it is completed.

Returns

f [FeatureUnion]

See also:

sklearn.pipeline.FeatureUnion Class for concatenating the results of multiple transformer objects.

Examples

```
>>> from sklearn.decomposition import PCA, TruncatedSVD
>>> from sklearn.pipeline import make_union
>>> make_union(PCA(), TruncatedSVD())
FeatureUnion(n_jobs=None,
       transformer_list=[('pca',
                          PCA(copy=True, iterated_power='auto',
                              n_components=None, random_state=None,
                              svd_solver='auto', tol=0.0, whiten=False)),
                         ('truncatedsvd',
                          TruncatedSVD(algorithm='randomized',
                          n_components=2, n_iter=5,
                          random_state=None, tol=0.0))],
       transformer_weights=None, verbose=False)
```

Examples using `sklearn.pipeline.make_union`

- *Imputing missing values before building an estimator*

6.33 `sklearn.inspection`: inspection

The *sklearn.inspection* module includes tools for model inspection.

inspection.partial_dependence(estimator, X, ...)	Partial dependence of `features`.
inspection.plot_partial_dependence(..., [, ...])	Partial dependence plots.

6.33.1 `sklearn.inspection`.partial_dependence

sklearn.inspection.**partial_dependence**(*estimator*, *X*, *features*, *response_method='auto'*, *percentiles=(0.05, 0.95)*, *grid_resolution=100*, *method='auto'*)

Partial dependence of `features`.

Partial dependence of a feature (or a set of features) corresponds to the average response of an estimator for each possible value of the feature.

Read more in the *User Guide*.

Parameters

estimator [BaseEstimator] A fitted estimator object implementing *predict*, *predict_proba*, or *decision_function*. Multioutput-multiclass classifiers are not supported.

X [array-like, shape (n_samples, n_features)] `X` is used both to generate a grid of values for the `features`, and to compute the averaged predictions when method is 'brute'.

features [list or array-like of int] The target features for which the partial dependency should be computed.

response_method ['auto', 'predict_proba' or 'decision_function', optional (default='auto')] Specifies whether to use *predict_proba* or *decision_function* as the target response. For regressors this parameter is ignored and the response is always the output of *predict*. By default, *predict_proba* is tried first and we revert to *decision_function* if it doesn't exist. If `method` is 'recursion', the response is always the output of *decision_function*.

percentiles [tuple of float, optional (default=(0.05, 0.95))] The lower and upper percentile used to create the extreme values for the grid. Must be in [0, 1].

grid_resolution [int, optional (default=100)] The number of equally spaced points on the grid, for each target feature.

method [str, optional (default='auto')] The method used to calculate the averaged predictions:

- 'recursion' is only supported for objects inheriting from `BaseGradientBoosting`, but is more efficient in terms of speed. With this method, `X` is only used to build the grid and the partial dependences are computed using the training data. This method does not account for the `init` predicor of the boosting process, which may lead to incorrect values (see warning below). With this method, the target response of a classifier is always the decision function, not the predicted probabilities.
- 'brute' is supported for any estimator, but is more computationally intensive.
- If 'auto', then 'recursion' will be used for `BaseGradientBoosting` estimators with `init=None`, and 'brute' for all other.

Returns

averaged_predictions [ndarray, shape (n_outputs, len(values[0]), len(values[1]), ...)] The predictions for all the points in the grid, averaged over all samples in X (or over the training data if `method` is 'recursion'). `n_outputs` corresponds to the number of classes in a multi-class setting, or to the number of tasks for multi-output regression. For classical regression and binary classification `n_outputs==1`. `n_values_feature_j` corresponds to the size `values[j]`.

values [seq of 1d ndarrays] The values with which the grid has been created. The generated grid is a cartesian product of the arrays in `values`. `len(values) == len(features)`. The size of each array `values[j]` is either `grid_resolution`, or the number of unique values in `X[:, j]`, whichever is smaller.

Warning: The 'recursion' method only works for gradient boosting estimators, and unlike the 'brute' method, it does not account for the `init` predictor of the boosting process. In practice this will produce the same values as 'brute' up to a constant offset in the target response, provided that `init` is a consant estimator (which is the default). However, as soon as `init` is not a constant estimator, the partial dependence values are incorrect for 'recursion'.

See also:

sklearn.inspection.plot_partial_dependence Plot partial dependence

Examples

```
>>> X = [[0, 0, 2], [1, 0, 0]]
>>> y = [0, 1]
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> gb = GradientBoostingClassifier(random_state=0).fit(X, y)
>>> partial_dependence(gb, features=[0], X=X, percentiles=(0, 1),
...                    grid_resolution=2)
(array([[-4.52...,  4.52...]]), [array([ 0.,  1.])])
```

Examples using `sklearn.inspection.partial_dependence`

- *Partial Dependence Plots*

6.33.2 `sklearn.inspection`.plot_partial_dependence

`sklearn.inspection.`**`plot_partial_dependence`**(*estimator*, *X*, *features*, *feature_names=None*, *target=None*, *response_method='auto'*, *n_cols=3*, *grid_resolution=100*, *percentiles=(0.05, 0.95)*, *method='auto'*, *n_jobs=None*, *verbose=0*, *fig=None*, *line_kw=None*, *contour_kw=None*)

Partial dependence plots.

The `len(features)` plots are arranged in a grid with `n_cols` columns. Two-way partial dependence plots are plotted as contour plots.

Read more in the *User Guide*.

Parameters

estimator [BaseEstimator] A fitted estimator object implementing *predict*, *predict_proba*, or *decision_function*. Multioutput-multiclass classifiers are not supported.

X [array-like, shape (n_samples, n_features)] The data to use to build the grid of values on which the dependence will be evaluated. This is usually the training data.

features [list of {int, str, pair of int, pair of str}] The target features for which to create the PDPs. If features[i] is an int or a string, a one-way PDP is created; if features[i] is a tuple, a two-way PDP is created. Each tuple must be of size 2. if any entry is a string, then it must be in `feature_names`.

feature_names [seq of str, shape (n_features,), optional] Name of each feature; feature_names[i] holds the name of the feature with index i. By default, the name of the feature corresponds to their numerical index.

target [int, optional (default=None)]

- In a multiclass setting, specifies the class for which the PDPs should be computed. Note that for binary classification, the positive class (index 1) is always used.
- In a multioutput setting, specifies the task for which the PDPs should be computed

Ignored in binary classification or classical regression settings.

response_method ['auto', 'predict_proba' or 'decision_function', optional (default='auto') :] Specifies whether to use *predict_proba* or *decision_function* as the target response. For regressors this parameter is ignored and the response is always the output of *predict*. By

default, *predict_proba* is tried first and we revert to *decision_function* if it doesn't exist. If `method` is 'recursion', the response is always the output of *decision_function*.

n_cols [int, optional (default=3)] The maximum number of columns in the grid plot.

grid_resolution [int, optional (default=100)] The number of equally spaced points on the axes of the plots, for each target feature.

percentiles [tuple of float, optional (default=(0.05, 0.95))] The lower and upper percentile used to create the extreme values for the PDP axes. Must be in [0, 1].

method [str, optional (default='auto')] The method to use to calculate the partial dependence predictions:

- 'recursion' is only supported for objects inheriting from `BaseGradientBoosting`, but is more efficient in terms of speed. With this method, `X` is optional and is only used to build the grid and the partial dependences are computed using the training data. This method does not account for the `init` predicor of the boosting process, which may lead to incorrect values (see warning below. With this method, the target response of a classifier is always the decision function, not the predicted probabilities.
- 'brute' is supported for any estimator, but is more computationally intensive.
- If 'auto', then 'recursion' will be used for `BaseGradientBoosting` estimators with `init=None`, and 'brute' for all other.

Unlike the 'brute' method, 'recursion' does not account for the `init` predictor of the boosting process. In practice this still produces the same plots, up to a constant offset in the target response.

n_jobs [int, optional (default=None)] The number of CPUs to use to compute the partial dependences. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

verbose [int, optional (default=0)] Verbose output during PD computations.

fig [Matplotlib figure object, optional (default=None)] A figure object onto which the plots will be drawn, after the figure has been cleared. By default, a new one is created.

line_kw [dict, optional] Dict with keywords passed to the `matplotlib.pyplot.plot` call. For one-way partial dependence plots.

contour_kw [dict, optional] Dict with keywords passed to the `matplotlib.pyplot.plot` call. For two-way partial dependence plots.

Warning: The 'recursion' method only works for gradient boosting estimators, and unlike the 'brute' method, it does not account for the `init` predictor of the boosting process. In practice this will produce the same values as 'brute' up to a constant offset in the target response, provided that `init` is a consant estimator (which is the default). However, as soon as `init` is not a constant estimator, the partial dependence values are incorrect for 'recursion'.

See also:

sklearn.inspection.partial_dependence Return raw partial dependence values

Examples

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> X, y = make_friedman1()
>>> clf = GradientBoostingRegressor(n_estimators=10).fit(X, y)
>>> plot_partial_dependence(clf, X, [0, (0, 1)])
```

Examples using `sklearn.inspection.plot_partial_dependence`

- *Partial Dependence Plots*

6.34 `sklearn.preprocessing`: Preprocessing and Normalization

The *sklearn.preprocessing* module includes scaling, centering, normalization, binarization and imputation methods.

User guide: See the *Preprocessing data* section for further details.

preprocessing.Binarizer([threshold, copy])	Binarize data (set feature values to 0 or 1) according to a threshold
preprocessing.FunctionTransformer([func, ...])	Constructs a transformer from an arbitrary callable.
preprocessing.KBinsDiscretizer([n_bins, ...])	Bin continuous data into intervals.
preprocessing.KernelCenterer()	Center a kernel matrix
preprocessing.LabelBinarizer([neg_label, ...])	Binarize labels in a one-vs-all fashion
preprocessing.LabelEncoder	Encode labels with value between 0 and n_classes-1.
preprocessing.MultiLabelBinarizer([classes, ...])	Transform between iterable of iterables and a multilabel format
preprocessing.MaxAbsScaler([copy])	Scale each feature by its maximum absolute value.
preprocessing.MinMaxScaler([feature_range, copy])	Transforms features by scaling each feature to a given range.
preprocessing.Normalizer([norm, copy])	Normalize samples individually to unit norm.
preprocessing.OneHotEncoder([n_values, ...])	Encode categorical integer features as a one-hot numeric array.
preprocessing.OrdinalEncoder([categories, dtype])	Encode categorical features as an integer array.
preprocessing.PolynomialFeatures([degree, ...])	Generate polynomial and interaction features.
preprocessing.PowerTransformer([method, ...])	Apply a power transform featurewise to make data more Gaussian-like.
preprocessing.QuantileTransformer([...])	Transform features using quantiles information.
preprocessing.RobustScaler([with_centering, ...])	Scale features using statistics that are robust to outliers.
preprocessing.StandardScaler([copy, ...])	Standardize features by removing the mean and scaling to unit variance

6.34.1 sklearn.preprocessing.Binarizer

class sklearn.preprocessing.**Binarizer**(*threshold=0.0*, *copy=True*)

Binarize data (set feature values to 0 or 1) according to a threshold

Values greater than the threshold map to 1, while values less than or equal to the threshold map to 0. With the default threshold of 0, only positive values map to 1.

Binarization is a common operation on text count data where the analyst can decide to only consider the presence or absence of a feature rather than a quantified number of occurrences for instance.

It can also be used as a pre-processing step for estimators that consider boolean random variables (e.g. modelled using the Bernoulli distribution in a Bayesian setting).

Read more in the *User Guide*.

Parameters

threshold [float, optional (0.0 by default)] Feature values below or equal to this are replaced by 0, above it by 1. Threshold may not be less than 0 for operations on sparse matrices.

copy [boolean, optional, default True] set to False to perform inplace binarization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix).

See also:

binarize Equivalent function without the estimator API.

Notes

If the input is a sparse matrix, only the non-zero values are subject to update by the Binarizer class.

This estimator is stateless (besides constructor parameters), the fit method does nothing but is useful when used in a pipeline.

Examples

```
>>> from sklearn.preprocessing import Binarizer
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
>>> transformer = Binarizer().fit(X)  # fit does nothing.
>>> transformer
Binarizer(copy=True, threshold=0.0)
>>> transformer.transform(X)
array([[1., 0., 1.],
       [1., 0., 0.],
       [0., 1., 0.]])
```

Methods

fit(self, X[, y])	Do nothing and return the estimator unchanged
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.

Continued on next page

Table 6.247 – continued from previous page

set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, copy])	Binarize each element of X

__init__ (*self*, *threshold=0.0*, *copy=True*)

fit (*self*, *X*, *y=None*)
Do nothing and return the estimator unchanged

This method is just there to implement the usual API and hence work in pipelines.

Parameters

X [array-like]

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, ***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.

Returns

self

transform (*self*, *X*, *copy=None*)
Binarize each element of X

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data to binarize, element by element. scipy.sparse matrices should be in CSR format to avoid an un-necessary copy.

copy [bool] Copy the input X or not.

6.34.2 sklearn.preprocessing.FunctionTransformer

class sklearn.preprocessing.**FunctionTransformer**(*func=None*, *inverse_func=None*, *validate=None*, *accept_sparse=False*, *pass_y='deprecated'*, *check_inverse=True*, *kw_args=None*, *inv_kw_args=None*)

Constructs a transformer from an arbitrary callable.

A FunctionTransformer forwards its X (and optionally y) arguments to a user-defined function or function object and returns the result of this function. This is useful for stateless transformations such as taking the log of frequencies, doing custom scaling, etc.

Note: If a lambda is used as the function, then the resulting transformer will not be pickleable.

New in version 0.17.

Read more in the *User Guide*.

Parameters

func [callable, optional default=None] The callable to use for the transformation. This will be passed the same arguments as transform, with args and kwargs forwarded. If func is None, then func will be the identity function.

inverse_func [callable, optional default=None] The callable to use for the inverse transformation. This will be passed the same arguments as inverse transform, with args and kwargs forwarded. If inverse_func is None, then inverse_func will be the identity function.

validate [bool, optional default=True] Indicate that the input X array should be checked before calling `func`. The possibilities are:

- If False, there is no input validation.
- If True, then X will be converted to a 2-dimensional NumPy array or sparse matrix. If the conversion is not possible an exception is raised.

Deprecated since version 0.20: `validate=True` as default will be replaced by `validate=False` in 0.22.

accept_sparse [boolean, optional] Indicate that func accepts a sparse matrix as input. If validate is False, this has no effect. Otherwise, if accept_sparse is false, sparse matrix inputs will cause an exception to be raised.

pass_y [bool, optional default=False] Indicate that transform should forward the y argument to the inner callable.

Deprecated since version 0.19.

check_inverse [bool, default=True] Whether to check that or `func` followed by `inverse_func` leads to the original inputs. It can be used for a sanity check, raising a warning when the condition is not fulfilled.

New in version 0.20.

kw_args [dict, optional] Dictionary of additional keyword arguments to pass to func.

inv_kw_args [dict, optional] Dictionary of additional keyword arguments to pass to inverse_func.

Methods

fit(self, X[, y])	Fit transformer by checking X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Transform X using the inverse function.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X using the forward function.

__init__(*self*, *func=None*, *inverse_func=None*, *validate=None*, *accept_sparse=False*, *pass_y='deprecated'*, *check_inverse=True*, *kw_args=None*, *inv_kw_args=None*)

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

Fit transformer by checking X.

If `validate` is `True`, `X` will be checked.

Parameters

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

Returns

self

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*)

Transform X using the inverse function.

Parameters

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

Returns

X_out [array-like, shape (n_samples, n_features)] Transformed input.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform X using the forward function.

Parameters

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

Returns

X_out [array-like, shape (n_samples, n_features)] Transformed input.

Examples using `sklearn.preprocessing.FunctionTransformer`

- *Using FunctionTransformer to select columns*

6.34.3 sklearn.preprocessing.KBinsDiscretizer

class sklearn.preprocessing.**KBinsDiscretizer**(*n_bins=5*, *encode='onehot'*, *strategy='quantile'*)
Bin continuous data into intervals.

Read more in the *User Guide*.

Parameters

n_bins [int or array-like, shape (n_features,) (default=5)] The number of bins to produce. Raises ValueError if `n_bins < 2`.

encode [{'onehot', 'onehot-dense', 'ordinal'}, (default='onehot')] Method used to encode the transformed result.

onehot Encode the transformed result with one-hot encoding and return a sparse matrix. Ignored features are always stacked to the right.

onehot-dense Encode the transformed result with one-hot encoding and return a dense array. Ignored features are always stacked to the right.

ordinal Return the bin identifier encoded as an integer value.

strategy [{'uniform', 'quantile', 'kmeans'}, (default='quantile')] Strategy used to define the widths of the bins.

uniform All bins in each feature have identical widths.

quantile All bins in each feature have the same number of points.

kmeans Values in each bin have the same nearest center of a 1D k-means cluster.

Attributes

n_bins_ [int array, shape (n_features,)] Number of bins per feature. Bins whose width are too small (i.e., <= 1e-8) are removed with a warning.

bin_edges_ [array of arrays, shape (n_features,)] The edges of each bin. Contain arrays of varying shapes `(n_bins_, )` Ignored features will have empty arrays.

See also:

sklearn.preprocessing.Binarizer class used to bin values as `0` or `1` based on a parameter `threshold`.

Notes

In bin edges for feature `i`, the first and last values are used only for `inverse_transform`. During transform, bin edges are extended to:

```
np.concatenate([-np.inf, bin_edges_[i][1:-1], np.inf])
```

You can combine `KBinsDiscretizer` with *sklearn.compose.ColumnTransformer* if you only want to preprocess part of the features.

`KBinsDiscretizer` might produce constant features (e.g., when `encode = 'onehot'` and certain bins do not contain any data). These features can be removed with feature selection algorithms (e.g., *sklearn.feature_selection.VarianceThreshold*).

Examples

```
>>> X = [[-2, 1, -4,   -1],
...      [-1, 2, -3, -0.5],
...      [ 0, 3, -2,  0.5],
...      [ 1, 4, -1,    2]]
>>> est = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='uniform')
>>> est.fit(X)
KBinsDiscretizer(...)
>>> Xt = est.transform(X)
>>> Xt
array([[ 0., 0., 0., 0.],
       [ 1., 1., 1., 0.],
       [ 2., 2., 2., 1.],
       [ 2., 2., 2., 2.]])
```

Sometimes it may be useful to convert the data back into the original feature space. The `inverse_transform` function converts the binned data into the original feature space. Each value will be equal to the mean of the two bin edges.

```
>>> est.bin_edges_[0]
array([-2., -1.,  0.,  1.])
>>> est.inverse_transform(Xt)
array([[-1.5,  1.5, -3.5, -0.5],
       [-0.5,  2.5, -2.5, -0.5],
       [ 0.5,  3.5, -1.5,  0.5],
       [ 0.5,  3.5, -1.5,  1.5]])
```

Methods

fit(self, X[, y])	Fits the estimator.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Xt)	Transforms discretized data back to original feature space.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Discretizes the data.

__init__(*self*, *n_bins=5*, *encode='onehot'*, *strategy='quantile'*)

fit(*self*, *X*, *y=None*)
Fits the estimator.

Parameters

X [numeric array-like, shape (n_samples, n_features)] Data to be discretized.

y [ignored]

Returns

self

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *Xt*)
Transforms discretized data back to original feature space.

Note that this function does not regenerate the original data due to discretization rounding.

Parameters

Xt [numeric array-like, shape (n_sample, n_features)] Transformed data in the binned space.

Returns

Xinv [numeric array-like] Data in the original feature space.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Discretizes the data.

Parameters

X [numeric array-like, shape (n_samples, n_features)] Data to be discretized.

Returns

Xt [numeric array-like or sparse matrix] Data in the binned space.

Examples using `sklearn.preprocessing.KBinsDiscretizer`

- *Using KBinsDiscretizer to discretize continuous features*
- *Demonstrating the different strategies of KBinsDiscretizer*
- *Feature discretization*

6.34.4 `sklearn.preprocessing`.KernelCenterer

class `sklearn.preprocessing.`**KernelCenterer**
Center a kernel matrix

Let K(x, z) be a kernel defined by phi(x)^T phi(z), where phi is a function mapping x to a Hilbert space. KernelCenterer centers (i.e., normalize to have zero mean) the data without explicitly computing phi(x). It is equivalent to centering phi(x) with sklearn.preprocessing.StandardScaler(with_std=False).

Read more in the *User Guide*.

Examples

```
>>> from sklearn.preprocessing import KernelCenterer
>>> from sklearn.metrics.pairwise import pairwise_kernels
>>> X = [[ 1., -2.,  2.],
...      [ -2.,  1.,  3.],
...      [ 4.,  1., -2.]]
>>> K = pairwise_kernels(X, metric='linear')
>>> K
array([[  9.,   2.,  -2.],
       [  2.,  14., -13.],
       [ -2., -13.,  21.]])
>>> transformer = KernelCenterer().fit(K)
>>> transformer
KernelCenterer()
>>> transformer.transform(K)
array([[  5.,   0.,  -5.],
```

```
       [  0.,  14., -14.],
       [ -5., -14.,  19.]])
```

Methods

fit(self, K[, y])	Fit KernelCenterer
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, K[, copy])	Center kernel matrix.

__init__(*self*)

fit(*self*, *K*, *y=None*)
Fit KernelCenterer

Parameters

K [numpy array of shape [n_samples, n_samples]] Kernel matrix.

Returns

self [returns an instance of self.]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *K*, *copy=True*)
Center kernel matrix.

Parameters

K [numpy array of shape [n_samples1, n_samples2]] Kernel matrix.

copy [boolean, optional, default True] Set to False to perform inplace computation.

Returns

K_new [numpy array of shape [n_samples1, n_samples2]]

6.34.5 sklearn.preprocessing.LabelBinarizer

class sklearn.preprocessing.**LabelBinarizer**(*neg_label=0*, *pos_label=1*, *sparse_output=False*)
Binarize labels in a one-vs-all fashion

Several regression and binary classification algorithms are available in scikit-learn. A simple way to extend these algorithms to the multi-class classification case is to use the so-called one-vs-all scheme.

At learning time, this simply consists in learning one regressor or binary classifier per class. In doing so, one needs to convert multi-class labels to binary labels (belong or does not belong to the class). LabelBinarizer makes this process easy with the transform method.

At prediction time, one assigns the class for which the corresponding model gave the greatest confidence. LabelBinarizer makes this easy with the inverse_transform method.

Read more in the *User Guide*.

Parameters

neg_label [int (default: 0)] Value with which negative labels must be encoded.

pos_label [int (default: 1)] Value with which positive labels must be encoded.

sparse_output [boolean (default: False)] True if the returned array from transform is desired to be in sparse CSR format.

Attributes

classes_ [array of shape [n_class]] Holds the label for each class.

y_type_ [str,] Represents the type of the target data as evaluated by utils.multiclass.type_of_target. Possible type are 'continuous', 'continuous-multioutput', 'binary', 'multiclass', 'multiclass-multioutput', 'multilabel-indicator', and 'unknown'.

sparse_input_ [boolean,] True if the input data to transform is given as a sparse matrix, False otherwise.

See also:

label_binarize function to perform the transform operation of LabelBinarizer with fixed classes.

sklearn.preprocessing.OneHotEncoder encode categorical features using a one-hot aka one-of-K scheme.

Examples

```
>>> from sklearn import preprocessing
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer(neg_label=0, pos_label=1, sparse_output=False)
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])
```

Binary targets transform to a column vector

```
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit_transform(['yes', 'no', 'no', 'yes'])
array([[1],
       [0],
       [0],
       [1]])
```

Passing a 2D matrix for multilabel classification

```
>>> import numpy as np
>>> lb.fit(np.array([[0, 1, 1], [1, 0, 0]]))
LabelBinarizer(neg_label=0, pos_label=1, sparse_output=False)
>>> lb.classes_
array([0, 1, 2])
>>> lb.transform([0, 1, 2, 1])
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1],
       [0, 1, 0]])
```

Methods

fit(self, y)	Fit label binarizer
fit_transform(self, y)	Fit label binarizer and transform multi-class labels to binary labels.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Y[, threshold])	Transform binary labels back to multi-class labels
set_params(self, **params)	Set the parameters of this estimator.
transform(self, y)	Transform multi-class labels to binary labels

__init__ (*self*, *neg_label=0*, *pos_label=1*, *sparse_output=False*)

fit (*self*, *y*)

Fit label binarizer

Parameters

y [array of shape [n_samples,] or [n_samples, n_classes]] Target values. The 2-d matrix should only contain 0 and 1, represents multilabel classification.

Returns

self [returns an instance of self.]

fit_transform(*self*, *y*)

Fit label binarizer and transform multi-class labels to binary labels.

The output of transform is sometimes referred to as the 1-of-K coding scheme.

Parameters

y [array or sparse matrix of shape [n_samples,] or [n_samples, n_classes]] Target values. The 2-d matrix should only contain 0 and 1, represents multilabel classification. Sparse matrix can be CSR, CSC, COO, DOK, or LIL.

Returns

Y [array or CSR matrix of shape [n_samples, n_classes]] Shape will be [n_samples, 1] for binary problems.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *Y*, *threshold=None*)

Transform binary labels back to multi-class labels

Parameters

Y [numpy array or sparse matrix with shape [n_samples, n_classes]] Target values. All sparse matrices are converted to CSR before inverse transformation.

threshold [float or None] Threshold used in the binary and multi-label cases.

Use 0 when `Y` contains the output of decision_function (classifier). Use 0.5 when `Y` contains the output of predict_proba.

If None, the threshold is assumed to be half way between neg_label and pos_label.

Returns

y [numpy array or CSR matrix of shape [n_samples] Target values.]

Notes

In the case when the binary labels are fractional (probabilistic), inverse_transform chooses the class with the greatest value. Typically, this allows to use the output of a linear model's decision_function method directly as the input of inverse_transform.

set_params(*self*, ***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.

Returns

self

transform(*self*, *y*)

Transform multi-class labels to binary labels

The output of transform is sometimes referred to by some authors as the 1-of-K coding scheme.

Parameters

y [array or sparse matrix of shape [n_samples,] or [n_samples, n_classes]] Target values. The 2-d matrix should only contain 0 and 1, represents multilabel classification. Sparse matrix can be CSR, CSC, COO, DOK, or LIL.

Returns

Y [numpy array or CSR matrix of shape [n_samples, n_classes]] Shape will be [n_samples, 1] for binary problems.

6.34.6 sklearn.preprocessing.LabelEncoder

class sklearn.preprocessing.**LabelEncoder**

Encode labels with value between 0 and n_classes-1.

Read more in the *User Guide*.

Attributes

classes_ [array of shape (n_class,)] Holds the label for each class.

See also:

sklearn.preprocessing.OrdinalEncoder encode categorical features using a one-hot or ordinal encoding scheme.

Examples

LabelEncoder can be used to normalize labels.

```
>>> from sklearn import preprocessing
>>> le = preprocessing.LabelEncoder()
>>> le.fit([1, 2, 2, 6])
LabelEncoder()
>>> le.classes_
array([1, 2, 6])
>>> le.transform([1, 1, 2, 6])
array([0, 0, 1, 2]...)
>>> le.inverse_transform([0, 0, 1, 2])
array([1, 1, 2, 6])
```

It can also be used to transform non-numerical labels (as long as they are hashable and comparable) to numerical labels.

```
>>> le = preprocessing.LabelEncoder()
>>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
LabelEncoder()
>>> list(le.classes_)
['amsterdam', 'paris', 'tokyo']
>>> le.transform(["tokyo", "tokyo", "paris"])
array([2, 2, 1]...)
```

```
>>> list(le.inverse_transform([2, 2, 1]))
['tokyo', 'tokyo', 'paris']
```

Methods

fit(self, y)	Fit label encoder
fit_transform(self, y)	Fit label encoder and return encoded labels
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, y)	Transform labels back to original encoding.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, y)	Transform labels to normalized encoding.

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

fit (*self*, *y*)
Fit label encoder

Parameters

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

Returns

self [returns an instance of self.]

fit_transform (*self*, *y*)
Fit label encoder and return encoded labels

Parameters

y [array-like of shape [n_samples]] Target values.

Returns

y [array-like of shape [n_samples]]

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform (*self*, *y*)
Transform labels back to original encoding.

Parameters

y [numpy array of shape [n_samples]] Target values.

Returns

y [numpy array of shape [n_samples]]

set_params(*self*, ***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.

Returns

self

transform(*self*, *y*)
Transform labels to normalized encoding.

Parameters

y [array-like of shape [n_samples]] Target values.

Returns

y [array-like of shape [n_samples]]

6.34.7 sklearn.preprocessing.MultiLabelBinarizer

class sklearn.preprocessing.**MultiLabelBinarizer**(*classes=None*, *sparse_output=False*)
Transform between iterable of iterables and a multilabel format

Although a list of sets or tuples is a very intuitive format for multilabel data, it is unwieldy to process. This transformer converts between this intuitive format and the supported multilabel format: a (samples x classes) binary matrix indicating the presence of a class label.

Parameters

classes [array-like of shape [n_classes] (optional)] Indicates an ordering for the class labels. All entries should be unique (cannot contain duplicate classes).

sparse_output [boolean (default: False),] Set to true if output binary array is desired in CSR sparse format

Attributes

classes_ [array of labels] A copy of the `classes` parameter where provided, or otherwise, the sorted set of classes found when fitting.

See also:

sklearn.preprocessing.OneHotEncoder encode categorical features using a one-hot aka one-of-K scheme.

Examples

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> mlb = MultiLabelBinarizer()
>>> mlb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
       [0, 0, 1]])
>>> mlb.classes_
array([1, 2, 3])
```

```
>>> mlb.fit_transform([{'sci-fi', 'thriller'}, {'comedy'}])
array([[0, 1, 1],
       [1, 0, 0]])
>>> list(mlb.classes_)
['comedy', 'sci-fi', 'thriller']
```

Methods

fit(self, y)	Fit the label sets binarizer, storing *classes_*
fit_transform(self, y)	Fit the label sets binarizer and transform the given label sets
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, yt)	Transform the given indicator matrix into label sets
set_params(self, **params)	Set the parameters of this estimator.
transform(self, y)	Transform the given label sets

__init__ (*self*, *classes=None*, *sparse_output=False*)

fit (*self*, *y*)
Fit the label sets binarizer, storing *classes_*

Parameters

y [iterable of iterables] A set of labels (any orderable and hashable object) for each sample. If the `classes` parameter is set, *y* will not be iterated.

Returns

self [returns this MultiLabelBinarizer instance]

fit_transform (*self*, *y*)
Fit the label sets binarizer and transform the given label sets

Parameters

y [iterable of iterables] A set of labels (any orderable and hashable object) for each sample. If the `classes` parameter is set, *y* will not be iterated.

Returns

y_indicator [array or CSR matrix, shape (n_samples, n_classes)] A matrix such that `y_indicator[i, j] = 1` iff `classes_[j]` is in `y[i]`, and 0 otherwise.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform (*self*, *yt*)
Transform the given indicator matrix into label sets

Parameters

yt [array or sparse matrix of shape (n_samples, n_classes)] A matrix containing only 1s ands 0s.

Returns

y [list of tuples] The set of labels for each sample such that `y[i]` consists of `classes_[j]` for each `yt[i, j] == 1`.

set_params(*self*, ***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.

Returns

self

transform(*self*, *y*)
Transform the given label sets

Parameters

y [iterable of iterables] A set of labels (any orderable and hashable object) for each sample. If the `classes` parameter is set, *y* will not be iterated.

Returns

y_indicator [array or CSR matrix, shape (n_samples, n_classes)] A matrix such that `y_indicator[i, j] = 1` iff `classes_[j]` is in `y[i]`, and 0 otherwise.

6.34.8 sklearn.preprocessing.MaxAbsScaler

class sklearn.preprocessing.**MaxAbsScaler**(*copy=True*)
Scale each feature by its maximum absolute value.

This estimator scales and translates each feature individually such that the maximal absolute value of each feature in the training set will be 1.0. It does not shift/center the data, and thus does not destroy any sparsity.

This scaler can also be applied to sparse CSR or CSC matrices.

New in version 0.17.

Parameters

copy [boolean, optional, default is True] Set to False to perform inplace scaling and avoid a copy (if the input is already a numpy array).

Attributes

scale_ [ndarray, shape (n_features,)] Per feature relative scaling of the data.

New in version 0.17: *scale_* attribute.

max_abs_ [ndarray, shape (n_features,)] Per feature maximum absolute value.

n_samples_seen_ [int] The number of samples processed by the estimator. Will be reset on new calls to fit, but increments across `partial_fit` calls.

See also:

maxabs_scale Equivalent function without the estimator API.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> from sklearn.preprocessing import MaxAbsScaler
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
>>> transformer = MaxAbsScaler().fit(X)
>>> transformer
MaxAbsScaler(copy=True)
>>> transformer.transform(X)
array([[ 0.5, -1. ,  1. ],
       [ 1. ,  0. ,  0. ],
       [ 0. ,  1. , -0.5]])
```

Methods

fit(self, X[, y])	Compute the maximum absolute value to be used for later scaling.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Scale back the data to the original representation
partial_fit(self, X[, y])	Online computation of max absolute value of X for later scaling.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Scale the data

__init__(*self*, *copy=True*)

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

Compute the maximum absolute value to be used for later scaling.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the per-feature minimum and maximum used for later scaling along the features axis.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*)
Scale back the data to the original representation

Parameters

X [{array-like, sparse matrix}] The data that should be transformed back.

partial_fit(*self*, *X*, *y=None*)
Online computation of max absolute value of X for later scaling. All of X is processed as a single batch. This is intended for cases when `fit` is not feasible due to very large number of *n_samples* or because X is read from a continuous stream.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y Ignored

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Scale the data

Parameters

X [{array-like, sparse matrix}] The data that should be scaled.

Examples using `sklearn.preprocessing.MaxAbsScaler`

- *Compare the effect of different scalers on data with outliers*

6.34.9 sklearn.preprocessing.MinMaxScaler

class sklearn.preprocessing.**MinMaxScaler**(*feature_range=(0, 1)*, *copy=True*)
Transforms features by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one.

The transformation is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where min, max = feature_range.

The transformation is calculated as:

```
X_scaled = scale * X + min - X.min(axis=0) * scale
where scale = (max - min) / (X.max(axis=0) - X.min(axis=0))
```

This transformation is often used as an alternative to zero mean, unit variance scaling.

Read more in the *User Guide*.

Parameters

feature_range [tuple (min, max), default=(0, 1)] Desired range of transformed data.

copy [boolean, optional, default True] Set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array).

Attributes

min_ [ndarray, shape (n_features,)] Per feature adjustment for minimum. Equivalent to `min - X.min(axis=0) * self.scale_`

scale_ [ndarray, shape (n_features,)] Per feature relative scaling of the data. Equivalent to `(max - min) / (X.max(axis=0) - X.min(axis=0))`

New in version 0.17: *scale_* attribute.

data_min_ [ndarray, shape (n_features,)] Per feature minimum seen in the data

New in version 0.17: *data_min_*

data_max_ [ndarray, shape (n_features,)] Per feature maximum seen in the data

New in version 0.17: *data_max_*

data_range_ [ndarray, shape (n_features,)] Per feature range `(data_max_ - data_min_)` seen in the data

New in version 0.17: *data_range_*

See also:

minmax_scale Equivalent function without the estimator API.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
>>> scaler = MinMaxScaler()
>>> print(scaler.fit(data))
MinMaxScaler(copy=True, feature_range=(0, 1))
>>> print(scaler.data_max_)
[ 1. 18.]
>>> print(scaler.transform(data))
[[0.   0.  ]
 [0.25 0.25]
 [0.5  0.5 ]
 [1.   1.  ]]
>>> print(scaler.transform([[2, 2]]))
[[1.5 0. ]]
```

Methods

fit(self, X[, y])	Compute the minimum and maximum to be used for later scaling.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Undo the scaling of X according to feature_range.
partial_fit(self, X[, y])	Online computation of min and max on X for later scaling.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Scaling features of X according to feature_range.

__init__ (*self*, *feature_range=(0, 1)*, *copy=True*)

fit (*self*, *X*, *y=None*)
Compute the minimum and maximum to be used for later scaling.

Parameters

X [array-like, shape [n_samples, n_features]] The data used to compute the per-feature minimum and maximum used for later scaling along the features axis.

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*)
Undo the scaling of X according to feature_range.

Parameters

X [array-like, shape [n_samples, n_features]] Input data that will be transformed. It cannot be sparse.

partial_fit(*self*, *X*, *y=None*)
Online computation of min and max on X for later scaling. All of X is processed as a single batch. This is intended for cases when `fit` is not feasible due to very large number of *n_samples* or because X is read from a continuous stream.

Parameters

X [array-like, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y Ignored

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Scaling features of X according to feature_range.

Parameters

X [array-like, shape [n_samples, n_features]] Input data that will be transformed.

Examples using `sklearn.preprocessing.MinMaxScaler`

- *Compare Stochastic learning strategies for MLPClassifier*
- *Compare the effect of different scalers on data with outliers*

6.34.10 `sklearn.preprocessing`.Normalizer

class sklearn.preprocessing.**Normalizer**(*norm='l2'*, *copy=True*)
Normalize samples individually to unit norm.

Each sample (i.e. each row of the data matrix) with at least one non zero component is rescaled independently of other samples so that its norm (l1 or l2) equals one.

This transformer is able to work both with dense numpy arrays and scipy.sparse matrix (use CSR format if you want to avoid the burden of a copy / conversion).

Scaling inputs to unit norms is a common operation for text classification or clustering for instance. For instance the dot product of two l2-normalized TF-IDF vectors is the cosine similarity of the vectors and is the base similarity metric for the Vector Space Model commonly used by the Information Retrieval community.

Read more in the *User Guide*.

Parameters

norm ['l1', 'l2', or 'max', optional ('l2' by default)] The norm to use to normalize each non zero sample.

copy [boolean, optional, default True] set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix).

See also:

normalize Equivalent function without the estimator API.

Notes

This estimator is stateless (besides constructor parameters), the fit method does nothing but is useful when used in a pipeline.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> from sklearn.preprocessing import Normalizer
>>> X = [[4, 1, 2, 2],
...      [1, 3, 9, 3],
...      [5, 7, 5, 1]]
>>> transformer = Normalizer().fit(X)  # fit does nothing.
>>> transformer
Normalizer(copy=True, norm='l2')
>>> transformer.transform(X)
array([[0.8, 0.2, 0.4, 0.4],
       [0.1, 0.3, 0.9, 0.3],
       [0.5, 0.7, 0.5, 0.1]])
```

Methods

fit(self, X[, y])	Do nothing and return the estimator unchanged
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, copy])	Scale each non zero row of X to unit norm

__init__ (*self*, *norm='l2'*, *copy=True*)

fit (*self*, *X*, *y=None*)
Do nothing and return the estimator unchanged

This method is just there to implement the usual API and hence work in pipelines.

Parameters

X [array-like]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*, *copy=None*)
Scale each non zero row of X to unit norm

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data to normalize, row by row. scipy.sparse matrices should be in CSR format to avoid an un-necessary copy.

copy [bool, optional (default: None)] Copy the input X or not.

Examples using `sklearn.preprocessing.Normalizer`

- *Compare the effect of different scalers on data with outliers*
- *Clustering text documents using k-means*

6.34.11 sklearn.preprocessing.OneHotEncoder

class sklearn.preprocessing.**OneHotEncoder**(*n_values=None*, *categorical_features=None*, *categories=None*, *drop=None*, *sparse=True*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*)

Encode categorical integer features as a one-hot numeric array.

The input to this transformer should be an array-like of integers or strings, denoting the values taken on by categorical (discrete) features. The features are encoded using a one-hot (aka 'one-of-K' or 'dummy') encoding scheme. This creates a binary column for each category and returns a sparse matrix or dense array.

By default, the encoder derives the categories based on the unique values in each feature. Alternatively, you can also specify the `categories` manually. The OneHotEncoder previously assumed that the input features take on values in the range [0, max(values)). This behaviour is deprecated.

This encoding is needed for feeding categorical data to many scikit-learn estimators, notably linear models and SVMs with the standard kernels.

Note: a one-hot encoding of y labels should use a LabelBinarizer instead.

Read more in the *User Guide*.

Parameters

categories ['auto' or a list of lists/arrays of values, default='auto'.] Categories (unique values) per feature:

- 'auto' : Determine categories automatically from the training data.
- list : `categories[i]` holds the categories expected in the ith column. The passed categories should not mix strings and numeric values within a single feature, and should be sorted in case of numeric values.

The used categories can be found in the `categories_` attribute.

drop ['first' or a list/array of shape (n_features,), default=None.] Specifies a methodology to use to drop one of the categories per feature. This is useful in situations where perfectly collinear features cause problems, such as when feeding the resulting data into a neural network or an unregularized regression.

- None : retain all features (the default).
- 'first' : drop the first category in each feature. If only one category is present, the feature will be dropped entirely.
- array : `drop[i]` is the category in feature `X[:, i]` that should be dropped.

sparse [boolean, default=True] Will return sparse matrix if set True else will return an array.

dtype [number type, default=np.float] Desired dtype of output.

handle_unknown ['error' or 'ignore', default='error'.] Whether to raise an error or ignore if an unknown categorical feature is present during transform (default is to raise). When this parameter is set to 'ignore' and an unknown category is encountered during transform, the resulting one-hot encoded columns for this feature will be all zeros. In the inverse transform, an unknown category will be denoted as None.

n_values ['auto', int or array of ints, default='auto'] Number of values per feature.

- 'auto' : determine value range from training data.
- **int** [number of categorical values per feature.] Each feature value should be in `range(n_values)`

- **array** [n_values[i] is the number of categorical values in] X[:, i]. Each feature value should be in range(n_values[i])

Deprecated since version 0.20: The n_values keyword was deprecated in version 0.20 and will be removed in 0.22. Use categories instead.

categorical_features ['all' or array of indices or mask, default='all'] Specify what features are treated as categorical.

- 'all': All features are treated as categorical.
- array of indices: Array of categorical feature indices.
- mask: Array of length n_features and with dtype=bool.

Non-categorical features are always stacked to the right of the matrix.

Deprecated since version 0.20: The categorical_features keyword was deprecated in version 0.20 and will be removed in 0.22. You can use the ColumnTransformer instead.

Attributes

categories_ [list of arrays] The categories of each feature determined during fitting (in order of the features in X and corresponding with the output of transform). This includes the category specified in drop (if any).

drop_idx_ [array of shape (n_features,)] drop_idx_[i] is the index in categories_[i] of the category to be dropped for each feature. None if all the transformed features will be retained.

active_features_ [array] Indices for active features, meaning values that actually occur in the training set. Only available when n_values is 'auto'.

Deprecated since version 0.20: The active_features_ attribute was deprecated in version 0.20 and will be removed in 0.22.

feature_indices_ [array of shape (n_features,)] Indices to feature ranges. Feature i in the original data is mapped to features from feature_indices_[i] to feature_indices_[i+1] (and then potentially masked by active_features_ afterwards)

Deprecated since version 0.20: The feature_indices_ attribute was deprecated in version 0.20 and will be removed in 0.22.

n_values_ [array of shape (n_features,)] Maximum number of values per feature.

Deprecated since version 0.20: The n_values_ attribute was deprecated in version 0.20 and will be removed in 0.22.

See also:

sklearn.preprocessing.OrdinalEncoder performs an ordinal (integer) encoding of the categorical features.

sklearn.feature_extraction.DictVectorizer performs a one-hot encoding of dictionary items (also handles string-valued features).

sklearn.feature_extraction.FeatureHasher performs an approximate one-hot encoding of dictionary items or strings.

sklearn.preprocessing.LabelBinarizer binarizes labels in a one-vs-all fashion.

sklearn.preprocessing.MultiLabelBinarizer transforms between iterable of iterables and a multilabel format, e.g. a (samples x classes) binary matrix indicating the presence of a class label.

Examples

Given a dataset with two features, we let the encoder find the unique values per feature and transform the data to a binary one-hot encoding.

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> enc = OneHotEncoder(handle_unknown='ignore')
>>> X = [['Male', 1], ['Female', 3], ['Female', 2]]
>>> enc.fit(X)
...
...
OneHotEncoder(categorical_features=None, categories=None, drop=None,
   dtype=<... 'numpy.float64'>, handle_unknown='ignore',
   n_values=None, sparse=True)
```

```
>>> enc.categories_
[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
>>> enc.transform([['Female', 1], ['Male', 4]]).toarray()
array([[1., 0., 1., 0., 0.],
       [0., 1., 0., 0., 0.]])
>>> enc.inverse_transform([[0, 1, 1, 0, 0], [0, 0, 0, 1, 0]])
array([['Male', 1],
       [None, 2]], dtype=object)
>>> enc.get_feature_names()
array(['x0_Female', 'x0_Male', 'x1_1', 'x1_2', 'x1_3'], dtype=object)
>>> drop_enc = OneHotEncoder(drop='first').fit(X)
>>> drop_enc.categories_
[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
>>> drop_enc.transform([['Female', 1], ['Male', 2]]).toarray()
array([[0., 0., 0.],
       [1., 1., 0.]])
```

Methods

fit(self, X[, y])	Fit OneHotEncoder to X.
fit_transform(self, X[, y])	Fit OneHotEncoder to X, then transform X.
get_feature_names(self[, input_features])	Return feature names for output features.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Convert the back data to the original representation.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X using one-hot encoding.

__init__(*self*, *n_values=None*, *categorical_features=None*, *categories=None*, *drop=None*, *sparse=True*, *dtype=<class 'numpy.float64'>*, *handle_unknown='error'*)

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

Fit OneHotEncoder to X.

Parameters

X [array-like, shape [n_samples, n_features]] The data to determine the categories of each

feature.

Returns

self

fit_transform(*self*, *X*, *y=None*)
Fit OneHotEncoder to X, then transform X.

Equivalent to fit(X).transform(X) but more convenient.

Parameters

X [array-like, shape [n_samples, n_features]] The data to encode.

Returns

X_out [sparse matrix if sparse=True else a 2-d array] Transformed input.

get_feature_names(*self*, *input_features=None*)
Return feature names for output features.

Parameters

input_features [list of string, length n_features, optional] String names for input features if available. By default, "x0", "x1", ... "xn_features" is used.

Returns

output_feature_names [array of string, length n_output_features]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*)
Convert the back data to the original representation.

In case unknown categories are encountered (all zeros in the one-hot encoding), `None` is used to represent this category.

Parameters

X [array-like or sparse matrix, shape [n_samples, n_encoded_features]] The transformed data.

Returns

X_tr [array-like, shape [n_samples, n_features]] Inverse transformed array.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform X using one-hot encoding.

Parameters

X [array-like, shape [n_samples, n_features]] The data to encode.

Returns

X_out [sparse matrix if sparse=True else a 2-d array] Transformed input.

Examples using `sklearn.preprocessing.OneHotEncoder`

- *Column Transformer with Mixed Types*
- *Feature transformations with ensembles of trees*

6.34.12 `sklearn.preprocessing`.OrdinalEncoder

class `sklearn.preprocessing.`**OrdinalEncoder**(*categories='auto'*, *dtype=<class 'numpy.float64'>*)
Encode categorical features as an integer array.

The input to this transformer should be an array-like of integers or strings, denoting the values taken on by categorical (discrete) features. The features are converted to ordinal integers. This results in a single column of integers (0 to n_categories - 1) per feature.

Read more in the *User Guide*.

Parameters

categories ['auto' or a list of lists/arrays of values.] Categories (unique values) per feature:

- 'auto' : Determine categories automatically from the training data.
- list : `categories[i]` holds the categories expected in the ith column. The passed categories should not mix strings and numeric values, and should be sorted in case of numeric values.

The used categories can be found in the `categories_` attribute.

dtype [number type, default np.float64] Desired dtype of output.

Attributes

categories_ [list of arrays] The categories of each feature determined during fitting (in order of the features in X and corresponding with the output of `transform`).

See also:

sklearn.preprocessing.OneHotEncoder performs a one-hot encoding of categorical features.

sklearn.preprocessing.LabelEncoder encodes target labels with values between 0 and n_classes-1.

Examples

Given a dataset with two features, we let the encoder find the unique values per feature and transform the data to an ordinal encoding.

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> enc = OrdinalEncoder()
>>> X = [['Male', 1], ['Female', 3], ['Female', 2]]
>>> enc.fit(X)
...
OrdinalEncoder(categories='auto', dtype=<... 'numpy.float64'>)
>>> enc.categories_
[array(['Female', 'Male'], dtype=object), array([1, 2, 3], dtype=object)]
>>> enc.transform([['Female', 3], ['Male', 1]])
array([[0., 2.],
       [1., 0.]])
```

```
>>> enc.inverse_transform([[1, 0], [0, 1]])
array([['Male', 1],
       ['Female', 2]], dtype=object)
```

Methods

fit(self, X[, y])	Fit the OrdinalEncoder to X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Convert the data back to the original representation.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform X to ordinal codes.

__init__(*self*, *categories='auto'*, *dtype=<class 'numpy.float64'>*)

fit(*self*, *X*, *y=None*)
Fit the OrdinalEncoder to X.

Parameters

X [array-like, shape [n_samples, n_features]] The data to determine the categories of each feature.

Returns

self

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*)
Convert the data back to the original representation.

Parameters

X [array-like or sparse matrix, shape [n_samples, n_encoded_features]] The transformed data.

Returns

X_tr [array-like, shape [n_samples, n_features]] Inverse transformed array.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform X to ordinal codes.

Parameters

X [array-like, shape [n_samples, n_features]] The data to encode.

Returns

X_out [sparse matrix or a 2-d array] Transformed input.

6.34.13 sklearn.preprocessing.PolynomialFeatures

class sklearn.preprocessing.**PolynomialFeatures**(*degree=2*, *interaction_only=False*, *include_bias=True*, *order='C'*)
Generate polynomial and interaction features.

Generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, if an input sample is two dimensional and of the form [a, b], the degree-2 polynomial features are [1, a, b, a^2, ab, b^2].

Parameters

degree [integer] The degree of the polynomial features. Default = 2.

interaction_only [boolean, default = False] If true, only interaction features are produced: features that are products of at most `degree` *distinct* input features (so not `x[1] ** 2`, `x[0] * x[2] ** 3`, etc.).

include_bias [boolean] If True (default), then include a bias column, the feature in which all polynomial powers are zero (i.e. a column of ones - acts as an intercept term in a linear model).

order [str in {'C', 'F'}, default 'C'] Order of output array in the dense case. 'F' order is faster to compute, but may slow down subsequent estimators.

New in version 0.21.

Attributes

powers_ [array, shape (n_output_features, n_input_features)] powers_[i, j] is the exponent of the jth input in the ith output.

n_input_features_ [int] The total number of input features.

n_output_features_ [int] The total number of polynomial output features. The number of output features is computed by iterating over all suitably sized combinations of input features.

Notes

Be aware that the number of features in the output array scales polynomially in the number of features of the input array, and exponentially in the degree. High degrees can cause overfitting.

See *examples/linear_model/plot_polynomial_interpolation.py*

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> poly = PolynomialFeatures(2)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
>>> poly = PolynomialFeatures(interaction_only=True)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  0.],
       [ 1.,  2.,  3.,  6.],
       [ 1.,  4.,  5., 20.]])
```

Methods

fit(self, X[, y])	Compute number of output features.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_feature_names(self[, input_features])	Return feature names for output features
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Transform data to polynomial features

__init__ (*self*, *degree=2*, *interaction_only=False*, *include_bias=True*, *order='C'*)

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

Compute number of output features.

Parameters

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

Returns

self [instance]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_feature_names(*self*, *input_features=None*)
Return feature names for output features

Parameters

input_features [list of string, length n_features, optional] String names for input features if available. By default, "x0", "x1", ... "xn_features" is used.

Returns

output_feature_names [list of string, length n_output_features]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Transform data to polynomial features

Parameters

X [array-like or CSR/CSC sparse matrix, shape [n_samples, n_features]] The data to transform, row by row.

Prefer CSR over CSC for sparse input (for speed), but CSC is required if the degree is 4 or higher. If the degree is less than 4 and the input format is CSC, it will be converted to CSR, have its polynomial features generated, then converted back to CSC.

If the degree is 2 or 3, the method described in "Leveraging Sparsity to Speed Up Polynomial Feature Expansions of CSR Matrices Using K-Simplex Numbers" by Andrew Nystrom and John Hughes is used, which is much faster than the method used on CSC input. For this reason, a CSC input will be converted to CSR, and the output will be converted back to CSC prior to being returned, hence the preference of CSR.

Returns

XP [np.ndarray or CSR/CSC sparse matrix, shape [n_samples, NP]] The matrix of features, where NP is the number of polynomial features generated from the combination of inputs.

Examples using `sklearn.preprocessing.PolynomialFeatures`

- *Polynomial interpolation*
- *Robust linear estimator fitting*
- *Underfitting vs. Overfitting*

6.34.14 `sklearn.preprocessing`.PowerTransformer

class `sklearn.preprocessing.`**`PowerTransformer`**(*method='yeo-johnson'*, *standardize=True*, *copy=True*)

Apply a power transform featurewise to make data more Gaussian-like.

Power transforms are a family of parametric, monotonic transformations that are applied to make data more Gaussian-like. This is useful for modeling issues related to heteroscedasticity (non-constant variance), or other situations where normality is desired.

Currently, PowerTransformer supports the Box-Cox transform and the Yeo-Johnson transform. The optimal parameter for stabilizing variance and minimizing skewness is estimated through maximum likelihood.

Box-Cox requires input data to be strictly positive, while Yeo-Johnson supports both positive or negative data.

By default, zero-mean, unit-variance normalization is applied to the transformed data.

Read more in the *User Guide*.

Parameters

method [str, (default='yeo-johnson')] The power transform method. Available methods are:

- 'yeo-johnson' *[Rf3e1504535de-1]*, works with positive and negative values
- 'box-cox' *[Rf3e1504535de-2]*, only works with strictly positive values

standardize [boolean, default=True] Set to True to apply zero-mean, unit-variance normalization to the transformed output.

copy [boolean, optional, default=True] Set to False to perform inplace computation during transformation.

Attributes

lambdas_ [array of float, shape (n_features,)] The parameters of the power transformation for the selected features.

See also:

power_transform Equivalent function without the estimator API.

QuantileTransformer Maps data to a standard normal distribution with the parameter `output_distribution='normal'`.

Notes

NaNs are treated as missing values: disregarded in `fit`, and maintained in `transform`.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

References

[Rf3e1504535de-1], *[Rf3e1504535de-2]*

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import PowerTransformer
>>> pt = PowerTransformer()
>>> data = [[1, 2], [3, 2], [4, 5]]
>>> print(pt.fit(data))
PowerTransformer(copy=True, method='yeo-johnson', standardize=True)
>>> print(pt.lambdas_)
[ 1.386... -3.100...]
>>> print(pt.transform(data))
[[-1.316... -0.707...]
 [ 0.209... -0.707...]
 [ 1.106...  1.414...]]
```

Methods

fit(self, X[, y])	Estimate the optimal parameter lambda for each feature.
fit_transform(self, X[, y])	
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Apply the inverse power transformation using the fitted lambdas.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Apply the power transform to each feature using the fitted lambdas.

__init__(*self*, *method='yeo-johnson'*, *standardize=True*, *copy=True*)

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

Estimate the optimal parameter lambda for each feature.

The optimal lambda parameter for minimizing skewness is estimated on each feature independently using maximum likelihood.

Parameters

X [array-like, shape (n_samples, n_features)] The data used to estimate the optimal transformation parameters.

y [Ignored]

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*)
Apply the inverse power transformation using the fitted lambdas.

The inverse of the Box-Cox transformation is given by:

```
if lambda == 0:
    X = exp(X_trans)
else:
    X = (X_trans * lambda + 1) ** (1 / lambda)
```

The inverse of the Yeo-Johnson transformation is given by:

```
if X >= 0 and lambda == 0:
    X = exp(X_trans) - 1
elif X >= 0 and lambda != 0:
    X = (X_trans * lambda + 1) ** (1 / lambda) - 1
elif X < 0 and lambda != 2:
    X = 1 - (-(2 - lambda) * X_trans + 1) ** (1 / (2 - lambda))
elif X < 0 and lambda == 2:
    X = 1 - exp(-X_trans)
```

Parameters

X [array-like, shape (n_samples, n_features)] The transformed data.

Returns

X [array-like, shape (n_samples, n_features)] The original data

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Apply the power transform to each feature using the fitted lambdas.

Parameters

X [array-like, shape (n_samples, n_features)] The data to be transformed using a power transformation.

Returns

X_trans [array-like, shape (n_samples, n_features)] The transformed data.

Examples using `sklearn.preprocessing.PowerTransformer`

- *Map data to a normal distribution*
- *Compare the effect of different scalers on data with outliers*

6.34.15 `sklearn.preprocessing`.QuantileTransformer

class `sklearn.preprocessing.`**`QuantileTransformer`**(*n_quantiles=1000*, *output_distribution='uniform'*, *ignore_implicit_zeros=False*, *subsample=100000*, *random_state=None*, *copy=True*)

Transform features using quantiles information.

This method transforms the features to follow a uniform or a normal distribution. Therefore, for a given feature, this transformation tends to spread out the most frequent values. It also reduces the impact of (marginal) outliers: this is therefore a robust preprocessing scheme.

The transformation is applied on each feature independently. First an estimate of the cumulative distribution function of a feature is used to map the original values to a uniform distribution. The obtained values are then mapped to the desired output distribution using the associated quantile function. Features values of new/unseen data that fall below or above the fitted range will be mapped to the bounds of the output distribution. Note that this transform is non-linear. It may distort linear correlations between variables measured at the same scale but renders variables measured at different scales more directly comparable.

Read more in the *User Guide*.

Parameters

n_quantiles [int, optional (default=1000 or n_samples)] Number of quantiles to be computed. It corresponds to the number of landmarks used to discretize the cumulative distribution function. If n_quantiles is larger than the number of samples, n_quantiles is set to the number of samples as a larger number of quantiles does not give a better approximation of the cumulative distribution function estimator.

output_distribution [str, optional (default='uniform')] Marginal distribution for the transformed data. The choices are 'uniform' (default) or 'normal'.

ignore_implicit_zeros [bool, optional (default=False)] Only applies to sparse matrices. If True, the sparse entries of the matrix are discarded to compute the quantile statistics. If False, these entries are treated as zeros.

subsample [int, optional (default=1e5)] Maximum number of samples used to estimate the quantiles for computational efficiency. Note that the subsampling procedure may differ for value-identical sparse and dense matrices.

random_state [int, RandomState instance or None, optional (default=None)] 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. Note that this is used by subsampling and smoothing noise.

copy [boolean, optional, (default=True)] Set to False to perform inplace transformation and avoid a copy (if the input is already a numpy array).

Attributes

n_quantiles_ [integer] The actual number of quantiles used to discretize the cumulative distribution function.

quantiles_ [ndarray, shape (n_quantiles, n_features)] The values corresponding the quantiles of reference.

references_ [ndarray, shape(n_quantiles,)] Quantiles of references.

See also:

quantile_transform Equivalent function without the estimator API.

PowerTransformer Perform mapping to a normal distribution using a power transform.

StandardScaler Perform standardization that is faster, but less robust to outliers.

RobustScaler Perform robust standardization that removes the influence of outliers but does not put outliers and inliers on the same scale.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import QuantileTransformer
>>> rng = np.random.RandomState(0)
>>> X = np.sort(rng.normal(loc=0.5, scale=0.25, size=(25, 1)), axis=0)
>>> qt = QuantileTransformer(n_quantiles=10, random_state=0)
>>> qt.fit_transform(X)
array([...])
```

Methods

fit(self, X[, y])	Compute the quantiles used for transforming.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Back-projection to the original space.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Feature-wise transformation of the data.

__init__(*self*, *n_quantiles=1000*, *output_distribution='uniform'*, *ignore_implicit_zeros=False*, *subsample=100000*, *random_state=None*, *copy=True*)

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

Compute the quantiles used for transforming.

Parameters

X [ndarray or sparse matrix, shape (n_samples, n_features)] The data used to scale along the features axis. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`. Additionally, the sparse matrix needs to be nonnegative if `ignore_implicit_zeros` is False.

Returns

self [object]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*)

Back-projection to the original space.

Parameters

X [ndarray or sparse matrix, shape (n_samples, n_features)] The data used to scale along the features axis. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`. Additionally, the sparse matrix needs to be nonnegative if `ignore_implicit_zeros` is False.

Returns

Xt [ndarray or sparse matrix, shape (n_samples, n_features)] The projected data.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)

Feature-wise transformation of the data.

Parameters

X [ndarray or sparse matrix, shape (n_samples, n_features)] The data used to scale along the features axis. If a sparse matrix is provided, it will be converted into a sparse `csc_matrix`. Additionally, the sparse matrix needs to be nonnegative if `ignore_implicit_zeros` is False.

Returns

Xt [ndarray or sparse matrix, shape (n_samples, n_features)] The projected data.

Examples using `sklearn.preprocessing.QuantileTransformer`

- *Effect of transforming the targets in regression model*
- *Map data to a normal distribution*
- *Compare the effect of different scalers on data with outliers*

6.34.16 `sklearn.preprocessing`.RobustScaler

class `sklearn.preprocessing.`**RobustScaler**(*with_centering=True*, *with_scaling=True*, *quantile_range=(25.0, 75.0)*, *copy=True*)

Scale features using statistics that are robust to outliers.

This Scaler removes the median and scales the data according to the quantile range (defaults to IQR: Interquartile Range). The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile).

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Median and interquartile range are then stored to be used on later data using the `transform` method.

Standardization of a dataset is a common requirement for many machine learning estimators. Typically this is done by removing the mean and scaling to unit variance. However, outliers can often influence the sample mean / variance in a negative way. In such cases, the median and the interquartile range often give better results.

New in version 0.17.

Read more in the *User Guide*.

Parameters

with_centering [boolean, True by default] If True, center the data before scaling. This will cause `transform` to raise an exception when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory.

with_scaling [boolean, True by default] If True, scale the data to interquartile range.

quantile_range [tuple (q_min, q_max), 0.0 < q_min < q_max < 100.0] Default: (25.0, 75.0) = (1st quantile, 3rd quantile) = IQR Quantile range used to calculate `scale_`.

New in version 0.18.

copy [boolean, optional, default is True] If False, try to avoid a copy and do inplace scaling instead. This is not guaranteed to always work inplace; e.g. if the data is not a NumPy array or scipy.sparse CSR matrix, a copy may still be returned.

Attributes

center_ [array of floats] The median value for each feature in the training set.

scale_ [array of floats] The (scaled) interquartile range for each feature in the training set.

New in version 0.17: *scale_* attribute.

See also:

robust_scale Equivalent function without the estimator API.

sklearn.decomposition.PCA Further removes the linear correlation across features with 'whiten=True'.

Notes

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

https://en.wikipedia.org/wiki/Median https://en.wikipedia.org/wiki/Interquartile_range

Examples

```
>>> from sklearn.preprocessing import RobustScaler
>>> X = [[ 1., -2.,  2.],
...      [ -2.,  1.,  3.],
...      [ 4.,  1., -2.]]
>>> transformer = RobustScaler().fit(X)
>>> transformer
RobustScaler(copy=True, quantile_range=(25.0, 75.0), with_centering=True,
       with_scaling=True)
>>> transformer.transform(X)
array([[ 0. , -2. ,  0. ],
       [-1. ,  0. ,  0.4],
       [ 1. ,  0. , -1.6]])
```

Methods

fit(self, X[, y])	Compute the median and quantiles to be used for scaling.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X)	Scale back the data to the original representation
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Center and scale the data.

__init__ (*self*, *with_centering=True*, *with_scaling=True*, *quantile_range=(25.0, 75.0)*, *copy=True*)

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

Compute the median and quantiles to be used for scaling.

Parameters

X [array-like, shape [n_samples, n_features]] The data used to compute the median and quantiles used for later scaling along the features axis.

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*)
Scale back the data to the original representation

Parameters

X [array-like] The data used to scale along the specified axis.

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*)
Center and scale the data.

Parameters

X [{array-like, sparse matrix}] The data used to scale along the specified axis.

Examples using `sklearn.preprocessing.RobustScaler`

- *Compare the effect of different scalers on data with outliers*

6.34.17 `sklearn.preprocessing`.StandardScaler

class `sklearn.preprocessing.`**StandardScaler**(*copy=True*, *with_mean=True*, *with_std=True*)
Standardize features by removing the mean and scaling to unit variance

The standard score of a sample *x* is calculated as:

z = (x - u) / s

where `u` is the mean of the training samples or zero if `with_mean=False`, and `s` is the standard deviation of the training samples or one if `with_std=False`.

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using the *transform* method.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance).

For instance many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the L1 and L2 regularizers of linear models) assume that all features are centered around 0 and have variance in the same order. If a feature has a variance that is orders of magnitude larger that others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

This scaler can also be applied to sparse CSR or CSC matrices by passing `with_mean=False` to avoid breaking the sparsity structure of the data.

Read more in the *User Guide*.

Parameters

copy [boolean, optional, default True] If False, try to avoid a copy and do inplace scaling instead. This is not guaranteed to always work inplace; e.g. if the data is not a NumPy array or scipy.sparse CSR matrix, a copy may still be returned.

with_mean [boolean, True by default] If True, center the data before scaling. This does not work (and will raise an exception) when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory.

with_std [boolean, True by default] If True, scale the data to unit variance (or equivalently, unit standard deviation).

Attributes

scale_ [ndarray or None, shape (n_features,)] Per feature relative scaling of the data. This is calculated using `np.sqrt(var_)`. Equal to `None` when `with_std=False`.

New in version 0.17: *scale_*

mean_ [ndarray or None, shape (n_features,)] The mean value for each feature in the training set. Equal to `None` when `with_mean=False`.

var_ [ndarray or None, shape (n_features,)] The variance for each feature in the training set. Used to compute `scale_`. Equal to `None` when `with_std=False`.

n_samples_seen_ [int or array, shape (n_features,)] The number of samples processed by the estimator for each feature. If there are not missing samples, the `n_samples_seen` will be an integer, otherwise it will be an array. Will be reset on new calls to fit, but increments across `partial_fit` calls.

See also:

scale Equivalent function without the estimator API.

sklearn.decomposition.PCA Further removes the linear correlation across features with 'whiten=True'.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

We use a biased estimator for the standard deviation, equivalent to `numpy.std(x, ddof=0)`. Note that the choice of `ddof` is unlikely to affect model performance.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> from sklearn.preprocessing import StandardScaler
>>> data = [[0, 0], [0, 0], [1, 1], [1, 1]]
>>> scaler = StandardScaler()
>>> print(scaler.fit(data))
StandardScaler(copy=True, with_mean=True, with_std=True)
>>> print(scaler.mean_)
[0.5 0.5]
>>> print(scaler.transform(data))
[[-1. -1.]
 [-1. -1.]
 [ 1.  1.]
 [ 1.  1.]]
>>> print(scaler.transform([[2, 2]]))
[[3. 3.]]
```

Methods

fit(self, X[, y])	Compute the mean and std to be used for later scaling.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, X[, copy])	Scale back the data to the original representation
partial_fit(self, X[, y])	Online computation of mean and std on X for later scaling.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X[, copy])	Perform standardization by centering and scaling

__init__ (*self*, *copy=True*, *with_mean=True*, *with_std=True*)

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

Compute the mean and std to be used for later scaling.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y Ignored

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

inverse_transform(*self*, *X*, *copy=None*)
Scale back the data to the original representation

Parameters

X [array-like, shape [n_samples, n_features]] The data used to scale along the features axis.

copy [bool, optional (default: None)] Copy the input X or not.

Returns

X_tr [array-like, shape [n_samples, n_features]] Transformed array.

partial_fit(*self*, *X*, *y=None*)
Online computation of mean and std on X for later scaling. All of X is processed as a single batch. This is intended for cases when `fit` is not feasible due to very large number of *n_samples* or because X is read from a continuous stream.

The algorithm for incremental mean and std is given in Equation 1.5a,b in Chan, Tony F., Gene H. Golub, and Randall J. LeVeque. "Algorithms for computing the sample variance: Analysis and recommendations." The American Statistician 37.3 (1983): 242-247:

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y Ignored

set_params(*self*, ***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.

Returns

self

transform(*self*, *X*, *copy=None*)
Perform standardization by centering and scaling

Parameters

X [array-like, shape [n_samples, n_features]] The data used to scale along the features axis.

copy [bool, optional (default: None)] Copy the input X or not.

Examples using `sklearn.preprocessing.StandardScaler`

- *Prediction Latency*
- *Classifier comparison*
- *Demo of DBSCAN clustering algorithm*
- *Comparing different hierarchical linkage methods on toy datasets*
- *Comparing different clustering algorithms on toy datasets*
- *Column Transformer with Mixed Types*
- *MNIST classfification using multinomial logistic + L1*
- *L1 Penalty and Sparsity in Logistic Regression*
- *Comparing Nearest Neighbors with and without Neighborhood Components Analysis*
- *Dimensionality Reduction with Neighborhood Components Analysis*
- *Varying regularization in Multi-layer Perceptron*
- *Importance of Feature Scaling*
- *Feature discretization*
- *Compare the effect of different scalers on data with outliers*
- *SVM-Anova: SVM with univariate feature selection*
- *RBF SVM parameters*

`preprocessing.add_dummy_feature`(X[, value])	Augment dataset with an additional dummy feature.
`preprocessing.binarize`(X[, threshold, copy])	Boolean thresholding of array-like or scipy.sparse matrix
`preprocessing.label_binarize`(y, classes[, ...])	Binarize labels in a one-vs-all fashion
`preprocessing.maxabs_scale`(X[, axis, copy])	Scale each feature to the [-1, 1] range without breaking the sparsity.
`preprocessing.minmax_scale`(X[, ...])	Transforms features by scaling each feature to a given range.
`preprocessing.normalize`(X[, norm, axis, ...])	Scale input vectors individually to unit norm (vector length).
`preprocessing.quantile_transform`(X[, axis, ...])	Transform features using quantiles information.
`preprocessing.robust_scale`(X[, axis, ...])	Standardize a dataset along any axis
`preprocessing.scale`(X[, axis, with_mean, ...])	Standardize a dataset along any axis
`preprocessing.power_transform`(X[, method, ...])	Power transforms are a family of parametric, monotonic transformations that are applied to make data more Gaussian-like.

6.34.18 sklearn.preprocessing.add_dummy_feature

sklearn.preprocessing.**add_dummy_feature**(*X*, *value=1.0*)

Augment dataset with an additional dummy feature.

This is useful for fitting an intercept term with implementations which cannot otherwise fit it directly.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] Data.

value [float] Value to use for the dummy feature.

Returns

X [{array, sparse matrix}, shape [n_samples, n_features + 1]] Same data with dummy feature added as first column.

Examples

```
>>> from sklearn.preprocessing import add_dummy_feature
>>> add_dummy_feature([[0, 1], [1, 0]])
array([[1., 0., 1.],
       [1., 1., 0.]])
```

6.34.19 sklearn.preprocessing.binarize

sklearn.preprocessing.**binarize**(*X*, *threshold=0.0*, *copy=True*)

Boolean thresholding of array-like or scipy.sparse matrix

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data to binarize, element by element. scipy.sparse matrices should be in CSR or CSC format to avoid an un-necessary copy.

threshold [float, optional (0.0 by default)] Feature values below or equal to this are replaced by 0, above it by 1. Threshold may not be less than 0 for operations on sparse matrices.

copy [boolean, optional, default True] set to False to perform inplace binarization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR / CSC matrix and if axis is 1).

See also:

Binarizer Performs binarization using the `Transformer` API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

6.34.20 sklearn.preprocessing.label_binarize

sklearn.preprocessing.**label_binarize**(*y*, *classes*, *neg_label=0*, *pos_label=1*, *sparse_output=False*)

Binarize labels in a one-vs-all fashion

Several regression and binary classification algorithms are available in scikit-learn. A simple way to extend these algorithms to the multi-class classification case is to use the so-called one-vs-all scheme.

This function makes it possible to compute this transformation for a fixed set of class labels known ahead of time.

Parameters

y [array-like] Sequence of integer labels or multilabel data to encode.

classes [array-like of shape [n_classes]] Uniquely holds the label for each class.

neg_label [int (default: 0)] Value with which negative labels must be encoded.

pos_label [int (default: 1)] Value with which positive labels must be encoded.

sparse_output [boolean (default: False),] Set to true if output binary array is desired in CSR sparse format

Returns

Y [numpy array or CSR matrix of shape [n_samples, n_classes]] Shape will be [n_samples, 1] for binary problems.

See also:

LabelBinarizer class used to wrap the functionality of label_binarize and allow for fitting to classes independently of the transform operation

Examples

```
>>> from sklearn.preprocessing import label_binarize
>>> label_binarize([1, 6], classes=[1, 2, 4, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])
```

The class ordering is preserved:

```
>>> label_binarize([1, 6], classes=[1, 6, 4, 2])
array([[1, 0, 0, 0],
       [0, 1, 0, 0]])
```

Binary targets transform to a column vector

```
>>> label_binarize(['yes', 'no', 'no', 'yes'], classes=['no', 'yes'])
array([[1],
       [0],
       [0],
       [1]])
```

Examples using `sklearn.preprocessing.label_binarize`

- *Receiver Operating Characteristic (ROC)*
- *Precision-Recall*

6.34.21 sklearn.preprocessing.maxabs_scale

sklearn.preprocessing.**maxabs_scale**(*X*, *axis=0*, *copy=True*)

Scale each feature to the [-1, 1] range without breaking the sparsity.

This estimator scales each feature individually such that the maximal absolute value of each feature in the training set will be 1.0.

This scaler can also be applied to sparse CSR or CSC matrices.

Parameters

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

axis [int (0 by default)] axis used to scale along. If 0, independently scale each feature, otherwise (if 1) scale each sample.

copy [boolean, optional, default is True] Set to False to perform inplace scaling and avoid a copy (if the input is already a numpy array).

See also:

MaxAbsScaler Performs scaling to the [-1, 1] range using the‘‘Transformer‘‘ API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

Notes

NaNs are treated as missing values: disregarded to compute the statistics, and maintained during the data transformation.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

6.34.22 sklearn.preprocessing.minmax_scale

sklearn.preprocessing.**minmax_scale**(*X*, *feature_range=(0, 1)*, *axis=0*, *copy=True*)

Transforms features by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training set, i.e. between zero and one.

The transformation is given by (when `axis=0`):

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where min, max = feature_range.

The transformation is calculated as (when `axis=0`):

```
X_scaled = scale * X + min - X.min(axis=0) * scale
where scale = (max - min) / (X.max(axis=0) - X.min(axis=0))
```

This transformation is often used as an alternative to zero mean, unit variance scaling.

Read more in the *User Guide*.

New in version 0.17: *minmax_scale* function interface to *sklearn.preprocessing.MinMaxScaler*.

Parameters

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

feature_range [tuple (min, max), default=(0, 1)] Desired range of transformed data.

axis [int (0 by default)] axis used to scale along. If 0, independently scale each feature, otherwise (if 1) scale each sample.

copy [boolean, optional, default is True] Set to False to perform inplace scaling and avoid a copy (if the input is already a numpy array).

See also:

MinMaxScaler Performs scaling to a given range using the''Transformer'' API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

Notes

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples using `sklearn.preprocessing.minmax_scale`

- *Compare the effect of different scalers on data with outliers*

6.34.23 `sklearn.preprocessing`.normalize

`sklearn.preprocessing.`**`normalize`**(*X*, *norm='l2'*, *axis=1*, *copy=True*, *return_norm=False*)

Scale input vectors individually to unit norm (vector length).

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data to normalize, element by element. scipy.sparse matrices should be in CSR format to avoid an un-necessary copy.

norm ['l1', 'l2', or 'max', optional ('l2' by default)] The norm to use to normalize each non zero sample (or each non-zero feature if axis is 0).

axis [0 or 1, optional (1 by default)] axis used to normalize the data along. If 1, independently normalize each sample, otherwise (if 0) normalize each feature.

copy [boolean, optional, default True] set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix and if axis is 1).

return_norm [boolean, default False] whether to return the computed norms

Returns

X [{array-like, sparse matrix}, shape [n_samples, n_features]] Normalized input X.

norms [array, shape [n_samples] if axis=1 else [n_features]] An array of norms along given axis for X. When X is sparse, a NotImplementedError will be raised for norm 'l1' or 'l2'.

See also:

Normalizer Performs normalization using the `Transformer` API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

Notes

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

6.34.24 sklearn.preprocessing.quantile_transform

sklearn.preprocessing.**quantile_transform**(*X*, *axis=0*, *n_quantiles=1000*, *output_distribution='uniform'*, *ignore_implicit_zeros=False*, *subsample=100000*, *random_state=None*, *copy='warn'*)

Transform features using quantiles information.

This method transforms the features to follow a uniform or a normal distribution. Therefore, for a given feature, this transformation tends to spread out the most frequent values. It also reduces the impact of (marginal) outliers: this is therefore a robust preprocessing scheme.

The transformation is applied on each feature independently. First an estimate of the cumulative distribution function of a feature is used to map the original values to a uniform distribution. The obtained values are then mapped to the desired output distribution using the associated quantile function. Features values of new/unseen data that fall below or above the fitted range will be mapped to the bounds of the output distribution. Note that this transform is non-linear. It may distort linear correlations between variables measured at the same scale but renders variables measured at different scales more directly comparable.

Read more in the *User Guide*.

Parameters

X [array-like, sparse matrix] The data to transform.

axis [int, (default=0)] Axis used to compute the means and standard deviations along. If 0, transform each feature, otherwise (if 1) transform each sample.

n_quantiles [int, optional (default=1000 or n_samples)] Number of quantiles to be computed. It corresponds to the number of landmarks used to discretize the cumulative distribution function. If n_quantiles is larger than the number of samples, n_quantiles is set to the number of samples as a larger number of quantiles does not give a better approximation of the cumulative distribution function estimator.

output_distribution [str, optional (default='uniform')] Marginal distribution for the transformed data. The choices are 'uniform' (default) or 'normal'.

ignore_implicit_zeros [bool, optional (default=False)] Only applies to sparse matrices. If True, the sparse entries of the matrix are discarded to compute the quantile statistics. If False, these entries are treated as zeros.

subsample [int, optional (default=1e5)] Maximum number of samples used to estimate the quantiles for computational efficiency. Note that the subsampling procedure may differ for value-identical sparse and dense matrices.

random_state [int, RandomState instance or None, optional (default=None)] 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. Note that this is used by subsampling and smoothing noise.

copy [boolean, optional, (default="warn")] Set to False to perform inplace transformation and avoid a copy (if the input is already a numpy array). If True, a copy of *X* is transformed, leaving the original *X* unchanged

Deprecated since version 0.21: The default value of parameter `copy` will be changed from False to True in 0.23. The current default of False is being changed to make it more consistent with the default `copy` values of other functions in `sklearn.preprocessing.data`. Furthermore, the current default of False may have unexpected side effects by modifying the value of *X* inplace

Returns

Xt [ndarray or sparse matrix, shape (n_samples, n_features)] The transformed data.

See also:

QuantileTransformer Performs quantile-based scaling using the `Transformer` API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

power_transform Maps data to a normal distribution using a power transformation.

scale Performs standardization that is faster, but less robust to outliers.

robust_scale Performs robust standardization that removes the influence of outliers but does not put outliers and inliers on the same scale.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import quantile_transform
>>> rng = np.random.RandomState(0)
>>> X = np.sort(rng.normal(loc=0.5, scale=0.25, size=(25, 1)), axis=0)
>>> quantile_transform(X, n_quantiles=10, random_state=0, copy=True)
...
array([...])
```

Examples using `sklearn.preprocessing.quantile_transform`

- *Effect of transforming the targets in regression model*

6.34.25 `sklearn.preprocessing.robust_scale`

`sklearn.preprocessing.`**`robust_scale`**(*X*, *axis=0*, *with_centering=True*, *with_scaling=True*, *quantile_range=(25.0, 75.0)*, *copy=True*)

Standardize a dataset along any axis

Center to the median and component wise scale according to the interquartile range.

Read more in the *User Guide*.

Parameters

X [array-like] The data to center and scale.

axis [int (0 by default)] axis used to compute the medians and IQR along. If 0, independently scale each feature, otherwise (if 1) scale each sample.

with_centering [boolean, True by default] If True, center the data before scaling.

with_scaling [boolean, True by default] If True, scale the data to unit variance (or equivalently, unit standard deviation).

quantile_range [tuple (q_min, q_max), 0.0 < q_min < q_max < 100.0] Default: (25.0, 75.0) = (1st quantile, 3rd quantile) = IQR Quantile range used to calculate `scale_`.

New in version 0.18.

copy [boolean, optional, default is True] set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSR matrix and if axis is 1).

See also:

RobustScaler Performs centering and scaling using the `Transformer` API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

Notes

This implementation will refuse to center scipy.sparse matrices since it would make them non-sparse and would potentially crash the program with memory exhaustion problems.

Instead the caller is expected to either set explicitly `with_centering=False` (in that case, only variance scaling will be performed on the features of the CSR matrix) or to call `X.toarray()` if he/she expects the materialized dense array to fit in memory.

To avoid memory copy the caller should pass a CSR matrix.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

6.34.26 sklearn.preprocessing.scale

`sklearn.preprocessing.`**`scale`**(*X*, *axis=0*, *with_mean=True*, *with_std=True*, *copy=True*)

Standardize a dataset along any axis

Center to the mean and component wise scale to unit variance.

Read more in the *User Guide*.

Parameters

X [{array-like, sparse matrix}] The data to center and scale.

axis [int (0 by default)] axis used to compute the means and standard deviations along. If 0, independently standardize each feature, otherwise (if 1) standardize each sample.

with_mean [boolean, True by default] If True, center the data before scaling.

with_std [boolean, True by default] If True, scale the data to unit variance (or equivalently, unit standard deviation).

copy [boolean, optional, default True] set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSC matrix and if axis is 1).

See also:

StandardScaler Performs scaling to unit variance using the‘‘Transformer‘‘ API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

Notes

This implementation will refuse to center scipy.sparse matrices since it would make them non-sparse and would potentially crash the program with memory exhaustion problems.

Instead the caller is expected to either set explicitly `with_mean=False` (in that case, only variance scaling will be performed on the features of the CSC matrix) or to call `X.toarray()` if he/she expects the materialized dense array to fit in memory.

To avoid memory copy the caller should pass a CSC matrix.

NaNs are treated as missing values: disregarded to compute the statistics, and maintained during the data transformation.

We use a biased estimator for the standard deviation, equivalent to `numpy.std(x, ddof=0)`. Note that the choice of `ddof` is unlikely to affect model performance.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples using `sklearn.preprocessing.scale`

- *A demo of K-Means clustering on the handwritten digits data*

6.34.27 `sklearn.preprocessing`.power_transform

`sklearn.preprocessing.`**`power_transform`**(*X*, *method='warn'*, *standardize=True*, *copy=True*)

Power transforms are a family of parametric, monotonic transformations that are applied to make data more Gaussian-like. This is useful for modeling issues related to heteroscedasticity (non-constant variance), or other situations where normality is desired.

Currently, power_transform supports the Box-Cox transform and the Yeo-Johnson transform. The optimal parameter for stabilizing variance and minimizing skewness is estimated through maximum likelihood.

Box-Cox requires input data to be strictly positive, while Yeo-Johnson supports both positive or negative data.

By default, zero-mean, unit-variance normalization is applied to the transformed data.

Read more in the *User Guide*.

Parameters

X [array-like, shape (n_samples, n_features)] The data to be transformed using a power transformation.

method [str] The power transform method. Available methods are:

- ‘yeo-johnson’ *[1]*, works with positive and negative values
- ‘box-cox’ *[2]*, only works with strictly positive values

The default method will be changed from ‘box-cox’ to ‘yeo-johnson’ in version 0.23. To suppress the FutureWarning, explicitly set the parameter.

standardize [boolean, default=True] Set to True to apply zero-mean, unit-variance normalization to the transformed output.

copy [boolean, optional, default=True] Set to False to perform inplace computation during transformation.

Returns

X_trans [array-like, shape (n_samples, n_features)] The transformed data.

See also:

PowerTransformer Equivalent transformation with the `Transformer` API (e.g. as part of a preprocessing *sklearn.pipeline.Pipeline*).

quantile_transform Maps data to a standard normal distribution with the parameter `output_distribution='normal'`.

Notes

NaNs are treated as missing values: disregarded in `fit`, and maintained in `transform`.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

References

[1], *[2]*

Examples

```
>>> import numpy as np
>>> from sklearn.preprocessing import power_transform
>>> data = [[1, 2], [3, 2], [4, 5]]
>>> print(power_transform(data, method='box-cox'))
[[-1.332... -0.707...]
 [ 0.256... -0.707...]
 [ 1.076...  1.414...]]
```

6.35 `sklearn.random_projection`: Random projection

Random Projection transformers

Random Projections are a simple and computationally efficient way to reduce the dimensionality of the data by trading a controlled amount of accuracy (as additional variance) for faster processing times and smaller model sizes.

The dimensions and distribution of Random Projections matrices are controlled so as to preserve the pairwise distances between any two samples of the dataset.

The main theoretical result behind the efficiency of random projection is the Johnson-Lindenstrauss lemma (quoting Wikipedia):

In mathematics, the Johnson-Lindenstrauss lemma is a result concerning low-distortion embeddings of points from high-dimensional into low-dimensional Euclidean space. The lemma states that a small set of points in a high-dimensional space can be embedded into a space of much lower dimension in such a way that distances between the points are nearly preserved. The map used for the embedding is at least Lipschitz, and can even be taken to be an orthogonal projection.

User guide: See the *Random Projection* section for further details.

random_projection.GaussianRandomProjection([...])	Reduce dimensionality through Gaussian random projection
random_projection.SparseRandomProjection([...])	Reduce dimensionality through sparse random projection

6.35.1 sklearn.random_projection.GaussianRandomProjection

class sklearn.random_projection.**GaussianRandomProjection**(*n_components='auto'*, *eps=0.1*, *random_state=None*)

Reduce dimensionality through Gaussian random projection

The components of the random matrix are drawn from N(0, 1 / n_components).

Read more in the *User Guide*.

Parameters

n_components [int or 'auto', optional (default = 'auto')] Dimensionality of the target projection space.

n_components can be automatically adjusted according to the number of samples in the dataset and the bound given by the Johnson-Lindenstrauss lemma. In that case the quality of the embedding is controlled by the `eps` parameter.

It should be noted that Johnson-Lindenstrauss lemma can yield very conservative estimated of the required number of components as it makes no assumption on the structure of the dataset.

eps [strictly positive float, optional (default=0.1)] Parameter to control the quality of the embedding according to the Johnson-Lindenstrauss lemma when n_components is set to 'auto'.

Smaller values lead to better embedding and higher number of dimensions (n_components) in the target projection space.

random_state [int, RandomState instance or None, optional (default=None)] Control the pseudo random number generator used to generate the matrix at fit time. 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`.

Attributes

n_component_ [int] Concrete number of components computed when n_components="auto".

components_ [numpy array of shape [n_components, n_features]] Random matrix used for the projection.

See also:

SparseRandomProjection

Examples

```
>>> import numpy as np
>>> from sklearn.random_projection import GaussianRandomProjection
>>> rng = np.random.RandomState(42)
>>> X = rng.rand(100, 10000)
>>> transformer = GaussianRandomProjection(random_state=rng)
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
```

Methods

fit(self, X[, y])	Generate a sparse random projection matrix
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Project the data by using matrix product with the random matrix

__init__ (*self*, *n_components='auto'*, *eps=0.1*, *random_state=None*)

fit (*self*, *X*, *y=None*)
Generate a sparse random projection matrix

Parameters

X [numpy array or scipy.sparse of shape [n_samples, n_features]] Training set: only the shape is used to find optimal random matrix dimensions based on the theory referenced in the afore mentioned papers.

y Ignored

Returns

self

fit_transform (*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Project the data by using matrix product with the random matrix

Parameters

X [numpy array or scipy.sparse of shape [n_samples, n_features]] The input data to project into a smaller dimensional space.

Returns

X_new [numpy array or scipy sparse of shape [n_samples, n_components]] Projected array.

6.35.2 sklearn.random_projection.SparseRandomProjection

class sklearn.random_projection.**SparseRandomProjection**(*n_components='auto'*, *density='auto'*, *eps=0.1*, *dense_output=False*, *random_state=None*)
Reduce dimensionality through sparse random projection

Sparse random matrix is an alternative to dense random projection matrix that guarantees similar embedding quality while being much more memory efficient and allowing faster computation of the projected data.

If we note `s = 1 / density` the components of the random matrix are drawn from:

- -sqrt(s) / sqrt(n_components) with probability 1 / 2s
- 0 with probability 1 - 1 / s
- +sqrt(s) / sqrt(n_components) with probability 1 / 2s

Read more in the *User Guide*.

Parameters

n_components [int or 'auto', optional (default = 'auto')] Dimensionality of the target projection space.

n_components can be automatically adjusted according to the number of samples in the dataset and the bound given by the Johnson-Lindenstrauss lemma. In that case the quality of the embedding is controlled by the `eps` parameter.

It should be noted that Johnson-Lindenstrauss lemma can yield very conservative estimated of the required number of components as it makes no assumption on the structure of the dataset.

density [float in range]0, 1], optional (default='auto')] Ratio of non-zero component in the random projection matrix.

If density = 'auto', the value is set to the minimum density as recommended by Ping Li et al.: 1 / sqrt(n_features).

Use density = 1 / 3.0 if you want to reproduce the results from Achlioptas, 2001.

eps [strictly positive float, optional, (default=0.1)] Parameter to control the quality of the embedding according to the Johnson-Lindenstrauss lemma when n_components is set to 'auto'.

Smaller values lead to better embedding and higher number of dimensions (n_components) in the target projection space.

dense_output [boolean, optional (default=False)] If True, ensure that the output of the random projection is a dense numpy array even if the input and random projection matrix are both sparse. In practice, if the number of components is small the number of zero components in the projected data will be very small and it will be more CPU and memory efficient to use a dense representation.

If False, the projected data uses a sparse representation if the input is sparse.

random_state [int, RandomState instance or None, optional (default=None)] Control the pseudo random number generator used to generate the matrix at fit time. 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`.

Attributes

n_component_ [int] Concrete number of components computed when n_components="auto".

components_ [CSR matrix with shape [n_components, n_features]] Random matrix used for the projection.

density_ [float in range 0.0 - 1.0] Concrete density computed from when density = "auto".

See also:

GaussianRandomProjection

References

[R0fecf191e4b8-1], *[R0fecf191e4b8-2]*

Examples

```
>>> import numpy as np
>>> from sklearn.random_projection import SparseRandomProjection
>>> rng = np.random.RandomState(42)
>>> X = rng.rand(100, 10000)
>>> transformer = SparseRandomProjection(random_state=rng)
>>> X_new = transformer.fit_transform(X)
>>> X_new.shape
(100, 3947)
>>> # very few components are non-zero
>>> np.mean(transformer.components_ != 0)
0.0100...
```

Methods

fit(self, X[, y])	Generate a sparse random projection matrix
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Project the data by using matrix product with the random matrix

__init__(*self*, *n_components='auto'*, *density='auto'*, *eps=0.1*, *dense_output=False*, *random_state=None*)

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

Generate a sparse random projection matrix

Parameters

X [numpy array or scipy.sparse of shape [n_samples, n_features]] Training set: only the shape is used to find optimal random matrix dimensions based on the theory referenced in the afore mentioned papers.

y Ignored

Returns

self

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Project the data by using matrix product with the random matrix

Parameters

X [numpy array or scipy.sparse of shape [n_samples, n_features]] The input data to project into a smaller dimensional space.

Returns

X_new [numpy array or scipy sparse of shape [n_samples, n_components]] Projected array.

Examples using `sklearn.random_projection.SparseRandomProjection`

- *The Johnson-Lindenstrauss bound for embedding with random projections*
- *Manifold learning on handwritten digits: Locally Linear Embedding, Isomap...*

random_projection.johnson_lindenstrauss_min_dim(...)	Find a 'safe' number of components to randomly project to

6.35.3 sklearn.random_projection.johnson_lindenstrauss_min_dim

sklearn.random_projection.**johnson_lindenstrauss_min_dim**(*n_samples*, *eps=0.1*)
Find a 'safe' number of components to randomly project to

The distortion introduced by a random projection `p` only changes the distance between two points by a factor (1 +- eps) in an euclidean space with good probability. The projection `p` is an eps-embedding as defined by:

(1 - eps) ||u - v||^2 < ||p(u) - p(v)||^2 < (1 + eps) ||u - v||^2

Where u and v are any rows taken from a dataset of shape [n_samples, n_features], eps is in]0, 1[and p is a projection by a random Gaussian N(0, 1) matrix with shape [n_components, n_features] (or a sparse Achlioptas matrix).

The minimum number of components to guarantee the eps-embedding is given by:

n_components >= 4 log(n_samples) / (eps^2 / 2 - eps^3 / 3)

Note that the number of dimensions is independent of the original number of features but instead depends on the size of the dataset: the larger the dataset, the higher is the minimal dimensionality of an eps-embedding.

Read more in the *User Guide*.

Parameters

n_samples [int or numpy array of int greater than 0,] Number of samples. If an array is given, it will compute a safe number of components array-wise.

eps [float or numpy array of float in]0,1[, optional (default=0.1)] Maximum distortion rate as defined by the Johnson-Lindenstrauss lemma. If an array is given, it will compute a safe number of components array-wise.

Returns

n_components [int or numpy array of int,] The minimal number of components to guarantee with good probability an eps-embedding with n_samples.

References

[1], *[2]*

Examples

```
>>> johnson_lindenstrauss_min_dim(1e6, eps=0.5)
663
```

```
>>> johnson_lindenstrauss_min_dim(1e6, eps=[0.5, 0.1, 0.01])
array([    663,   11841, 1112658])
```

```
>>> johnson_lindenstrauss_min_dim([1e4, 1e5, 1e6], eps=0.1)
array([ 7894,  9868, 11841])
```

Examples using `sklearn.random_projection.johnson_lindenstrauss_min_dim`

- *The Johnson-Lindenstrauss bound for embedding with random projections*

6.36 `sklearn.semi_supervised` Semi-Supervised Learning

The *sklearn.semi_supervised* module implements semi-supervised learning algorithms. These algorithms utilized small amounts of labeled data and large amounts of unlabeled data for classification tasks. This module includes Label Propagation.

User guide: See the *Semi-Supervised* section for further details.

semi_supervised.LabelPropagation([kernel, ...])	Label Propagation classifier
semi_supervised.LabelSpreading([kernel, ...])	LabelSpreading model for semi-supervised learning

6.36.1 `sklearn.semi_supervised`.LabelPropagation

class sklearn.semi_supervised.**LabelPropagation**(*kernel='rbf'*, *gamma=20*, *n_neighbors=7*, *max_iter=1000*, *tol=0.001*, *n_jobs=None*)

Label Propagation classifier

Read more in the *User Guide*.

Parameters

kernel [{'knn', 'rbf', callable}] String identifier for kernel function to use or the kernel function itself. Only 'rbf' and 'knn' strings are valid inputs. The function passed should take two inputs, each of shape [n_samples, n_features], and return a [n_samples, n_samples] shaped weight matrix.

gamma [float] Parameter for rbf kernel

n_neighbors [integer > 0] Parameter for knn kernel

max_iter [integer] Change maximum number of iterations allowed

tol [float] Convergence tolerance: threshold to consider the system at steady state

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

Attributes

X_ [array, shape = [n_samples, n_features]] Input array.

classes_ [array, shape = [n_classes]] The distinct labels used in classifying instances.

label_distributions_ [array, shape = [n_samples, n_classes]] Categorical distribution for each item.

transduction_ [array, shape = [n_samples]] Label assigned to each item via the transduction.

n_iter_ [int] Number of iterations run.

See also:

LabelSpreading Alternate label propagation strategy more robust to noise

References

Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical Report CMU-CALD-02-107, Carnegie Mellon University, 2002 http://pages.cs.wisc.edu/~jerryzhu/pub/CMU-CALD-02-107.pdf

Examples

```
>>> import numpy as np
>>> from sklearn import datasets
>>> from sklearn.semi_supervised import LabelPropagation
>>> label_prop_model = LabelPropagation()
>>> iris = datasets.load_iris()
>>> rng = np.random.RandomState(42)
>>> random_unlabeled_points = rng.rand(len(iris.target)) < 0.3
>>> labels = np.copy(iris.target)
>>> labels[random_unlabeled_points] = -1
>>> label_prop_model.fit(iris.data, labels)
...
LabelPropagation(...)
```

Methods

`fit`(self, X, y)	
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Performs inductive inference across the model.
predict_proba(self, X)	Predict probability for each possible outcome.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.

Continued on next page

Table 6.270 – continued from previous page

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

__init__ (*self*, *kernel='rbf'*, *gamma=20*, *n_neighbors=7*, *max_iter=1000*, *tol=0.001*, *n_jobs=None*)

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Performs inductive inference across the model.

Parameters

X [array_like, shape = [n_samples, n_features]]

Returns

y [array_like, shape = [n_samples]] Predictions for input data

predict_proba (*self*, *X*)
Predict probability for each possible outcome.

Compute the probability estimates for each single sample in X and each possible outcome seen during training (categorical distribution).

Parameters

X [array_like, shape = [n_samples, n_features]]

Returns

probabilities [array, shape = [n_samples, n_classes]] Normalized probability distributions across class labels

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

6.36.2 sklearn.semi_supervised.LabelSpreading

class sklearn.semi_supervised.**LabelSpreading**(*kernel='rbf'*, *gamma=20*, *n_neighbors=7*, *alpha=0.2*, *max_iter=30*, *tol=0.001*, *n_jobs=None*)

LabelSpreading model for semi-supervised learning

This model is similar to the basic Label Propagation algorithm, but uses affinity matrix based on the normalized graph Laplacian and soft clamping across the labels.

Read more in the *User Guide*.

Parameters

kernel [{'knn', 'rbf', callable}] String identifier for kernel function to use or the kernel function itself. Only 'rbf' and 'knn' strings are valid inputs. The function passed should take two inputs, each of shape [n_samples, n_features], and return a [n_samples, n_samples] shaped weight matrix

gamma [float] parameter for rbf kernel

n_neighbors [integer > 0] parameter for knn kernel

alpha [float] Clamping factor. A value in (0, 1) that specifies the relative amount that an instance should adopt the information from its neighbors as opposed to its initial label. alpha=0 means keeping the initial label information; alpha=1 means replacing all initial information.

max_iter [integer] maximum number of iterations allowed

tol [float] Convergence tolerance: threshold to consider the system at steady state

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

Attributes

X_ [array, shape = [n_samples, n_features]] Input array.

classes_ [array, shape = [n_classes]] The distinct labels used in classifying instances.

label_distributions_ [array, shape = [n_samples, n_classes]] Categorical distribution for each item.

transduction_ [array, shape = [n_samples]] Label assigned to each item via the transduction.

n_iter_ [int] Number of iterations run.

See also:

LabelPropagation Unregularized graph based semi-supervised learning

References

Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, Bernhard Schoelkopf. Learning with local and global consistency (2004) http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.115.3219

Examples

```
>>> import numpy as np
>>> from sklearn import datasets
>>> from sklearn.semi_supervised import LabelSpreading
>>> label_prop_model = LabelSpreading()
>>> iris = datasets.load_iris()
>>> rng = np.random.RandomState(42)
>>> random_unlabeled_points = rng.rand(len(iris.target)) < 0.3
>>> labels = np.copy(iris.target)
>>> labels[random_unlabeled_points] = -1
>>> label_prop_model.fit(iris.data, labels)
...
LabelSpreading(...)
```

Methods

fit(self, X, y)	Fit a semi-supervised label propagation model based
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Performs inductive inference across the model.
predict_proba(self, X)	Predict probability for each possible outcome.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *kernel='rbf'*, *gamma=20*, *n_neighbors=7*, *alpha=0.2*, *max_iter=30*, *tol=0.001*, *n_jobs=None*)

fit(*self*, *X*, *y*)

Fit a semi-supervised label propagation model based

All the input data is provided matrix X (labeled and unlabeled) and corresponding label matrix y with a dedicated marker value for unlabeled samples.

Parameters

X [array-like, shape = [n_samples, n_features]] A {n_samples by n_samples} size matrix will be created from this

y [array_like, shape = [n_samples]] n_labeled_samples (unlabeled points are marked as -1) All unlabeled samples will be transductively assigned labels

Returns

self [returns an instance of self.]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Performs inductive inference across the model.

Parameters

X [array_like, shape = [n_samples, n_features]]

Returns

y [array_like, shape = [n_samples]] Predictions for input data

predict_proba (*self*, *X*)
Predict probability for each possible outcome.

Compute the probability estimates for each single sample in X and each possible outcome seen during training (categorical distribution).

Parameters

X [array_like, shape = [n_samples, n_features]]

Returns

probabilities [array, shape = [n_samples, n_classes]] Normalized probability distributions across class labels

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.semi_supervised.LabelSpreading`

- *Decision boundary of label propagation versus SVM on the Iris dataset*
- *Label Propagation learning a complex structure*
- *Label Propagation digits: Demonstrating performance*
- *Label Propagation digits active learning*

6.37 sklearn.svm: Support Vector Machines

The *sklearn.svm* module includes Support Vector Machine algorithms.

User guide: See the *Support Vector Machines* section for further details.

6.37.1 Estimators

svm.LinearSVC([penalty, loss, dual, tol, C, ...])	Linear Support Vector Classification.
svm.LinearSVR([epsilon, tol, C, loss, ...])	Linear Support Vector Regression.
svm.NuSVC([nu, kernel, degree, gamma, ...])	Nu-Support Vector Classification.
svm.NuSVR([nu, C, kernel, degree, gamma, ...])	Nu Support Vector Regression.
svm.OneClassSVM([kernel, degree, gamma, ...])	Unsupervised Outlier Detection.
svm.SVC([C, kernel, degree, gamma, coef0, ...])	C-Support Vector Classification.
svm.SVR([kernel, degree, gamma, coef0, tol, ...])	Epsilon-Support Vector Regression.

sklearn.svm.LinearSVC

class sklearn.svm.**LinearSVC**(*penalty='l2'*, *loss='squared_hinge'*, *dual=True*, *tol=0.0001*, *C=1.0*, *multi_class='ovr'*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *verbose=0*, *random_state=None*, *max_iter=1000*)

Linear Support Vector Classification.

Similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

Read more in the *User Guide*.

Parameters

penalty [string, 'l1' or 'l2' (default='l2')] Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to `coef_` vectors that are sparse.

loss [string, 'hinge' or 'squared_hinge' (default='squared_hinge')] Specifies the loss function. 'hinge' is the standard SVM loss (used e.g. by the SVC class) while 'squared_hinge' is the square of the hinge loss.

dual [bool, (default=True)] Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.

tol [float, optional (default=1e-4)] Tolerance for stopping criteria.

C [float, optional (default=1.0)] Penalty parameter C of the error term.

multi_class [string, 'ovr' or 'crammer_singer' (default='ovr')] Determines the multi-class strategy if *y* contains more than two classes. `"ovr"` trains n_classes one-vs-rest classifiers, while `"crammer_singer"` optimizes a joint objective over all classes. While `crammer_singer` is interesting from a theoretical perspective as it is consistent, it is seldom used in practice as it rarely leads to better accuracy and is more expensive to compute. If `"crammer_singer"` is chosen, the options loss, penalty and dual will be ignored.

fit_intercept [boolean, optional (default=True)] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to

be already centered).

intercept_scaling [float, optional (default=1)] When self.fit_intercept is True, instance vector x becomes `[x, self.intercept_scaling]`, i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

class_weight [{dict, 'balanced'}, optional] Set the parameter C of class i to `class_weight[i]*C` for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

verbose [int, (default=0)] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data for the dual coordinate descent (if `dual=True`). When `dual=False` the underlying implementation of *LinearSVC* is not random and `random_state` has no effect on the results. 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`.

max_iter [int, (default=1000)] The maximum number of iterations to be run.

Attributes

coef_ [array, shape = [n_features] if n_classes == 2 else [n_classes, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

`coef_` is a readonly property derived from `raw_coef_` that follows the internal memory layout of liblinear.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

See also:

SVC Implementation of Support Vector Machine classifier using libsvm: the kernel can be non-linear but its SMO algorithm does not scale to large number of samples as LinearSVC does. Furthermore SVC multi-class mode is implemented using one vs one scheme while LinearSVC uses one vs the rest. It is possible to implement one vs the rest with SVC by using the *sklearn.multiclass.OneVsRestClassifier* wrapper. Finally SVC can fit dense data without memory copy if the input is C-contiguous. Sparse data will still incur memory copy though.

sklearn.linear_model.SGDClassifier SGDClassifier can optimize the same cost function as LinearSVC by adjusting the penalty and loss parameters. In addition it requires less memory, allows incremental (online) learning, and implements various loss functions and regularization regimes.

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon to have slightly different results for the same input data. If that happens, try with a smaller `tol` parameter.

The underlying implementation, liblinear, uses a sparse internal representation for the data that will incur a memory copy.

Predict output may not match that of standalone liblinear in certain cases. See *differences from liblinear* in the narrative documentation.

References

LIBLINEAR: A Library for Large Linear Classification

Examples

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_features=4, random_state=0)
>>> clf = LinearSVC(random_state=0, tol=1e-5)
>>> clf.fit(X, y)
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=0, tol=1e-05, verbose=0)
>>> print(clf.coef_)
[[0.085... 0.394... 0.498... 0.375...]]
>>> print(clf.intercept_)
[0.284...]
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

Methods

decision_function(self, X)	Predict confidence scores for samples.
densify(self)	Convert coefficient matrix to dense array format.
fit(self, X, y[, sample_weight])	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict class labels for samples in X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.
sparsify(self)	Convert coefficient matrix to sparse format.

__init__ (*self*, *penalty='l2'*, *loss='squared_hinge'*, *dual=True*, *tol=0.0001*, *C=1.0*, *multi_class='ovr'*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *verbose=0*, *random_state=None*, *max_iter=1000*)

decision_function (*self*, *X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

densify(*self*)

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a numpy.ndarray. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns

self [estimator]

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

Fit the model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target vector relative to X

sample_weight [array-like, shape = [n_samples], optional] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

Returns

self [object]

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)

Predict class labels for samples in X.

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape [n_samples]] Predicted class label per sample.

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

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params(*self*, ***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.

Returns

self

sparsify(*self*)
Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual numpy.ndarray representation.

The `intercept_` member is not converted.

Returns

self [estimator]

Notes

For non-sparse models, i.e. when there are not many zeros in `coef_`, this may actually *increase* memory usage, so use this method with care. A rule of thumb is that the number of zero elements, which can be computed with `(coef_ == 0).sum()`, must be more than 50% for this to provide significant benefits.

After calling this method, further fitting with the partial_fit method (if any) will not work until you call densify.

Examples using `sklearn.svm.LinearSVC`

- *Explicit feature map approximation for RBF kernels*
- *Comparison of Calibration of Classifiers*
- *Probability Calibration curves*
- *Selecting dimensionality reduction with Pipeline and GridSearchCV*
- *Column Transformer with Heterogeneous Data Sources*
- *Pipeline Anova SVM*
- *Balance model complexity and cross-validated score*
- *Precision-Recall*
- *Feature discretization*
- *Plot different SVM classifiers in the iris dataset*
- *Scaling the regularization parameter for SVCs*
- *Classification of text documents using sparse features*

sklearn.svm.LinearSVR

class sklearn.svm.**LinearSVR**(*epsilon=0.0*, *tol=0.0001*, *C=1.0*, *loss='epsilon_insensitive'*, *fit_intercept=True*, *intercept_scaling=1.0*, *dual=True*, *verbose=0*, *random_state=None*, *max_iter=1000*)

Linear Support Vector Regression.

Similar to SVR with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input.

Read more in the *User Guide*.

Parameters

epsilon [float, optional (default=0.0)] Epsilon parameter in the epsilon-insensitive loss function. Note that the value of this parameter depends on the scale of the target variable y. If unsure, set `epsilon=0`.

tol [float, optional (default=1e-4)] Tolerance for stopping criteria.

C [float, optional (default=1.0)] Penalty parameter C of the error term. The penalty is a squared l2 penalty. The bigger this parameter, the less regularization is used.

loss [string, optional (default='epsilon_insensitive')] Specifies the loss function. The epsilon-insensitive loss (standard SVR) is the L1 loss, while the squared epsilon-insensitive loss ('squared_epsilon_insensitive') is the L2 loss.

fit_intercept [boolean, optional (default=True)] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be already centered).

intercept_scaling [float, optional (default=1)] When self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

dual [bool, (default=True)] Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.

verbose [int, (default=0)] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. 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`.

max_iter [int, (default=1000)] The maximum number of iterations to be run.

Attributes

coef_ [array, shape = [n_features] if n_classes == 2 else [n_classes, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

*coef_* is a readonly property derived from `raw_coef_` that follows the internal memory layout of liblinear.

intercept_ [array, shape = [1] if n_classes == 2 else [n_classes]] Constants in decision function.

See also:

LinearSVC Implementation of Support Vector Machine classifier using the same library as this class (liblinear).

SVR Implementation of Support Vector Machine regression using libsvm: the kernel can be non-linear but its SMO algorithm does not scale to large number of samples as LinearSVC does.

sklearn.linear_model.SGDRegressor SGDRegressor can optimize the same cost function as LinearSVR by adjusting the penalty and loss parameters. In addition it requires less memory, allows incremental (online) learning, and implements various loss functions and regularization regimes.

Examples

```
>>> from sklearn.svm import LinearSVR
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_features=4, random_state=0)
>>> regr = LinearSVR(random_state=0, tol=1e-5)
>>> regr.fit(X, y)
LinearSVR(C=1.0, dual=True, epsilon=0.0, fit_intercept=True,
     intercept_scaling=1.0, loss='epsilon_insensitive', max_iter=1000,
     random_state=0, tol=1e-05, verbose=0)
>>> print(regr.coef_)
[16.35... 26.91... 42.30... 60.47...]
>>> print(regr.intercept_)
[-4.29...]
>>> print(regr.predict([[0, 0, 0, 0]]))
[-4.29...]
```

Methods

fit(self, X, y[, sample_weight])	Fit the model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Predict using the linear model
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *epsilon=0.0*, *tol=0.0001*, *C=1.0*, *loss='epsilon_insensitive'*, *fit_intercept=True*, *intercept_scaling=1.0*, *dual=True*, *verbose=0*, *random_state=None*, *max_iter=1000*)

fit(*self*, *X*, *y*, *sample_weight=None*)
Fit the model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target vector relative to X

sample_weight [array-like, shape = [n_samples], optional] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

Returns

self [object]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)
Predict using the linear model

Parameters

X [array_like or sparse matrix, shape (n_samples, n_features)] Samples.

Returns

C [array, shape (n_samples,)] Returns predicted values.

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

sklearn.svm.NuSVC

class sklearn.svm.**NuSVC**(*nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma='auto_deprecated'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *random_state=None*)
Nu-Support Vector Classification.

Similar to SVC but uses a parameter to control the number of support vectors.

The implementation is based on libsvm.

Read more in the *User Guide*.

Parameters

nu [float, optional (default=0.5)] An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1].

kernel [string, optional (default='rbf')] Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [float, optional (default='auto')] Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

Current default is 'auto' which uses 1 / n_features, if `gamma='scale'` is passed then it uses 1 / (n_features * X.var()) as value of gamma. The current default of gamma, 'auto', will change to 'scale' in version 0.22. 'auto_deprecated', a deprecated version of 'auto' is used as a default indicating that no explicit value of gamma was passed.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking [boolean, optional (default=True)] Whether to use the shrinking heuristic.

probability [boolean, optional (default=False)] Whether to enable probability estimates. This must be enabled prior to calling *fit*, and will slow down that method.

tol [float, optional (default=1e-3)] Tolerance for stopping criterion.

cache_size [float, optional] Specify the size of the kernel cache (in MB).

class_weight [{dict, 'balanced'}, optional] Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies as `n_samples / (n_classes * np.bincount(y))`

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape ['ovo', 'ovr', default='ovr'] Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2).

Changed in version 0.19: decision_function_shape is 'ovr' by default.

New in version 0.17: *decision_function_shape='ovr'* is recommended.

Changed in version 0.17: Deprecated *decision_function_shape='ovo' and None*.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator used when shuffling the data for probability estimates. 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`.

Attributes

support_ [array-like, shape = [n_SV]] Indices of support vectors.

support_vectors_ [array-like, shape = [n_SV, n_features]] Support vectors.

n_support_ [array-like, dtype=int32, shape = [n_class]] Number of support vectors for each class.

dual_coef_ [array, shape = [n_class-1, n_SV]] Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide for details.

coef_ [array, shape = [n_class * (n_class-1) / 2, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

*coef_* is readonly property derived from `dual_coef_` and `support_vectors_`.

intercept_ [array, shape = [n_class * (n_class-1) / 2]] Constants in decision function.

See also:

SVC Support Vector Machine for classification using libsvm.

LinearSVC Scalable linear Support Vector Machine for classification using liblinear.

Notes

References: LIBSVM: A Library for Support Vector Machines

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from sklearn.svm import NuSVC
```

```
>>> clf = NuSVC(gamma='scale')
>>> clf.fit(X, y)
NuSVC(cache_size=200, class_weight=None, coef0=0.0,
      decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
      max_iter=-1, nu=0.5, probability=False, random_state=None,
      shrinking=True, tol=0.001, verbose=False)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(self, X)	Evaluates the decision function for the samples in X.
fit(self, X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform classification on samples in X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *nu=0.5*, *kernel='rbf'*, *degree=3*, *gamma='auto_deprecated'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *random_state=None*)

decision_function(*self*, *X*)

Evaluates the decision function for the samples in X.

Parameters

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

Returns

X [array-like, shape (n_samples, n_classes * (n_classes-1) / 2)] Returns the decision function of the sample for each class in the model. If decision_function_shape='ovr', the shape is (n_samples, n_classes).

Notes

If decision_function_shape='ovo', the function values are proportional to the distance of the samples X to the separating hyperplane. If the exact distances are required, divide the function values by the norm of the weight vector (`coef_`). See also this question for further details. If decision_function_shape='ovr', the decision function is a monotonic transformation of ovo decision function.

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

Fit the SVM model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features. For kernel="precomputed", the expected shape of X is (n_samples, n_samples).

y [array-like, shape (n_samples,)] Target values (class labels in classification, real numbers in regression)

sample_weight [array-like, shape (n_samples,)] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns

self [object]

Notes

If X and y are not C-ordered and contiguous arrays of np.float64 and X is not a scipy.sparse.csr_matrix, X and/or y may be copied.

If X is a dense array, then the other methods will not support sparse matrices as input.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)
Perform classification on samples in X.

For an one-class model, +1 or -1 is returned.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

y_pred [array, shape (n_samples,)] Class labels for samples in X.

predict_log_proba
Compute log probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute `probability` set to True.

Parameters

X [array-like, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

T [array-like, shape (n_samples, n_classes)] Returns the log-probabilities of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

predict_proba
Compute probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute `probability` set to True.

Parameters

X [array-like, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

T [array-like, shape (n_samples, n_classes)] Returns the probability of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.svm.NuSVC`

- *Non-linear SVM*

sklearn.svm.NuSVR

class sklearn.svm.**NuSVR**(*nu=0.5*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='auto_deprecated'*, *coef0=0.0*, *shrinking=True*, *tol=0.001*, *cache_size=200*, *verbose=False*, *max_iter=-1*)

Nu Support Vector Regression.

Similar to NuSVC, for regression, uses a parameter nu to control the number of support vectors. However, unlike NuSVC, where nu replaces C, here nu replaces the parameter epsilon of epsilon-SVR.

The implementation is based on libsvm.

Read more in the *User Guide*.

Parameters

nu [float, optional] An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken.

C [float, optional (default=1.0)] Penalty parameter C of the error term.

kernel [string, optional (default='rbf')] Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [float, optional (default='auto')] Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

Current default is 'auto' which uses 1 / n_features, if `gamma='scale'` is passed then it uses 1 / (n_features * X.var()) as value of gamma. The current default of gamma, 'auto', will change to 'scale' in version 0.22. 'auto_deprecated', a deprecated version of 'auto' is used as a default indicating that no explicit value of gamma was passed.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking [boolean, optional (default=True)] Whether to use the shrinking heuristic.

tol [float, optional (default=1e-3)] Tolerance for stopping criterion.

cache_size [float, optional] Specify the size of the kernel cache (in MB).

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

Attributes

support_ [array-like, shape = [n_SV]] Indices of support vectors.

support_vectors_ [array-like, shape = [nSV, n_features]] Support vectors.

dual_coef_ [array, shape = [1, n_SV]] Coefficients of the support vector in the decision function.

coef_ [array, shape = [1, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

*coef_* is readonly property derived from `dual_coef_` and `support_vectors_`.

intercept_ [array, shape = [1]] Constants in decision function.

See also:

NuSVC Support Vector Machine for classification implemented with libsvm with a parameter to control the number of support vectors.

SVR epsilon Support Vector Machine for regression implemented with libsvm.

Notes

References: LIBSVM: A Library for Support Vector Machines

Examples

```
>>> from sklearn.svm import NuSVR
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> np.random.seed(0)
>>> y = np.random.randn(n_samples)
>>> X = np.random.randn(n_samples, n_features)
>>> clf = NuSVR(gamma='scale', C=1.0, nu=0.1)
>>> clf.fit(X, y)
NuSVR(C=1.0, cache_size=200, coef0=0.0, degree=3, gamma='scale',
      kernel='rbf', max_iter=-1, nu=0.1, shrinking=True, tol=0.001,
      verbose=False)
```

Methods

fit(self, X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform regression on samples in X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *nu=0.5*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='auto_deprecated'*, *coef0=0.0*, *shrinking=True*, *tol=0.001*, *cache_size=200*, *verbose=False*, *max_iter=-1*)

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

Fit the SVM model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features. For kernel="precomputed", the expected shape of X is (n_samples, n_samples).

y [array-like, shape (n_samples,)] Target values (class labels in classification, real numbers in regression)

sample_weight [array-like, shape (n_samples,)] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns

self [object]

Notes

If X and y are not C-ordered and contiguous arrays of np.float64 and X is not a scipy.sparse.csr_matrix, X and/or y may be copied.

If X is a dense array, then the other methods will not support sparse matrices as input.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*)
Perform regression on samples in X.

For an one-class model, +1 (inlier) or -1 (outlier) is returned.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returns

y_pred [array, shape (n_samples,)]

score(*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*).

To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.svm.NuSVR`

- *Model Complexity Influence*

`sklearn.svm`.OneClassSVM

class `sklearn.svm.`**OneClassSVM**(*kernel='rbf'*, *degree=3*, *gamma='auto_deprecated'*, *coef0=0.0*, *tol=0.001*, *nu=0.5*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*, *random_state=None*)
Unsupervised Outlier Detection.

Estimate the support of a high-dimensional distribution.

The implementation is based on libsvm.

Read more in the *User Guide*.

Parameters

kernel [string, optional (default='rbf')] Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [float, optional (default='auto')] Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

Current default is 'auto' which uses 1 / n_features, if `gamma='scale'` is passed then it uses 1 / (n_features * X.var()) as value of gamma. The current default of gamma, 'auto', will change to 'scale' in version 0.22. 'auto_deprecated', a deprecated version of 'auto' is used as a default indicating that no explicit value of gamma was passed.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

tol [float, optional] Tolerance for stopping criterion.

nu [float, optional] An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1]. By default 0.5 will be taken.

shrinking [boolean, optional] Whether to use the shrinking heuristic.

cache_size [float, optional] Specify the size of the kernel cache (in MB).

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

random_state [int, RandomState instance or None, optional (default=None)] Ignored.

Deprecated since version 0.20: `random_state` has been deprecated in 0.20 and will be removed in 0.22.

Attributes

support_ [array-like, shape = [n_SV]] Indices of support vectors.

support_vectors_ [array-like, shape = [nSV, n_features]] Support vectors.

dual_coef_ [array, shape = [1, n_SV]] Coefficients of the support vectors in the decision function.

coef_ [array, shape = [1, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

*coef_* is readonly property derived from `dual_coef_` and `support_vectors_`

intercept_ [array, shape = [1,]] Constant in the decision function.

offset_ [float] Offset used to define the decision function from the raw scores. We have the relation: decision_function = score_samples - `offset_`. The offset is the opposite of `intercept_` and is provided for consistency with other outlier detection algorithms.

Examples

```
>>> from sklearn.svm import OneClassSVM
>>> X = [[0], [0.44], [0.45], [0.46], [1]]
>>> clf = OneClassSVM(gamma='auto').fit(X)
>>> clf.predict(X)
array([-1,  1,  1,  1, -1])
>>> clf.score_samples(X)
array([1.7798..., 2.0547..., 2.0556..., 2.0561..., 1.7332...])
```

Methods

decision_function(self, X)	Signed distance to the separating hyperplane.
fit(self, X[, y, sample_weight])	Detects the soft boundary of the set of samples X.
fit_predict(self, X[, y])	Performs fit on X and returns labels for X.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform classification on samples in X.
score_samples(self, X)	Raw scoring function of the samples.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *kernel='rbf'*, *degree=3*, *gamma='auto_deprecated'*, *coef0=0.0*, *tol=0.001*, *nu=0.5*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*, *random_state=None*)

decision_function(*self*, *X*)
Signed distance to the separating hyperplane.

Signed distance is positive for an inlier and negative for an outlier.

Parameters

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

Returns

dec [array-like, shape (n_samples,)] Returns the decision function of the samples.

fit (*self*, *X*, *y=None*, *sample_weight=None*, ***params*)
Detects the soft boundary of the set of samples X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Set of samples, where n_samples is the number of samples and n_features is the number of features.

sample_weight [array-like, shape (n_samples,)] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

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

Returns

self [object]

Notes

If X is not a C-ordered contiguous array it is copied.

fit_predict (*self*, *X*, *y=None*)
Performs fit on X and returns labels for X.

Returns -1 for outliers and 1 for inliers.

Parameters

X [ndarray, shape (n_samples, n_features)] Input data.

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

Returns

y [ndarray, shape (n_samples,)] 1 for inliers, -1 for outliers.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Perform classification on samples in X.

For a one-class model, +1 or -1 is returned.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

y_pred [array, shape (n_samples,)] Class labels for samples in X.

score_samples(*self*, *X*)
Raw scoring function of the samples.

Parameters

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

Returns

score_samples [array-like, shape (n_samples,)] Returns the (unshifted) scoring function of the samples.

set_params(*self*, ***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.

Returns

self

Examples using `sklearn.svm.OneClassSVM`

- *Comparing anomaly detection algorithms for outlier detection on toy datasets*
- *Outlier detection on a real data set*
- *Species distribution modeling*
- *Libsvm GUI*
- *One-class SVM with non-linear kernel (RBF)*

`sklearn.svm.SVC`

class `sklearn.svm.`**SVC**(*C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='auto_deprecated'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *random_state=None*)
C-Support Vector Classification.

The implementation is based on libsvm. The fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples. For large datasets consider using `sklearn.linear_model.LinearSVC` or *`sklearn.linear_model.SGDClassifier`* instead, possibly after a *`sklearn.kernel_approximation.Nystroem`* transformer.

The multiclass support is handled according to a one-vs-one scheme.

For details on the precise mathematical formulation of the provided kernel functions and how `gamma`, `coef0` and `degree` affect each other, see the corresponding section in the narrative documentation: *Kernel functions*.

Read more in the *User Guide*.

Parameters

C [float, optional (default=1.0)] Penalty parameter C of the error term.

kernel [string, optional (default='rbf')] Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [float, optional (default='auto')] Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

Current default is 'auto' which uses 1 / n_features, if `gamma='scale'` is passed then it uses 1 / (n_features * X.var()) as value of gamma. The current default of gamma, 'auto', will change to 'scale' in version 0.22. 'auto_deprecated', a deprecated version of 'auto' is used as a default indicating that no explicit value of gamma was passed.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking [boolean, optional (default=True)] Whether to use the shrinking heuristic.

probability [boolean, optional (default=False)] Whether to enable probability estimates. This must be enabled prior to calling *fit*, and will slow down that method.

tol [float, optional (default=1e-3)] Tolerance for stopping criterion.

cache_size [float, optional] Specify the size of the kernel cache (in MB).

class_weight [{dict, 'balanced'}, optional] Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape ['ovo', 'ovr', default='ovr'] Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, one-vs-one ('ovo') is always used as multi-class strategy.

Changed in version 0.19: decision_function_shape is 'ovr' by default.

New in version 0.17: *decision_function_shape='ovr'* is recommended.

Changed in version 0.17: Deprecated *decision_function_shape='ovo' and None*.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator used when shuffling the data for probability estimates. 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`.

Attributes

support_ [array-like, shape = [n_SV]] Indices of support vectors.

support_vectors_ [array-like, shape = [n_SV, n_features]] Support vectors.

n_support_ [array-like, dtype=int32, shape = [n_class]] Number of support vectors for each class.

dual_coef_ [array, shape = [n_class-1, n_SV]] Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide for details.

coef_ [array, shape = [n_class * (n_class-1) / 2, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

*coef_* is a readonly property derived from `dual_coef_` and `support_vectors_`.

intercept_ [array, shape = [n_class * (n_class-1) / 2]] Constants in decision function.

fit_status_ [int] 0 if correctly fitted, 1 otherwise (will raise warning)

probA_ [array, shape = [n_class * (n_class-1) / 2]]

probB_ [array, shape = [n_class * (n_class-1) / 2]] If probability=True, the parameters learned in Platt scaling to produce probability estimates from decision values. If probability=False, an empty array. Platt scaling uses the logistic function `1 / (1 + exp(decision_value * probA_ + probB_))` where `probA_` and `probB_` are learned from the dataset *[R20c70293ef72-2]*. For more information on the multiclass case and training procedure see section 8 of *[R20c70293ef72-1]*.

See also:

SVR Support Vector Machine for Regression implemented using libsvm.

LinearSVC Scalable Linear Support Vector Machine for classification implemented using liblinear. Check the See also section of LinearSVC for more comparison element.

References

[R20c70293ef72-1], *[R20c70293ef72-2]*

Examples

```
>>> import numpy as np
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from sklearn.svm import SVC
>>> clf = SVC(gamma='auto')
>>> clf.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
>>> print(clf.predict([[-0.8, -1]]))
[1]
```

Methods

decision_function(self, X)	Evaluates the decision function for the samples in X.
fit(self, X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform classification on samples in X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='auto_deprecated'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *random_state=None*)

decision_function(*self*, *X*)

Evaluates the decision function for the samples in X.

Parameters

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

Returns

X [array-like, shape (n_samples, n_classes * (n_classes-1) / 2)] Returns the decision function of the sample for each class in the model. If decision_function_shape='ovr', the shape is (n_samples, n_classes).

Notes

If decision_function_shape='ovo', the function values are proportional to the distance of the samples X to the separating hyperplane. If the exact distances are required, divide the function values by the norm of the weight vector (`coef_`). See also this question for further details. If decision_function_shape='ovr', the decision function is a monotonic transformation of ovo decision function.

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

Fit the SVM model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features. For kernel="precomputed", the expected shape of X is (n_samples, n_samples).

y [array-like, shape (n_samples,)] Target values (class labels in classification, real numbers in regression)

sample_weight [array-like, shape (n_samples,)] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns

self [object]

Notes

If X and y are not C-ordered and contiguous arrays of np.float64 and X is not a scipy.sparse.csr_matrix, X and/or y may be copied.

If X is a dense array, then the other methods will not support sparse matrices as input.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Perform classification on samples in X.

For an one-class model, +1 or -1 is returned.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

y_pred [array, shape (n_samples,)] Class labels for samples in X.

predict_log_proba
Compute log probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute `probability` set to True.

Parameters

X [array-like, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

T [array-like, shape (n_samples, n_classes)] Returns the log-probabilities of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

predict_proba
Compute probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute `probability` set to True.

Parameters

X [array-like, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is [n_samples_test, n_samples_train]

Returns

T [array-like, shape (n_samples, n_classes)] Returns the probability of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute *classes_*.

Notes

The probability model is created using cross validation, so the results can be slightly different than those obtained by predict. Also, it will produce meaningless results on very small datasets.

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.svm.SVC`

- *Multilabel classification*
- *Explicit feature map approximation for RBF kernels*
- *Faces recognition example using eigenfaces and SVMs*
- *Libsvm GUI*
- *Recognizing hand-written digits*
- *Plot classification probability*
- *Classifier comparison*
- *Concatenating multiple feature extraction methods*
- *Plot the decision boundaries of a VotingClassifier*
- *Cross-validation on Digits Dataset Exercise*
- *SVM Exercise*
- *Recursive feature elimination*

- *Recursive feature elimination with cross-validation*
- *Test with permutations the significance of a classification score*
- *Univariate Feature Selection*
- *Plotting Validation Curves*
- *Parameter estimation using grid search with cross-validation*
- *Receiver Operating Characteristic (ROC) with cross validation*
- *Nested versus non-nested cross-validation*
- *Confusion matrix*
- *Receiver Operating Characteristic (ROC)*
- *Plotting Learning Curves*
- *Feature discretization*
- *Decision boundary of label propagation versus SVM on the Iris dataset*
- *SVM: Maximum margin separating hyperplane*
- *SVM with custom kernel*
- *SVM: Weighted samples*
- *SVM: Separating hyperplane for unbalanced classes*
- *SVM-Kernels*
- *SVM-Anova: SVM with univariate feature selection*
- *SVM Margins Example*
- *Plot different SVM classifiers in the iris dataset*
- *RBF SVM parameters*

sklearn.svm.SVR

class sklearn.svm.**SVR**(*kernel='rbf'*, *degree=3*, *gamma='auto_deprecated'*, *coef0=0.0*, *tol=0.001*, *C=1.0*, *epsilon=0.1*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*)

Epsilon-Support Vector Regression.

The free parameters in the model are C and epsilon.

The implementation is based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to datasets with more than a couple of 10000 samples. For large datasets consider using `sklearn.linear_model.LinearSVR` or *sklearn.linear_model.SGDRegressor* instead, possibly after a *sklearn.kernel_approximation.Nystroem* transformer.

Read more in the *User Guide*.

Parameters

kernel [string, optional (default='rbf')] Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

degree [int, optional (default=3)] Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma [float, optional (default='auto')] Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

Current default is 'auto' which uses 1 / n_features, if `gamma='scale'` is passed then it uses 1 / (n_features * X.var()) as value of gamma. The current default of gamma, 'auto', will change to 'scale' in version 0.22. 'auto_deprecated', a deprecated version of 'auto' is used as a default indicating that no explicit value of gamma was passed.

coef0 [float, optional (default=0.0)] Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

tol [float, optional (default=1e-3)] Tolerance for stopping criterion.

C [float, optional (default=1.0)] Penalty parameter C of the error term.

epsilon [float, optional (default=0.1)] Epsilon in the epsilon-SVR model. It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.

shrinking [boolean, optional (default=True)] Whether to use the shrinking heuristic.

cache_size [float, optional] Specify the size of the kernel cache (in MB).

verbose [bool, default: False] Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter [int, optional (default=-1)] Hard limit on iterations within solver, or -1 for no limit.

Attributes

support_ [array-like, shape = [n_SV]] Indices of support vectors.

support_vectors_ [array-like, shape = [nSV, n_features]] Support vectors.

dual_coef_ [array, shape = [1, n_SV]] Coefficients of the support vector in the decision function.

coef_ [array, shape = [1, n_features]] Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

*coef_* is readonly property derived from `dual_coef_` and `support_vectors_`.

intercept_ [array, shape = [1]] Constants in decision function.

See also:

NuSVR Support Vector Machine for regression implemented using libsvm using a parameter to control the number of support vectors.

LinearSVR Scalable Linear Support Vector Machine for regression implemented using liblinear.

Notes

References: LIBSVM: A Library for Support Vector Machines

Examples

```
>>> from sklearn.svm import SVR
>>> import numpy as np
>>> n_samples, n_features = 10, 5
>>> rng = np.random.RandomState(0)
```

```
>>> y = rng.randn(n_samples)
>>> X = rng.randn(n_samples, n_features)
>>> clf = SVR(gamma='scale', C=1.0, epsilon=0.2)
>>> clf.fit(X, y)
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.2, gamma='scale',
    kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

Methods

fit(self, X, y[, sample_weight])	Fit the SVM model according to the given training data.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X)	Perform regression on samples in X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__ (*self*, *kernel='rbf'*, *degree=3*, *gamma='auto_deprecated'*, *coef0=0.0*, *tol=0.001*, *C=1.0*, *epsilon=0.1*, *shrinking=True*, *cache_size=200*, *verbose=False*, *max_iter=-1*)

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

Fit the SVM model according to the given training data.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Training vectors, where n_samples is the number of samples and n_features is the number of features. For kernel="precomputed", the expected shape of X is (n_samples, n_samples).

y [array-like, shape (n_samples,)] Target values (class labels in classification, real numbers in regression)

sample_weight [array-like, shape (n_samples,)] Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns

self [object]

Notes

If X and y are not C-ordered and contiguous arrays of np.float64 and X is not a scipy.sparse.csr_matrix, X and/or y may be copied.

If X is a dense array, then the other methods will not support sparse matrices as input.

get_params (*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*)
Perform regression on samples in X.

For an one-class model, +1 (inlier) or -1 (outlier) is returned.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returns

y_pred [array, shape (n_samples,)]

score (*self*, *X*, *y*, *sample_weight=None*)
Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.svm.SVR`

- *Comparison of kernel ridge regression and SVR*
- *Prediction Latency*

- *Support Vector Regression (SVR) using linear and non-linear kernels*

svm.l1_min_c(X, y[, loss, fit_intercept, ...])	Return the lowest bound for C such that for C in (l1_min_C, infinity) the model is guaranteed not to be empty.

sklearn.svm.l1_min_c

sklearn.svm.**l1_min_c**(*X*, *y*, *loss='squared_hinge'*, *fit_intercept=True*, *intercept_scaling=1.0*)

Return the lowest bound for C such that for C in (l1_min_C, infinity) the model is guaranteed not to be empty. This applies to l1 penalized classifiers, such as LinearSVC with penalty='l1' and linear_model.LogisticRegression with penalty='l1'.

This value is valid if class_weight parameter in fit() is not set.

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] Training vector, where n_samples in the number of samples and n_features is the number of features.

y [array, shape = [n_samples]] Target vector relative to X

loss [{'squared_hinge', 'log'}, default 'squared_hinge'] Specifies the loss function. With 'squared_hinge' it is the squared hinge loss (a.k.a. L2 loss). With 'log' it is the loss of logistic regression models.

fit_intercept [bool, default: True] Specifies if the intercept should be fitted by the model. It must match the fit() method parameter.

intercept_scaling [float, default: 1] when fit_intercept is True, instance vector x becomes [x, intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. It must match the fit() method parameter.

Returns

l1_min_c [float] minimum value for C

Examples using sklearn.svm.l1_min_c

- *Regularization path of L1- Logistic Regression*

6.37.2 Low-level methods

svm.libsvm.cross_validation()	Binding of the cross-validation routine (low-level routine)
svm.libsvm.decision_function()	Predict margin (libsvm name for this is predict_values)
svm.libsvm.fit()	Train the model using libsvm (low-level method)
svm.libsvm.predict()	Predict target values of X given a model (low-level method)
svm.libsvm.predict_proba()	Predict probabilities

sklearn.svm.libsvm.cross_validation

sklearn.svm.libsvm.**cross_validation**()

Binding of the cross-validation routine (low-level routine)

Parameters

X [array-like, dtype=float, size=[n_samples, n_features]]

Y [array, dtype=float, size=[n_samples]] target vector

svm_type [{0, 1, 2, 3, 4}] Type of SVM: C SVC, nu SVC, one class, epsilon SVR, nu SVR

kernel [{'linear', 'rbf', 'poly', 'sigmoid', 'precomputed'}] Kernel to use in the model: linear, polynomial, RBF, sigmoid or precomputed.

degree [int] Degree of the polynomial kernel (only relevant if kernel is set to polynomial)

gamma [float] Gamma parameter in rbf, poly and sigmoid kernels. Ignored by other kernels. 0.1 by default.

coef0 [float] Independent parameter in poly/sigmoid kernel.

tol [float] Stopping criteria.

C [float] C parameter in C-Support Vector Classification

nu [float]

cache_size [float]

random_seed [int, optional] Seed for the random number generator used for probability estimates. 0 by default.

Returns

target [array, float]

sklearn.svm.libsvm.decision_function

sklearn.svm.libsvm.**decision_function**()

Predict margin (libsvm name for this is predict_values)

We have to reconstruct model and parameters to make sure we stay in sync with the python object.

sklearn.svm.libsvm.fit

sklearn.svm.libsvm.**fit**()

Train the model using libsvm (low-level method)

Parameters

X [array-like, dtype=float64, size=[n_samples, n_features]]

Y [array, dtype=float64, size=[n_samples]] target vector

svm_type [{0, 1, 2, 3, 4}, optional] Type of SVM: C_SVC, NuSVC, OneClassSVM, EpsilonSVR or NuSVR respectively. 0 by default.

kernel [{'linear', 'rbf', 'poly', 'sigmoid', 'precomputed'}, optional] Kernel to use in the model: linear, polynomial, RBF, sigmoid or precomputed. 'rbf' by default.

degree [int32, optional] Degree of the polynomial kernel (only relevant if kernel is set to polynomial), 3 by default.

gamma [float64, optional] Gamma parameter in rbf, poly and sigmoid kernels. Ignored by other kernels. 0.1 by default.

coef0 [float64, optional] Independent parameter in poly/sigmoid kernel. 0 by default.

tol [float64, optional] Numeric stopping criterion (WRITEME). 1e-3 by default.

C [float64, optional] C parameter in C-Support Vector Classification. 1 by default.

nu [float64, optional] 0.5 by default.

epsilon [double, optional] 0.1 by default.

class_weight [array, dtype float64, shape (n_classes,), optional] np.empty(0) by default.

sample_weight [array, dtype float64, shape (n_samples,), optional] np.empty(0) by default.

shrinking [int, optional] 1 by default.

probability [int, optional] 0 by default.

cache_size [float64, optional] Cache size for gram matrix columns (in megabytes). 100 by default.

max_iter [int (-1 for no limit), optional.] Stop solver after this many iterations regardless of accuracy (XXX Currently there is no API to know whether this kicked in.) -1 by default.

random_seed [int, optional] Seed for the random number generator used for probability estimates. 0 by default.

Returns

support [array, shape=[n_support]] index of support vectors

support_vectors [array, shape=[n_support, n_features]] support vectors (equivalent to X[support]). Will return an empty array in the case of precomputed kernel.

n_class_SV [array] number of support vectors in each class.

sv_coef [array] coefficients of support vectors in decision function.

intercept [array] intercept in decision function

probA, probB [array] probability estimates, empty array for probability=False

sklearn.svm.libsvm.predict

sklearn.svm.libsvm.**predict**()

Predict target values of X given a model (low-level method)

Parameters

X [array-like, dtype=float, size=[n_samples, n_features]]

svm_type [{0, 1, 2, 3, 4}] Type of SVM: C SVC, nu SVC, one class, epsilon SVR, nu SVR

kernel [{'linear', 'rbf', 'poly', 'sigmoid', 'precomputed'}] Type of kernel.

degree [int] Degree of the polynomial kernel.

gamma [float] Gamma parameter in rbf, poly and sigmoid kernels. Ignored by other kernels. 0.1 by default.

coef0 [float] Independent parameter in poly/sigmoid kernel.

Returns

dec_values [array] predicted values.

sklearn.svm.libsvm.predict_proba

sklearn.svm.libsvm.**predict_proba**()

Predict probabilities

svm_model stores all parameters needed to predict a given value.

For speed, all real work is done at the C level in function copy_predict (libsvm_helper.c).

We have to reconstruct model and parameters to make sure we stay in sync with the python object.

See sklearn.svm.predict for a complete list of parameters.

Parameters

X [array-like, dtype=float]

kernel [{'linear', 'rbf', 'poly', 'sigmoid', 'precomputed'}]

Returns

dec_values [array] predicted values.

6.38 sklearn.tree: Decision Trees

The *sklearn.tree* module includes decision tree-based models for classification and regression.

User guide: See the *Decision Trees* section for further details.

tree.DecisionTreeClassifier([criterion, ...])	A decision tree classifier.
tree.DecisionTreeRegressor([criterion, ...])	A decision tree regressor.
tree.ExtraTreeClassifier([criterion, ...])	An extremely randomized tree classifier.
tree.ExtraTreeRegressor([criterion, ...])	An extremely randomized tree regressor.

6.38.1 sklearn.tree.DecisionTreeClassifier

class sklearn.tree.**DecisionTreeClassifier**(*criterion='gini'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *class_weight=None*, *presort=False*)

A decision tree classifier.

Read more in the *User Guide*.

Parameters

criterion [string, optional (default="gini")] The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

splitter [string, optional (default="best")] The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth [int or None, optional (default=None)] The maximum depth of the tree. If None,

then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, string or None, optional (default=None)] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

max_leaf_nodes [int or None, optional (default=None)] Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where N is the total number of samples, N_t is the number of samples at the current node, N_t_L is the number of samples in the left child, and N_t_R is the number of samples in the right child.

N, N_t, N_t_R and N_t_L all refer to the weighted sum, if sample_weight is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: min_impurity_split has been deprecated in favor of min_impurity_decrease in 0.19. The default value of min_impurity_split will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use min_impurity_decrease instead.

class_weight [dict, list of dicts, "balanced" or None, default=None] Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y))

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

presort [bool, optional (default=False)] Whether to presort the data to speed up the finding of best splits in fitting. For the default settings of a decision tree on large datasets, setting this to true may slow down the training process. When using either a smaller dataset or a restricted depth, this may speed up the training.

Attributes

classes_ [array of shape = [n_classes] or a list of such arrays] The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

feature_importances_ [array of shape = [n_features]] Return the feature importances.

max_features_ [int,] The inferred value of max_features.

n_classes_ [int or list] The number of classes (for single output problems), or a list containing the number of classes for each output (for multi-output problems).

n_features_ [int] The number of features when fit is performed.

n_outputs_ [int] The number of outputs when fit is performed.

tree_ [Tree object] The underlying Tree object. Please refer to help(sklearn.tree._tree.Tree) for attributes of Tree object and *Understanding the decision tree structure* for basic usage of these attributes.

See also:

DecisionTreeRegressor

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data and `max_features=n_features`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

References

[Rb1ec977cd307-1], *[Rb1ec977cd307-2]*, *[Rb1ec977cd307-3]*, *[Rb1ec977cd307-4]*

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = DecisionTreeClassifier(random_state=0)
>>> iris = load_iris()
>>> cross_val_score(clf, iris.data, iris.target, cv=10)
...
...
array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
        0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
```

Methods

apply(self, X[, check_input])	Returns the index of the leaf that each sample is predicted as.
decision_path(self, X[, check_input])	Return the decision path in the tree
fit(self, X, y[, sample_weight, ...])	Build a decision tree classifier from the training set (X, y).
get_depth(self)	Returns the depth of the decision tree.
get_n_leaves(self)	Returns the number of leaves of the decision tree.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, check_input])	Predict class or regression value for X.
predict_log_proba(self, X)	Predict class log-probabilities of the input samples X.
predict_proba(self, X[, check_input])	Predict class probabilities of the input samples X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *criterion='gini'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *class_weight=None*, *presort=False*)

apply(*self*, *X*, *check_input=True*)
Returns the index of the leaf that each sample is predicted as.

New in version 0.17.

Parameters

X [array_like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

X_leaves [array_like, shape = [n_samples,]] For each datapoint x in X, return the index of the leaf x ends up in. Leaves are numbered within `[0; self.tree_.node_count)`, possibly with gaps in the numbering.

decision_path(*self*, *X*, *check_input=True*)
Return the decision path in the tree

New in version 0.18.

Parameters

X [array_like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

feature_importances_
Return the feature importances.

The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Returns

feature_importances_ [array, shape = [n_features]]

fit(*self*, *X*, *y*, *sample_weight=None*, *check_input=True*, *X_idx_sorted=None*)
Build a decision tree classifier from the training set (X, y).

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] The target values (class labels) as integers or strings.

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

X_idx_sorted [array-like, shape = [n_samples, n_features], optional] The indexes of the sorted training input samples. If many tree are grown on the same dataset, this allows the ordering to be cached between trees. If None, the data will be sorted here. Don't use this parameter unless you know what to do.

Returns

self [object]

get_depth (*self*)
Returns the depth of the decision tree.

The depth of a tree is the maximum distance between the root and any leaf.

get_n_leaves (*self*)
Returns the number of leaves of the decision tree.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*, *check_input=True*)
Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

y [array of shape = [n_samples] or [n_samples, n_outputs]] The predicted classes, or the predict values.

predict_log_proba (*self*, *X*)
Predict class log-probabilities of the input samples X.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

p [array of shape = [n_samples, n_classes], or a list of n_outputs] such arrays if n_outputs > 1. The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba (*self*, *X*, *check_input=True*)

Predict class probabilities of the input samples X.

The predicted class probability is the fraction of samples of the same class in a leaf.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool] Run check_array on X.

Returns

p [array of shape = [n_samples, n_classes], or a list of n_outputs] such arrays if n_outputs > 1. The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

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

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.tree.DecisionTreeClassifier`

- *Classifier comparison*
- *Plot the decision boundaries of a VotingClassifier*
- *Two-class AdaBoost*
- *Multi-class AdaBoosted Decision Trees*

- *Discrete versus Real AdaBoost*
- *Plot the decision surfaces of ensembles of trees on the iris dataset*
- *Demonstration of multi-metric evaluation on cross_val_score and GridSearchCV*
- *Plot the decision surface of a decision tree on the iris dataset*
- *Understanding the decision tree structure*

6.38.2 sklearn.tree.DecisionTreeRegressor

class sklearn.tree.**DecisionTreeRegressor**(*criterion='mse'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *presort=False*)

A decision tree regressor.

Read more in the *User Guide*.

Parameters

criterion [string, optional (default="mse")] The function to measure the quality of a split. Supported criteria are "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion and minimizes the L2 loss using the mean of each terminal node, "friedman_mse", which uses mean squared error with Friedman's improvement score for potential splits, and "mae" for the mean absolute error, which minimizes the L1 loss using the median of each terminal node.

New in version 0.18: Mean Absolute Error (MAE) criterion.

splitter [string, optional (default="best")] The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth [int or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, string or None, optional (default=None)] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

max_leaf_nodes [int or None, optional (default=None)] Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

presort [bool, optional (default=False)] Whether to presort the data to speed up the finding of best splits in fitting. For the default settings of a decision tree on large datasets, setting this to true may slow down the training process. When using either a smaller dataset or a restricted depth, this may speed up the training.

Attributes

feature_importances_ [array of shape = [n_features]] Return the feature importances.

max_features_ [int,] The inferred value of max_features.

n_features_ [int] The number of features when `fit` is performed.

n_outputs_ [int] The number of outputs when `fit` is performed.

tree_ [Tree object] The underlying Tree object. Please refer to `help(sklearn.tree._tree.Tree)` for attributes of Tree object and *Understanding the decision tree structure* for basic usage of these attributes.

See also:

DecisionTreeClassifier

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data and `max_features=n_features`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed.

References

[Ra37b7e3adb19-1], *[Ra37b7e3adb19-2]*, *[Ra37b7e3adb19-3]*, *[Ra37b7e3adb19-4]*

Examples

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.tree import DecisionTreeRegressor
>>> boston = load_boston()
>>> regressor = DecisionTreeRegressor(random_state=0)
>>> cross_val_score(regressor, boston.data, boston.target, cv=10)
...
...
array([ 0.61..., 0.57..., -0.34..., 0.41..., 0.75...,
        0.07..., 0.29..., 0.33..., -1.42..., -1.77...])
```

Methods

apply(self, X[, check_input])	Returns the index of the leaf that each sample is predicted as.
decision_path(self, X[, check_input])	Return the decision path in the tree
fit(self, X, y[, sample_weight, ...])	Build a decision tree regressor from the training set (X, y).
get_depth(self)	Returns the depth of the decision tree.

Continued on next page

Table 6.284 – continued from previous page

get_n_leaves(self)	Returns the number of leaves of the decision tree.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, check_input])	Predict class or regression value for X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *criterion='mse'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *presort=False*)

apply(*self*, *X*, *check_input=True*)

Returns the index of the leaf that each sample is predicted as.

New in version 0.17.

Parameters

X [array_like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

X_leaves [array_like, shape = [n_samples,]] For each datapoint x in X, return the index of the leaf x ends up in. Leaves are numbered within `[0; self.tree_.node_count)`, possibly with gaps in the numbering.

decision_path(*self*, *X*, *check_input=True*)

Return the decision path in the tree

New in version 0.18.

Parameters

X [array_like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

feature_importances_

Return the feature importances.

The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Returns

feature_importances_ [array, shape = [n_features]]

fit(*self*, *X*, *y*, *sample_weight=None*, *check_input=True*, *X_idx_sorted=None*)
Build a decision tree regressor from the training set (X, y).

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] The target values (real numbers). Use `dtype=np.float64` and `order='C'` for maximum efficiency.

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

X_idx_sorted [array-like, shape = [n_samples, n_features], optional] The indexes of the sorted training input samples. If many tree are grown on the same dataset, this allows the ordering to be cached between trees. If None, the data will be sorted here. Don't use this parameter unless you know what to do.

Returns

self [object]

get_depth(*self*)
Returns the depth of the decision tree.

The depth of a tree is the maximum distance between the root and any leaf.

get_n_leaves(*self*)
Returns the number of leaves of the decision tree.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, *X*, *check_input=True*)
Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

y [array of shape = [n_samples] or [n_samples, n_outputs]] The predicted classes, or the predict values.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

Examples using `sklearn.tree.DecisionTreeRegressor`

- *Decision Tree Regression with AdaBoost*
- *Single estimator versus bagging: bias-variance decomposition*
- *Imputing missing values with variants of IterativeImputer*
- *Using KBinsDiscretizer to discretize continuous features*
- *Decision Tree Regression*
- *Multi-output Decision Tree Regression*

6.38.3 sklearn.tree.ExtraTreeClassifier

class sklearn.tree.**ExtraTreeClassifier**(*criterion='gini'*, *splitter='random'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *class_weight=None*)

An extremely randomized tree classifier.

Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the `max_features` randomly selected features and the best split among those is chosen. When `max_features` is set 1, this amounts to building a totally random decision tree.

Warning: Extra-trees should only be used within ensemble methods.

Read more in the *User Guide*.

Parameters

criterion [string, optional (default="gini")] The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.

splitter [string, optional (default="random")] The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth [int or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.

- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

max_leaf_nodes [int or None, optional (default=None)] Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

class_weight [dict, list of dicts, "balanced" or None, default=None] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

Attributes

feature_importances_ Return the feature importances.

See also:

ExtraTreeRegressor, ***sklearn.ensemble.ExtraTreesClassifier***

sklearn.ensemble.ExtraTreesRegressor

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

References

[Rdd99a0224c6e-1]

Methods

apply(self, X[, check_input])	Returns the index of the leaf that each sample is predicted as.
decision_path(self, X[, check_input])	Return the decision path in the tree
fit(self, X, y[, sample_weight, . . .])	Build a decision tree classifier from the training set (X, y).
get_depth(self)	Returns the depth of the decision tree.
get_n_leaves(self)	Returns the number of leaves of the decision tree.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, check_input])	Predict class or regression value for X.
predict_log_proba(self, X)	Predict class log-probabilities of the input samples X.
predict_proba(self, X[, check_input])	Predict class probabilities of the input samples X.
score(self, X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *criterion='gini'*, *splitter='random'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *class_weight=None*)

apply(*self*, *X*, *check_input=True*)

Returns the index of the leaf that each sample is predicted as.

New in version 0.17.

Parameters

X [array_like or sparse matrix, shape = [n_samples, n_features]] The input samples. Inter-

nally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

X_leaves [array_like, shape = [n_samples,]] For each datapoint x in X, return the index of the leaf x ends up in. Leaves are numbered within `[0; self.tree_.node_count)`, possibly with gaps in the numbering.

decision_path (*self*, *X*, *check_input=True*)

Return the decision path in the tree

New in version 0.18.

Parameters

X [array_like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

feature_importances_

Return the feature importances.

The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Returns

feature_importances_ [array, shape = [n_features]]

fit (*self*, *X*, *y*, *sample_weight=None*, *check_input=True*, *X_idx_sorted=None*)

Build a decision tree classifier from the training set (X, y).

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] The target values (class labels) as integers or strings.

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

X_idx_sorted [array-like, shape = [n_samples, n_features], optional] The indexes of the sorted training input samples. If many tree are grown on the same dataset, this allows the

ordering to be cached between trees. If None, the data will be sorted here. Don't use this parameter unless you know what to do.

Returns

self [object]

get_depth (*self*)
Returns the depth of the decision tree.

The depth of a tree is the maximum distance between the root and any leaf.

get_n_leaves (*self*)
Returns the number of leaves of the decision tree.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*, *check_input=True*)
Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

y [array of shape = [n_samples] or [n_samples, n_outputs]] The predicted classes, or the predict values.

predict_log_proba (*self*, *X*)
Predict class log-probabilities of the input samples X.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

Returns

p [array of shape = [n_samples, n_classes], or a list of n_outputs] such arrays if n_outputs > 1. The class log-probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

predict_proba (*self*, *X*, *check_input=True*)
Predict class probabilities of the input samples X.

The predicted class probability is the fraction of samples of the same class in a leaf.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [bool] Run check_array on X.

Returns

p [array of shape = [n_samples, n_classes], or a list of n_outputs] such arrays if n_outputs > 1. The class probabilities of the input samples. The order of the classes corresponds to that in the attribute *classes_*.

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

Returns 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, shape = (n_samples, n_features)] Test samples.

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

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

set_params (*self*, ***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.

Returns

self

6.38.4 sklearn.tree.ExtraTreeRegressor

class sklearn.tree.**ExtraTreeRegressor**(*criterion='mse'*, *splitter='random'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *random_state=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *max_leaf_nodes=None*)

An extremely randomized tree regressor.

Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the `max_features` randomly selected features and the best split among those is chosen. When `max_features` is set 1, this amounts to building a totally random decision tree.

Warning: Extra-trees should only be used within ensemble methods.

Read more in the *User Guide*.

Parameters

criterion [string, optional (default="mse")] The function to measure the quality of a split. Supported criteria are "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion, and "mae" for the mean absolute error.

New in version 0.18: Mean Absolute Error (MAE) criterion.

splitter [string, optional (default="random")] The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

max_depth [int or None, optional (default=None)] The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int, float, optional (default=2)] The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf [int, float, optional (default=1)] The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version 0.18: Added float values for fractions.

min_weight_fraction_leaf [float, optional (default=0.)] The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features [int, float, string or None, optional (default="auto")] The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

min_impurity_decrease [float, optional (default=0.)] A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where `N` is the total number of samples, `N_t` is the number of samples at the current node, `N_t_L` is the number of samples in the left child, and `N_t_R` is the number of samples in the right child.

`N`, `N_t`, `N_t_R` and `N_t_L` all refer to the weighted sum, if `sample_weight` is passed.

New in version 0.19.

min_impurity_split [float, (default=1e-7)] Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Deprecated since version 0.19: `min_impurity_split` has been deprecated in favor of `min_impurity_decrease` in 0.19. The default value of `min_impurity_split` will change from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use `min_impurity_decrease` instead.

max_leaf_nodes [int or None, optional (default=None)] Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

Attributes

feature_importances_ Return the feature importances.

See also:

ExtraTreeClassifier*, *sklearn.ensemble.ExtraTreesClassifier

sklearn.ensemble.ExtraTreesRegressor

Notes

The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

References

[R4939d63d5a49-1]

Methods

apply(self, X[, check_input])	Returns the index of the leaf that each sample is predicted as.
decision_path(self, X[, check_input])	Return the decision path in the tree
fit(self, X, y[, sample_weight, ...])	Build a decision tree regressor from the training set (X, y).
get_depth(self)	Returns the depth of the decision tree.
get_n_leaves(self)	Returns the number of leaves of the decision tree.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, check_input])	Predict class or regression value for X.
score(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params(self, **params)	Set the parameters of this estimator.

__init__(*self*, *criterion='mse'*, *splitter='random'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *random_state=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *max_leaf_nodes=None*)

apply(*self*, *X*, *check_input=True*)

Returns the index of the leaf that each sample is predicted as.

New in version 0.17.

Parameters

X [array_like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

X_leaves [array_like, shape = [n_samples,]] For each datapoint x in X, return the index of the leaf x ends up in. Leaves are numbered within `[0; self.tree_.node_count)`, possibly with gaps in the numbering.

decision_path(*self*, *X*, *check_input=True*)

Return the decision path in the tree

New in version 0.18.

Parameters

X [array_like or sparse matrix, shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

indicator [sparse csr array, shape = [n_samples, n_nodes]] Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes.

feature_importances_

Return the feature importances.

The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Returns

feature_importances_ [array, shape = [n_features]]

fit (*self*, *X*, *y*, *sample_weight=None*, *check_input=True*, *X_idx_sorted=None*)
Build a decision tree regressor from the training set (X, y).

Parameters

X [array-like or sparse matrix, shape = [n_samples, n_features]] The training input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csc_matrix`.

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] The target values (real numbers). Use `dtype=np.float64` and `order='C'` for maximum efficiency.

sample_weight [array-like, shape = [n_samples] or None] Sample weights. If None, then samples are equally weighted. Splits that would create child nodes with net zero or negative weight are ignored while searching for a split in each node.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

X_idx_sorted [array-like, shape = [n_samples, n_features], optional] The indexes of the sorted training input samples. If many tree are grown on the same dataset, this allows the ordering to be cached between trees. If None, the data will be sorted here. Don't use this parameter unless you know what to do.

Returns

self [object]

get_depth (*self*)
Returns the depth of the decision tree.

The depth of a tree is the maximum distance between the root and any leaf.

get_n_leaves (*self*)
Returns the number of leaves of the decision tree.

get_params (*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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 (*self*, *X*, *check_input=True*)
Predict class or regression value for X.

For a classification model, the predicted class for each sample in X is returned. For a regression model, the predicted value based on X is returned.

Parameters

X [array-like or sparse matrix of shape = [n_samples, n_features]] The input samples. Internally, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input [boolean, (default=True)] Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns

y [array of shape = [n_samples] or [n_samples, n_outputs]] The predicted classes, or the predict values.

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

Returns 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, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix 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, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

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

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with *metrics.r2_score*. This will influence the `score` method of all the multioutput regressors (except for *multioutput.MultiOutputRegressor*). To specify the default value manually and avoid the warning, please either call *metrics.r2_score* directly or make a custom scorer with *metrics.make_scorer* (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***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.

Returns

self

tree.export_graphviz(decision_tree[, ...])	Export a decision tree in DOT format.
tree.plot_tree(decision_tree[, max_depth, ...])	Plot a decision tree.
tree.export_text(decision_tree[, ...])	Build a text report showing the rules of a decision tree.

6.38.5 sklearn.tree.export_graphviz

sklearn.tree.**export_graphviz**(*decision_tree*, *out_file=None*, *max_depth=None*, *feature_names=None*, *class_names=None*, *label='all'*, *filled=False*, *leaves_parallel=False*, *impurity=True*, *node_ids=False*, *proportion=False*, *rotate=False*, *rounded=False*, *special_characters=False*, *precision=3*)

Export a decision tree in DOT format.

This function generates a GraphViz representation of the decision tree, which is then written into `out_file`. Once exported, graphical renderings can be generated using, for example:

```
$ dot -Tps tree.dot -o tree.ps      (PostScript format)
$ dot -Tpng tree.dot -o tree.png    (PNG format)
```

The sample counts that are shown are weighted with any sample_weights that might be present.

Read more in the *User Guide*.

Parameters

decision_tree [decision tree classifier] The decision tree to be exported to GraphViz.

out_file [file object or string, optional (default=None)] Handle or name of the output file. If `None`, the result is returned as a string.

Changed in version 0.20: Default of out_file changed from "tree.dot" to None.

max_depth [int, optional (default=None)] The maximum depth of the representation. If None, the tree is fully generated.

feature_names [list of strings, optional (default=None)] Names of each of the features.

class_names [list of strings, bool or None, optional (default=None)] Names of each of the target classes in ascending numerical order. Only relevant for classification and not supported for multi-output. If `True`, shows a symbolic representation of the class name.

label [{'all', 'root', 'none'}, optional (default='all')] Whether to show informative labels for impurity, etc. Options include 'all' to show at every node, 'root' to show only at the top root node, or 'none' to not show at any node.

filled [bool, optional (default=False)] When set to `True`, paint nodes to indicate majority class for classification, extremity of values for regression, or purity of node for multi-output.

leaves_parallel [bool, optional (default=False)] When set to `True`, draw all leaf nodes at the bottom of the tree.

impurity [bool, optional (default=True)] When set to `True`, show the impurity at each node.

node_ids [bool, optional (default=False)] When set to `True`, show the ID number on each node.

proportion [bool, optional (default=False)] When set to `True`, change the display of 'values' and/or 'samples' to be proportions and percentages respectively.

rotate [bool, optional (default=False)] When set to `True`, orient tree left to right rather than top-down.

rounded [bool, optional (default=False)] When set to `True`, draw node boxes with rounded corners and use Helvetica fonts instead of Times-Roman.

special_characters [bool, optional (default=False)] When set to `False`, ignore special characters for PostScript compatibility.

precision [int, optional (default=3)] Number of digits of precision for floating point in the values of impurity, threshold and value attributes of each node.

Returns

dot_data [string] String representation of the input tree in GraphViz dot format. Only returned if `out_file` is None.

New in version 0.18.

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
```

```
>>> clf = tree.DecisionTreeClassifier()
>>> iris = load_iris()
```

```
>>> clf = clf.fit(iris.data, iris.target)
>>> tree.export_graphviz(clf)
'digraph Tree {...
```

6.38.6 sklearn.tree.plot_tree

sklearn.tree.**plot_tree**(*decision_tree*, *max_depth=None*, *feature_names=None*, *class_names=None*, *label='all'*, *filled=False*, *impurity=True*, *node_ids=False*, *proportion=False*, *rotate=False*, *rounded=False*, *precision=3*, *ax=None*, *fontsize=None*)

Plot a decision tree.

The sample counts that are shown are weighted with any sample_weights that might be present. This function requires matplotlib, and works best with matplotlib >= 1.5.

The visualization is fit automatically to the size of the axis. Use the `figsize` or `dpi` arguments of `plt.figure` to control the size of the rendering.

Read more in the *User Guide*.

New in version 0.21.

Parameters

decision_tree [decision tree regressor or classifier] The decision tree to be exported to GraphViz.

max_depth [int, optional (default=None)] The maximum depth of the representation. If None, the tree is fully generated.

feature_names [list of strings, optional (default=None)] Names of each of the features.

class_names [list of strings, bool or None, optional (default=None)] Names of each of the target classes in ascending numerical order. Only relevant for classification and not supported for multi-output. If `True`, shows a symbolic representation of the class name.

label [{'all', 'root', 'none'}, optional (default='all')] Whether to show informative labels for impurity, etc. Options include 'all' to show at every node, 'root' to show only at the top root node, or 'none' to not show at any node.

filled [bool, optional (default=False)] When set to `True`, paint nodes to indicate majority class for classification, extremity of values for regression, or purity of node for multi-output.

impurity [bool, optional (default=True)] When set to `True`, show the impurity at each node.

node_ids [bool, optional (default=False)] When set to `True`, show the ID number on each node.

proportion [bool, optional (default=False)] When set to `True`, change the display of 'values' and/or 'samples' to be proportions and percentages respectively.

rotate [bool, optional (default=False)] When set to `True`, orient tree left to right rather than top-down.

rounded [bool, optional (default=False)] When set to `True`, draw node boxes with rounded corners and use Helvetica fonts instead of Times-Roman.

precision [int, optional (default=3)] Number of digits of precision for floating point in the values of impurity, threshold and value attributes of each node.

ax [matplotlib axis, optional (default=None)] Axes to plot to. If None, use current axis. Any previous content is cleared.

fontsize [int, optional (default=None)] Size of text font. If None, determined automatically to fit figure.

Returns

annotations [list of artists] List containing the artists for the annotation boxes making up the tree.

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
```

```
>>> clf = tree.DecisionTreeClassifier(random_state=0)
>>> iris = load_iris()
```

```
>>> clf = clf.fit(iris.data, iris.target)
>>> tree.plot_tree(clf)
[Text(251.5,345.217,'X[3] <= 0.8...
```

Examples using `sklearn.tree.plot_tree`

- *Plot the decision surface of a decision tree on the iris dataset*

6.38.7 `sklearn.tree`.export_text

`sklearn.tree.`**`export_text`**(*decision_tree*, *feature_names=None*, *max_depth=10*, *spacing=3*, *decimals=2*, *show_weights=False*)

Build a text report showing the rules of a decision tree.

Note that backwards compatibility may not be supported.

Parameters

decision_tree [object] The decision tree estimator to be exported. It can be an instance of DecisionTreeClassifier or DecisionTreeRegressor.

feature_names [list, optional (default=None)] A list of length n_features containing the feature names. If None generic names will be used ("feature_0", "feature_1", ...).

max_depth [int, optional (default=10)] Only the first max_depth levels of the tree are exported. Truncated branches will be marked with "...".

spacing [int, optional (default=3)] Number of spaces between edges. The higher it is, the wider the result.

decimals [int, optional (default=2)] Number of decimal digits to display.

show_weights [bool, optional (default=False)] If true the classification weights will be exported on each leaf. The classification weights are the number of samples each class.

Returns

report [string] Text summary of all the rules in the decision tree.

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.tree.export import export_text
>>> iris = load_iris()
>>> X = iris['data']
>>> y = iris['target']
>>> decision_tree = DecisionTreeClassifier(random_state=0, max_depth=2)
>>> decision_tree = decision_tree.fit(X, y)
>>> r = export_text(decision_tree, feature_names=iris['feature_names'])
>>> print(r)
|--- petal width (cm) <= 0.80
|   |--- class: 0
|--- petal width (cm) >  0.80
|   |--- petal width (cm) <= 1.75
|   |   |--- class: 1
|   |--- petal width (cm) >  1.75
|   |   |--- class: 2
...
```

6.39 `sklearn.utils`: Utilities

The *sklearn.utils* module includes various utilities.

Developer guide: See the *Utilities for Developers* page for further details.

utils.arrayfuncs.cholesky_delete(L, go_out)	
utils.arrayfuncs.min_pos()	Find the minimum value of an array over positive values
utils.as_float_array(X[, copy, force_all_finite])	Converts an array-like to an array of floats.
utils.assert_all_finite(X[, allow_nan])	Throw a ValueError if X contains NaN or infinity.
utils.check_X_y(X, y[, accept_sparse, ...])	Input validation for standard estimators.

Continued on next page

Table 6.288 – continued from previous page

utils.check_array(array[, accept_sparse, ...])	Input validation on an array, list, sparse matrix or similar.
utils.check_scalar(x, name, target_type[, ...])	Validate scalar parameters type and value.
utils.check_consistent_length(*arrays)	Check that all arrays have consistent first dimensions.
utils.check_random_state(seed)	Turn seed into a np.random.RandomState instance
utils.class_weight.compute_class_weight(...)	Estimate class weights for unbalanced datasets.
utils.class_weight.compute_sample_weight(...)	Estimate sample weights by class for unbalanced datasets.
utils.deprecated([extra])	Decorator to mark a function or class as deprecated.
utils.estimator_checks.check_estimator(Estimator)	Check if estimator adheres to scikit-learn conventions.
utils.extmath.safe_sparse_dot(a, b[, ...])	Dot product that handle the sparse matrix case correctly
utils.extmath.randomized_range_finder(A, ...)	Computes an orthonormal matrix whose range approximates the range of A.
utils.extmath.randomized_svd(M, n_components)	Computes a truncated randomized SVD
utils.extmath.fast_logdet(A)	Compute log(det(A)) for A symmetric
utils.extmath.density(w, **kwargs)	Compute density of a sparse vector
utils.extmath.weighted_mode(a, w[, axis])	Returns an array of the weighted modal (most common) value in a
utils.gen_even_slices(n, n_packs[, n_samples])	Generator to create n_packs slices going up to n.
utils.graph.single_source_shortest_path_length(...)	Return the shortest path length from source to all reachable nodes.
utils.graph_shortest_path.graph_shortest_path()	Perform a shortest-path graph search on a positive directed or undirected graph.
utils.indexable(*iterables)	Make arrays indexable for cross-validation.
utils.metaestimators.if_delegate_has_method(...)	Create a decorator for methods that are delegated to a sub-estimator
utils.multiclass.type_of_target(y)	Determine the type of data indicated by the target.
utils.multiclass.is_multilabel(y)	Check if y is in a multilabel format.
utils.multiclass.unique_labels(*ys)	Extract an ordered array of unique labels
utils.murmurhash3_32()	Compute the 32bit murmurhash3 of key at seed.
utils.resample(*arrays, **options)	Resample arrays or sparse matrices in a consistent way
utils.safe_indexing(X, indices)	Return items or rows from X using indices.
utils.safe_mask(X, mask)	Return a mask which is safe to use on X.
utils.safe_sqr(X[, copy])	Element wise squaring of array-likes and sparse matrices.
utils.shuffle(*arrays, **options)	Shuffle arrays or sparse matrices in a consistent way
utils.sparsefuncs.incr_mean_variance_axis(X, ...)	Compute incremental mean and variance along an axix on a CSR or CSC matrix.
utils.sparsefuncs.inplace_column_scale(X, scale)	Inplace column scaling of a CSC/CSR matrix.
utils.sparsefuncs.inplace_row_scale(X, scale)	Inplace row scaling of a CSR or CSC matrix.
utils.sparsefuncs.inplace_swap_row(X, m, n)	Swaps two rows of a CSC/CSR matrix in-place.
utils.sparsefuncs.inplace_swap_column(X, m, n)	Swaps two columns of a CSC/CSR matrix in-place.
utils.sparsefuncs.mean_variance_axis(X, axis)	Compute mean and variance along an axix on a CSR or CSC matrix
utils.sparsefuncs.inplace_csr_column_scale(X, ...)	Inplace column scaling of a CSR matrix.

Continued on next page

Table 6.288 – continued from previous page

utils.sparsefuncs_fast.inplace_csr_row_normalize_l1()	Inplace row normalize using the l1 norm
utils.sparsefuncs_fast.inplace_csr_row_normalize_l2()	Inplace row normalize using the l2 norm
utils.random.sample_without_replacement()	Sample integers without replacement.
utils.validation.check_is_fitted(estimator, ...)	Perform is_fitted validation for estimator.
utils.validation.check_memory(memory)	Check that `memory` is joblib.Memory-like.
utils.validation.check_symmetric(array[, ...])	Make sure that array is 2D, square and symmetric.
utils.validation.column_or_1d(y[, warn])	Ravel column or 1d numpy array, else raises an error
utils.validation.has_fit_parameter(...)	Checks whether the estimator's fit method supports the given parameter.
utils.testing.assert_in(member, container[, msg])	Just like self.assertTrue(a in b), but with a nicer default message.
utils.testing.assert_not_in(member, container)	Just like self.assertTrue(a not in b), but with a nicer default message.
utils.testing.assert_raise_message(...)	Helper function to test the message raised in an exception.
utils.testing.all_estimators([...])	Get a list of all estimators from sklearn.

6.39.1 sklearn.utils.arrayfuncs.cholesky_delete

6.39.2 sklearn.utils.arrayfuncs.min_pos

sklearn.utils.arrayfuncs.**min_pos**()

Find the minimum value of an array over positive values

Returns a huge value if none of the values are positive

6.39.3 sklearn.utils.as_float_array

sklearn.utils.**as_float_array**(*X*, *copy=True*, *force_all_finite=True*)

Converts an array-like to an array of floats.

The new dtype will be np.float32 or np.float64, depending on the original type. The function can create a copy or modify the argument depending on the argument copy.

Parameters

X [{array-like, sparse matrix}]

copy [bool, optional] If True, a copy of X will be created. If False, a copy may still be returned if X's dtype is not a floating point type.

force_all_finite [boolean or 'allow-nan', (default=True)] Whether to raise an error on np.inf and np.nan in X. The possibilities are:

- True: Force all values of X to be finite.
- False: accept both np.inf and np.nan in X.
- 'allow-nan': accept only np.nan values in X. Values cannot be infinite.

New in version 0.20: `force_all_finite` accepts the string `'allow-nan'`.

Returns

XT [{array, sparse matrix}] An array of type np.float

6.39.4 sklearn.utils.assert_all_finite

sklearn.utils.**assert_all_finite**(*X*, *allow_nan=False*)

Throw a ValueError if X contains NaN or infinity.

Parameters

X [array or sparse matrix]

allow_nan [bool]

6.39.5 sklearn.utils.check_X_y

sklearn.utils.**check_X_y**(*X*, *y*, *accept_sparse=False*, *accept_large_sparse=True*, *dtype='numeric'*, *order=None*, *copy=False*, *force_all_finite=True*, *ensure_2d=True*, *allow_nd=False*, *multi_output=False*, *ensure_min_samples=1*, *ensure_min_features=1*, *y_numeric=False*, *warn_on_dtype=None*, *estimator=None*)

Input validation for standard estimators.

Checks X and y for consistent length, enforces X to be 2D and y 1D. By default, X is checked to be non-empty and containing only finite values. Standard input checks are also applied to y, such as checking that y does not have np.nan or np.inf targets. For multi-label y, set multi_output=True to allow 2D and sparse y. If the dtype of X is object, attempt converting to float, raising on failure.

Parameters

X [nd-array, list or sparse matrix] Input data.

y [nd-array, list or sparse matrix] Labels.

accept_sparse [string, boolean or list of string (default=False)] String[s] representing allowed sparse matrix formats, such as 'csc', 'csr', etc. If the input is sparse but not in the allowed format, it will be converted to the first listed format. True allows the input to be any format. False means that a sparse matrix input will raise an error.

accept_large_sparse [bool (default=True)] If a CSR, CSC, COO or BSR sparse matrix is supplied and accepted by accept_sparse, accept_large_sparse will cause it to be accepted only if its indices are stored with a 32-bit dtype.

New in version 0.20.

dtype [string, type, list of types or None (default="numeric")] Data type of result. If None, the dtype of the input is preserved. If "numeric", dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.

order ['F', 'C' or None (default=None)] Whether an array will be forced to be fortran or c-style.

copy [boolean (default=False)] Whether a forced copy will be triggered. If copy=False, a copy might be triggered by a conversion.

force_all_finite [boolean or 'allow-nan', (default=True)] Whether to raise an error on np.inf and np.nan in X. This parameter does not influence whether y can have np.inf or np.nan values. The possibilities are:

- True: Force all values of X to be finite.

- False: accept both np.inf and np.nan in X.
- 'allow-nan': accept only np.nan values in X. Values cannot be infinite.

New in version 0.20: `force_all_finite` accepts the string `'allow-nan'`.

ensure_2d [boolean (default=True)] Whether to raise a value error if X is not 2D.

allow_nd [boolean (default=False)] Whether to allow X.ndim > 2.

multi_output [boolean (default=False)] Whether to allow 2D y (array or sparse matrix). If false, y will be validated as a vector. y cannot have np.nan or np.inf values if multi_output=True.

ensure_min_samples [int (default=1)] Make sure that X has a minimum number of samples in its first axis (rows for a 2D array).

ensure_min_features [int (default=1)] Make sure that the 2D array has some minimum number of features (columns). The default value of 1 rejects empty datasets. This check is only enforced when X has effectively 2 dimensions or is originally 1D and `ensure_2d` is True. Setting to 0 disables this check.

y_numeric [boolean (default=False)] Whether to ensure that y has a numeric type. If dtype of y is object, it is converted to float64. Should only be used for regression algorithms.

warn_on_dtype [boolean or None, optional (default=None)] Raise DataConversionWarning if the dtype of the input data structure does not match the requested dtype, causing a memory copy.

Deprecated since version 0.21: `warn_on_dtype` is deprecated in version 0.21 and will be removed in 0.23.

estimator [str or estimator instance (default=None)] If passed, include the name of the estimator in warning messages.

Returns

X_converted [object] The converted and validated X.

y_converted [object] The converted and validated y.

6.39.6 sklearn.utils.check_array

sklearn.utils.**check_array**(*array*, *accept_sparse=False*, *accept_large_sparse=True*, *dtype='numeric'*, *order=None*, *copy=False*, *force_all_finite=True*, *ensure_2d=True*, *allow_nd=False*, *ensure_min_samples=1*, *ensure_min_features=1*, *warn_on_dtype=None*, *estimator=None*)

Input validation on an array, list, sparse matrix or similar.

By default, the input is checked to be a non-empty 2D array containing only finite values. If the dtype of the array is object, attempt converting to float, raising on failure.

Parameters

array [object] Input object to check / convert.

accept_sparse [string, boolean or list/tuple of strings (default=False)] String[s] representing allowed sparse matrix formats, such as 'csc', 'csr', etc. If the input is sparse but not in the allowed format, it will be converted to the first listed format. True allows the input to be any format. False means that a sparse matrix input will raise an error.

accept_large_sparse [bool (default=True)] If a CSR, CSC, COO or BSR sparse matrix is supplied and accepted by accept_sparse, accept_large_sparse=False will cause it to be accepted only if its indices are stored with a 32-bit dtype.

New in version 0.20.

dtype [string, type, list of types or None (default="numeric")] Data type of result. If None, the dtype of the input is preserved. If "numeric", dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.

order ['F', 'C' or None (default=None)] Whether an array will be forced to be fortran or c-style. When order is None (default), then if copy=False, nothing is ensured about the memory layout of the output array; otherwise (copy=True) the memory layout of the returned array is kept as close as possible to the original array.

copy [boolean (default=False)] Whether a forced copy will be triggered. If copy=False, a copy might be triggered by a conversion.

force_all_finite [boolean or 'allow-nan', (default=True)] Whether to raise an error on np.inf and np.nan in array. The possibilities are:

- True: Force all values of array to be finite.
- False: accept both np.inf and np.nan in array.
- 'allow-nan': accept only np.nan values in array. Values cannot be infinite.

For object dtyped data, only np.nan is checked and not np.inf.

New in version 0.20: `force_all_finite` accepts the string `'allow-nan'`.

ensure_2d [boolean (default=True)] Whether to raise a value error if array is not 2D.

allow_nd [boolean (default=False)] Whether to allow array.ndim > 2.

ensure_min_samples [int (default=1)] Make sure that the array has a minimum number of samples in its first axis (rows for a 2D array). Setting to 0 disables this check.

ensure_min_features [int (default=1)] Make sure that the 2D array has some minimum number of features (columns). The default value of 1 rejects empty datasets. This check is only enforced when the input data has effectively 2 dimensions or is originally 1D and `ensure_2d` is True. Setting to 0 disables this check.

warn_on_dtype [boolean or None, optional (default=None)] Raise DataConversionWarning if the dtype of the input data structure does not match the requested dtype, causing a memory copy.

Deprecated since version 0.21: `warn_on_dtype` is deprecated in version 0.21 and will be removed in 0.23.

estimator [str or estimator instance (default=None)] If passed, include the name of the estimator in warning messages.

Returns

array_converted [object] The converted and validated array.

6.39.7 `sklearn.utils`.check_scalar

`sklearn.utils.`**`check_scalar`**(*x*, *name*, *target_type*, *min_val=None*, *max_val=None*)

Validate scalar parameters type and value.

Parameters

x [object] The scalar parameter to validate.

name [str] The name of the parameter to be printed in error messages.

target_type [type or tuple] Acceptable data types for the parameter.

min_val [float or int, optional (default=None)] The minimum valid value the parameter can take. If None (default) it is implied that the parameter does not have a lower bound.

max_val [float or int, optional (default=None)] The maximum valid value the parameter can take. If None (default) it is implied that the parameter does not have an upper bound.

Raises

TypeError If the parameter's type does not match the desired type.

ValueError If the parameter's value violates the given bounds.

6.39.8 sklearn.utils.check_consistent_length

sklearn.utils.**check_consistent_length**(**arrays*)

Check that all arrays have consistent first dimensions.

Checks whether all objects in arrays have the same shape or length.

Parameters

***arrays** [list or tuple of input objects.] Objects that will be checked for consistent length.

6.39.9 sklearn.utils.check_random_state

sklearn.utils.**check_random_state**(*seed*)

Turn seed into a np.random.RandomState instance

Parameters

seed [None | int | instance of RandomState] If seed is None, return the RandomState singleton used by np.random. If seed is an int, return a new RandomState instance seeded with seed. If seed is already a RandomState instance, return it. Otherwise raise ValueError.

Examples using sklearn.utils.check_random_state

- *Isotonic Regression*
- *Face completion with a multi-output estimators*
- *Empirical evaluation of the impact of k-means initialization*
- *MNIST classfification using multinomial logistic + L1*
- *Manifold Learning methods on a severed sphere*
- *Scaling the regularization parameter for SVCs*

6.39.10 sklearn.utils.class_weight.compute_class_weight

sklearn.utils.class_weight.**compute_class_weight**(*class_weight*, *classes*, *y*)
Estimate class weights for unbalanced datasets.

Parameters

class_weight [dict, 'balanced' or None] If 'balanced', class weights will be given by `n_samples / (n_classes * np.bincount(y))`. If a dictionary is given, keys are classes and values are corresponding class weights. If None is given, the class weights will be uniform.

classes [ndarray] Array of the classes occurring in the data, as given by `np.unique(y_org)` with `y_org` the original class labels.

y [array-like, shape (n_samples,)] Array of original class labels per sample;

Returns

class_weight_vect [ndarray, shape (n_classes,)] Array with class_weight_vect[i] the weight for i-th class

References

The "balanced" heuristic is inspired by Logistic Regression in Rare Events Data, King, Zen, 2001.

6.39.11 sklearn.utils.class_weight.compute_sample_weight

sklearn.utils.class_weight.**compute_sample_weight**(*class_weight*, *y*, *indices=None*)
Estimate sample weights by class for unbalanced datasets.

Parameters

class_weight [dict, list of dicts, "balanced", or None, optional] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data: `n_samples / (n_classes * np.bincount(y))`.

For multi-output, the weights of each column of y will be multiplied.

y [array-like, shape = [n_samples] or [n_samples, n_outputs]] Array of original class labels per sample.

indices [array-like, shape (n_subsample,), or None] Array of indices to be used in a subsample. Can be of length less than n_samples in the case of a subsample, or equal to n_samples in the case of a bootstrap subsample with repeated indices. If None, the sample weight will be calculated over the full sample. Only "balanced" is supported for class_weight if this is provided.

Returns

sample_weight_vect [ndarray, shape (n_samples,)] Array with sample weights as applied to the original y

6.39.12 sklearn.utils.deprecated

sklearn.utils.**deprecated**(*extra=''*)

Decorator to mark a function or class as deprecated.

Issue a warning when the function is called/the class is instantiated and adds a warning to the docstring.

The optional extra argument will be appended to the deprecation message and the docstring. Note: to use this with the default value for extra, put in an empty of parentheses:

```
>>> from sklearn.utils import deprecated
>>> deprecated()
<sklearn.utils.deprecation.deprecated object at ...>
```

```
>>> @deprecated()
... def some_function(): pass
```

Parameters

extra [string] to be added to the deprecation messages

6.39.13 sklearn.utils.estimator_checks.check_estimator

sklearn.utils.estimator_checks.**check_estimator**(*Estimator*)

Check if estimator adheres to scikit-learn conventions.

This estimator will run an extensive test-suite for input validation, shapes, etc. Additional tests for classifiers, regressors, clustering or transformers will be run if the Estimator class inherits from the corresponding mixin from sklearn.base.

This test can be applied to classes or instances. Classes currently have some additional tests that related to construction, while passing instances allows the testing of multiple options.

Parameters

estimator [estimator object or class] Estimator to check. Estimator is a class object or instance.

6.39.14 sklearn.utils.extmath.safe_sparse_dot

sklearn.utils.extmath.**safe_sparse_dot**(*a*, *b*, *dense_output=False*)

Dot product that handle the sparse matrix case correctly

Uses BLAS GEMM as replacement for numpy.dot where possible to avoid unnecessary copies.

Parameters

a [array or sparse matrix]

b [array or sparse matrix]

dense_output [boolean, default False] When False, either `a` or `b` being sparse will yield sparse output. When True, output will always be an array.

Returns

dot_product [array or sparse matrix] sparse if `a` or `b` is sparse and `dense_output=False`.

6.39.15 sklearn.utils.extmath.randomized_range_finder

sklearn.utils.extmath.**randomized_range_finder**(*A*, *size*, *n_iter*, *power_iteration_normalizer='auto'*, *random_state=None*)

Computes an orthonormal matrix whose range approximates the range of A.

Parameters

A [2D array] The input data matrix

size [integer] Size of the return array

n_iter [integer] Number of power iterations used to stabilize the result

power_iteration_normalizer ['auto' (default), 'QR', 'LU', 'none'] Whether the power iterations are normalized with step-by-step QR factorization (the slowest but most accurate), 'none' (the fastest but numerically unstable when `n_iter` is large, e.g. typically 5 or larger), or 'LU' factorization (numerically stable but can lose slightly in accuracy). The 'auto' mode applies no normalization if `n_iter` <= 2 and switches to LU otherwise.

New in version 0.18.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. 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`.

Returns

Q [2D array] A (size x size) projection matrix, the range of which approximates well the range of the input matrix A.

Notes

Follows Algorithm 4.3 of Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 (arXiv:909) https://arxiv.org/pdf/0909.4061.pdf

An implementation of a randomized algorithm for principal component analysis A. Szlam et al. 2014

6.39.16 sklearn.utils.extmath.randomized_svd

sklearn.utils.extmath.**randomized_svd**(*M*, *n_components*, *n_oversamples=10*, *n_iter='auto'*, *power_iteration_normalizer='auto'*, *transpose='auto'*, *flip_sign=True*, *random_state=0*)

Computes a truncated randomized SVD

Parameters

M [ndarray or sparse matrix] Matrix to decompose

n_components [int] Number of singular values and vectors to extract.

n_oversamples [int (default is 10)] Additional number of random vectors to sample the range of M so as to ensure proper conditioning. The total number of random vectors used to find

the range of M is n_components + n_oversamples. Smaller number can improve speed but can negatively impact the quality of approximation of singular vectors and singular values.

n_iter [int or 'auto' (default is 'auto')] Number of power iterations. It can be used to deal with very noisy problems. When 'auto', it is set to 4, unless *n_components* is small (< .1 * min(X.shape)) `n_iter` in which case is set to 7. This improves precision with few components.

Changed in version 0.18.

power_iteration_normalizer ['auto' (default), 'QR', 'LU', 'none'] Whether the power iterations are normalized with step-by-step QR factorization (the slowest but most accurate), 'none' (the fastest but numerically unstable when `n_iter` is large, e.g. typically 5 or larger), or 'LU' factorization (numerically stable but can lose slightly in accuracy). The 'auto' mode applies no normalization if `n_iter` <= 2 and switches to LU otherwise.

New in version 0.18.

transpose [True, False or 'auto' (default)] Whether the algorithm should be applied to M.T instead of M. The result should approximately be the same. The 'auto' mode will trigger the transposition if M.shape[1] > M.shape[0] since this implementation of randomized SVD tend to be a little faster in that case.

Changed in version 0.18.

flip_sign [boolean, (True by default)] The output of a singular value decomposition is only unique up to a permutation of the signs of the singular vectors. If `flip_sign` is set to `True`, the sign ambiguity is resolved by making the largest loadings for each component in the left singular vectors positive.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. 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`.

Notes

This algorithm finds a (usually very good) approximate truncated singular value decomposition using randomization to speed up the computations. It is particularly fast on large matrices on which you wish to extract only a small number of components. In order to obtain further speed up, `n_iter` can be set <=2 (at the cost of loss of precision).

References

- Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 https://arxiv.org/abs/0909.4061
- A randomized algorithm for the decomposition of matrices Per-Gunnar Martinsson, Vladimir Rokhlin and Mark Tygert
- An implementation of a randomized algorithm for principal component analysis A. Szlam et al. 2014

6.39.17 sklearn.utils.extmath.fast_logdet

sklearn.utils.extmath.**fast_logdet**(*A*)

Compute log(det(A)) for A symmetric

Equivalent to : np.log(nl.det(A)) but more robust. It returns -Inf if det(A) is non positive or is not defined.

Parameters

A [array_like] The matrix

6.39.18 sklearn.utils.extmath.density

sklearn.utils.extmath.**density**(*w*, ***kwargs*)

Compute density of a sparse vector

Parameters

w [array_like] The sparse vector

Returns

float The density of w, between 0 and 1

Examples using sklearn.utils.extmath.density

- *Classification of text documents using sparse features*

6.39.19 sklearn.utils.extmath.weighted_mode

sklearn.utils.extmath.**weighted_mode**(*a*, *w*, *axis=0*)

Returns an array of the weighted modal (most common) value in a

If there is more than one such value, only the first is returned. The bin-count for the modal bins is also returned.

This is an extension of the algorithm in scipy.stats.mode.

Parameters

a [array_like] n-dimensional array of which to find mode(s).

w [array_like] n-dimensional array of weights for each value

axis [int, optional] Axis along which to operate. Default is 0, i.e. the first axis.

Returns

vals [ndarray] Array of modal values.

score [ndarray] Array of weighted counts for each mode.

See also:

scipy.stats.mode

Examples

```
>>> from sklearn.utils.extmath import weighted_mode
>>> x = [4, 1, 4, 2, 4, 2]
>>> weights = [1, 1, 1, 1, 1, 1]
>>> weighted_mode(x, weights)
(array([4.]), array([3.]))
```

The value 4 appears three times: with uniform weights, the result is simply the mode of the distribution.

```
>>> weights = [1, 3, 0.5, 1.5, 1, 2]  # deweight the 4's
>>> weighted_mode(x, weights)
(array([2.]), array([3.5]))
```

The value 2 has the highest score: it appears twice with weights of 1.5 and 2: the sum of these is 3.5.

6.39.20 sklearn.utils.gen_even_slices

sklearn.utils.**gen_even_slices**(*n*, *n_packs*, *n_samples=None*)

Generator to create n_packs slices going up to n.

Parameters

n [int]

n_packs [int] Number of slices to generate.

n_samples [int or None (default = None)] Number of samples. Pass n_samples when the slices are to be used for sparse matrix indexing; slicing off-the-end raises an exception, while it works for NumPy arrays.

Yields

slice

Examples

```
>>> from sklearn.utils import gen_even_slices
>>> list(gen_even_slices(10, 1))
[slice(0, 10, None)]
>>> list(gen_even_slices(10, 10))
[slice(0, 1, None), slice(1, 2, None), ..., slice(9, 10, None)]
>>> list(gen_even_slices(10, 5))
[slice(0, 2, None), slice(2, 4, None), ..., slice(8, 10, None)]
>>> list(gen_even_slices(10, 3))
[slice(0, 4, None), slice(4, 7, None), slice(7, 10, None)]
```

6.39.21 sklearn.utils.graph.single_source_shortest_path_length

sklearn.utils.graph.**single_source_shortest_path_length**(*graph*, *source*, *cutoff=None*)

Return the shortest path length from source to all reachable nodes.

Returns a dictionary of shortest path lengths keyed by target.

Parameters

graph [sparse matrix or 2D array (preferably LIL matrix)] Adjacency matrix of the graph

source [integer] Starting node for path

cutoff [integer, optional] Depth to stop the search - only paths of length <= cutoff are returned.

Examples

```
>>> from sklearn.utils.graph import single_source_shortest_path_length
>>> import numpy as np
>>> graph = np.array([[ 0, 1, 0, 0],
...                   [ 1, 0, 1, 0],
...                   [ 0, 1, 0, 1],
...                   [ 0, 0, 1, 0]])
>>> list(sorted(single_source_shortest_path_length(graph, 0).items()))
[(0, 0), (1, 1), (2, 2), (3, 3)]
>>> graph = np.ones((6, 6))
>>> list(sorted(single_source_shortest_path_length(graph, 2).items()))
[(0, 1), (1, 1), (2, 0), (3, 1), (4, 1), (5, 1)]
```

6.39.22 sklearn.utils.graph_shortest_path.graph_shortest_path

sklearn.utils.graph_shortest_path.**graph_shortest_path**()

Perform a shortest-path graph search on a positive directed or undirected graph.

Parameters

dist_matrix [arraylike or sparse matrix, shape = (N,N)] Array of positive distances. If vertex i is connected to vertex j, then dist_matrix[i,j] gives the distance between the vertices. If vertex i is not connected to vertex j, then dist_matrix[i,j] = 0

directed [boolean] if True, then find the shortest path on a directed graph: only progress from a point to its neighbors, not the other way around. if False, then find the shortest path on an undirected graph: the algorithm can progress from a point to its neighbors and vice versa.

method [string ['auto'|'FW'|'D']] method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]

Returns

G [np.ndarray, float, shape = [N,N]] G[i,j] gives the shortest distance from point i to point j along the graph.

Notes

As currently implemented, Dijkstra's algorithm does not work for graphs with direction-dependent distances when directed == False. i.e., if dist_matrix[i,j] and dist_matrix[j,i] are not equal and both are nonzero, method='D' will not necessarily yield the correct result.

Also, these routines have not been tested for graphs with negative distances. Negative distances can lead to infinite cycles that must be handled by specialized algorithms.

6.39.23 sklearn.utils.indexable

sklearn.utils.**indexable**(**iterables*)

Make arrays indexable for cross-validation.

Checks consistent length, passes through None, and ensures that everything can be indexed by converting sparse matrices to csr and converting non-interable objects to arrays.

Parameters

***iterables** [lists, dataframes, arrays, sparse matrices] List of objects to ensure sliceability.

6.39.24 sklearn.utils.metaestimators.if_delegate_has_method

sklearn.utils.metaestimators.**if_delegate_has_method**(*delegate*)

Create a decorator for methods that are delegated to a sub-estimator

This enables ducktyping by hasattr returning True according to the sub-estimator.

Parameters

delegate [string, list of strings or tuple of strings] Name of the sub-estimator that can be accessed as an attribute of the base object. If a list or a tuple of names are provided, the first sub-estimator that is an attribute of the base object will be used.

Examples using sklearn.utils.metaestimators.if_delegate_has_method

- *Inductive Clustering*

6.39.25 sklearn.utils.multiclass.type_of_target

sklearn.utils.multiclass.**type_of_target**(*y*)

Determine the type of data indicated by the target.

Note that this type is the most specific type that can be inferred. For example:

- `binary` is more specific but compatible with `multiclass`.
- `multiclass` of integers is more specific but compatible with `continuous`.
- `multilabel-indicator` is more specific but compatible with `multiclass-multioutput`.

Parameters

y [array-like]

Returns

target_type [string] One of:

- 'continuous': *y* is an array-like of floats that are not all integers, and is 1d or a column vector.
- 'continuous-multioutput': *y* is a 2d array of floats that are not all integers, and both dimensions are of size > 1.
- 'binary': *y* contains <= 2 discrete values and is 1d or a column vector.
- 'multiclass': *y* contains more than two discrete values, is not a sequence of sequences, and is 1d or a column vector.
- 'multiclass-multioutput': *y* is a 2d array that contains more than two discrete values, is not a sequence of sequences, and both dimensions are of size > 1.
- 'multilabel-indicator': *y* is a label indicator matrix, an array of two dimensions with at least two columns, and at most 2 unique values.

- 'unknown': y is array-like but none of the above, such as a 3d array, sequence of sequences, or an array of non-sequence objects.

Examples

```
>>> import numpy as np
>>> type_of_target([0.1, 0.6])
'continuous'
>>> type_of_target([1, -1, -1, 1])
'binary'
>>> type_of_target(['a', 'b', 'a'])
'binary'
>>> type_of_target([1.0, 2.0])
'binary'
>>> type_of_target([1, 0, 2])
'multiclass'
>>> type_of_target([1.0, 0.0, 3.0])
'multiclass'
>>> type_of_target(['a', 'b', 'c'])
'multiclass'
>>> type_of_target(np.array([[1, 2], [3, 1]]))
'multiclass-multioutput'
>>> type_of_target([[1, 2]])
'multiclass-multioutput'
>>> type_of_target(np.array([[1.5, 2.0], [3.0, 1.6]]))
'continuous-multioutput'
>>> type_of_target(np.array([[0, 1], [1, 1]]))
'multilabel-indicator'
```

6.39.26 sklearn.utils.multiclass.is_multilabel

sklearn.utils.multiclass.**is_multilabel**(*y*)

Check if `y` is in a multilabel format.

Parameters

y [numpy array of shape [n_samples]] Target values.

Returns

out [bool,] Return `True`, if `y` is in a multilabel format, else ``False`.

Examples

```
>>> import numpy as np
>>> from sklearn.utils.multiclass import is_multilabel
>>> is_multilabel([0, 1, 0, 1])
False
>>> is_multilabel([[1], [0, 2], []])
False
>>> is_multilabel(np.array([[1, 0], [0, 0]]))
True
>>> is_multilabel(np.array([[1], [0], [0]]))
False
```

```
>>> is_multilabel(np.array([[1, 0, 0]]))
True
```

6.39.27 sklearn.utils.multiclass.unique_labels

sklearn.utils.multiclass.**unique_labels**(**ys*)

Extract an ordered array of unique labels

We don't allow:

- mix of multilabel and multiclass (single label) targets
- mix of label indicator matrix and anything else, because there are no explicit labels)
- mix of label indicator matrices of different sizes
- mix of string and integer labels

At the moment, we also don't allow "multiclass-multioutput" input type.

Parameters

***ys** [array-likes]

Returns

out [numpy array of shape [n_unique_labels]] An ordered array of unique labels.

Examples

```
>>> from sklearn.utils.multiclass import unique_labels
>>> unique_labels([3, 5, 5, 5, 7, 7])
array([3, 5, 7])
>>> unique_labels([1, 2, 3, 4], [2, 2, 3, 4])
array([1, 2, 3, 4])
>>> unique_labels([1, 2, 10], [5, 11])
array([ 1,  2,  5, 10, 11])
```

Examples using sklearn.utils.multiclass.unique_labels

- *Confusion matrix*

6.39.28 sklearn.utils.murmurhash3_32

sklearn.utils.**murmurhash3_32**()

Compute the 32bit murmurhash3 of key at seed.

The underlying implementation is MurmurHash3_x86_32 generating low latency 32bits hash suitable for implementing lookup tables, Bloom filters, count min sketch or feature hashing.

Parameters

key [int32, bytes, unicode or ndarray with dtype int32] the physical object to hash

seed [int, optional default is 0] integer seed for the hashing algorithm.

positive [boolean, optional default is False]

True: the results is casted to an unsigned int from 0 to 2 ** 32 - 1

False: the results is casted to a signed int from -(2 ** 31) to 2 ** 31 - 1

6.39.29 sklearn.utils.resample

sklearn.utils.**resample**(**arrays*, ***options*)

Resample arrays or sparse matrices in a consistent way

The default strategy implements one step of the bootstrapping procedure.

Parameters

***arrays** [sequence of indexable data-structures] Indexable data-structures can be arrays, lists, dataframes or scipy sparse matrices with consistent first dimension.

Returns

resampled_arrays [sequence of indexable data-structures] Sequence of resampled copies of the collections. The original arrays are not impacted.

Other Parameters

replace [boolean, True by default] Implements resampling with replacement. If False, this will implement (sliced) random permutations.

n_samples [int, None by default] Number of samples to generate. If left to None this is automatically set to the first dimension of the arrays. If replace is False it should not be larger than the length of arrays.

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. 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`.

stratify [array-like or None (default=None)] If not None, data is split in a stratified fashion, using this as the class labels.

See also:

sklearn.utils.shuffle

Examples

It is possible to mix sparse and dense arrays in the same run:

```
>>> X = np.array([[1., 0.], [2., 1.], [0., 0.]])
>>> y = np.array([0, 1, 2])

>>> from scipy.sparse import coo_matrix
>>> X_sparse = coo_matrix(X)

>>> from sklearn.utils import resample
>>> X, X_sparse, y = resample(X, X_sparse, y, random_state=0)
>>> X
array([[1., 0.],
       [2., 1.],
       [1., 0.]])
```

```
>>> X_sparse
<3x2 sparse matrix of type '<... 'numpy.float64'>'
    with 4 stored elements in Compressed Sparse Row format>

>>> X_sparse.toarray()
array([[1., 0.],
       [2., 1.],
       [1., 0.]])

>>> y
array([0, 1, 0])

>>> resample(y, n_samples=2, random_state=0)
array([0, 1])
```

Example using stratification:

```
>>> y = [0, 0, 1, 1, 1, 1, 1, 1, 1]
>>> resample(y, n_samples=5, replace=False, stratify=y,
...          random_state=0)
[1, 1, 1, 0, 1]
```

6.39.30 sklearn.utils.safe_indexing

sklearn.utils.**safe_indexing**(*X*, *indices*)

Return items or rows from X using indices.

Allows simple indexing of lists or arrays.

Parameters

X [array-like, sparse-matrix, list, pandas.DataFrame, pandas.Series.] Data from which to sample rows or items.

indices [array-like of int] Indices according to which X will be subsampled.

Returns

subset Subset of X on first axis

Notes

CSR, CSC, and LIL sparse matrices are supported. COO sparse matrices are not supported.

6.39.31 sklearn.utils.safe_mask

sklearn.utils.**safe_mask**(*X*, *mask*)

Return a mask which is safe to use on X.

Parameters

X [{array-like, sparse matrix}] Data on which to apply mask.

mask [array] Mask to be used on X.

Returns

mask

6.39.32 sklearn.utils.safe_sqr

sklearn.utils.**safe_sqr**(*X*, *copy=True*)

Element wise squaring of array-likes and sparse matrices.

Parameters

X [array like, matrix, sparse matrix]

copy [boolean, optional, default True] Whether to create a copy of X and operate on it or to perform inplace computation (default behaviour).

Returns

X ** 2 [element wise square]

6.39.33 sklearn.utils.shuffle

sklearn.utils.**shuffle**(**arrays*, ***options*)

Shuffle arrays or sparse matrices in a consistent way

This is a convenience alias to `resample(*arrays, replace=False)` to do random permutations of the collections.

Parameters

***arrays** [sequence of indexable data-structures] Indexable data-structures can be arrays, lists, dataframes or scipy sparse matrices with consistent first dimension.

Returns

shuffled_arrays [sequence of indexable data-structures] Sequence of shuffled copies of the collections. The original arrays are not impacted.

Other Parameters

random_state [int, RandomState instance or None, optional (default=None)] The seed of the pseudo random number generator to use when shuffling the data. 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`.

n_samples [int, None by default] Number of samples to generate. If left to None this is automatically set to the first dimension of the arrays.

See also:

sklearn.utils.resample

Examples

It is possible to mix sparse and dense arrays in the same run:

```
>>> X = np.array([[1., 0.], [2., 1.], [0., 0.]])
>>> y = np.array([0, 1, 2])

>>> from scipy.sparse import coo_matrix
>>> X_sparse = coo_matrix(X)

>>> from sklearn.utils import shuffle
>>> X, X_sparse, y = shuffle(X, X_sparse, y, random_state=0)
>>> X
array([[0., 0.],
       [2., 1.],
       [1., 0.]])

>>> X_sparse
<3x2 sparse matrix of type '<... 'numpy.float64'>'
    with 3 stored elements in Compressed Sparse Row format>

>>> X_sparse.toarray()
array([[0., 0.],
       [2., 1.],
       [1., 0.]])

>>> y
array([2, 1, 0])

>>> shuffle(y, n_samples=2, random_state=0)
array([0, 1])
```

Examples using `sklearn.utils.shuffle`

- *Model Complexity Influence*
- *Prediction Latency*
- *Color Quantization using K-Means*
- *Empirical evaluation of the impact of k-means initialization*
- *Gradient Boosting regression*
- *Early stopping of Stochastic Gradient Descent*

6.39.34 `sklearn.utils.sparsefuncs.incr_mean_variance_axis`

`sklearn.utils.sparsefuncs.`**`incr_mean_variance_axis`**(*X*, *axis*, *last_mean*, *last_var*, *last_n*)

Compute incremental mean and variance along an axix on a CSR or CSC matrix.

last_mean, last_var are the statistics computed at the last step by this function. Both must be initialized to 0-arrays of the proper size, i.e. the number of features in X. last_n is the number of samples encountered until now.

Parameters

X [CSR or CSC sparse matrix, shape (n_samples, n_features)] Input data.

axis [int (either 0 or 1)] Axis along which the axis should be computed.

last_mean [float array with shape (n_features,)] Array of feature-wise means to update with the new data X.

last_var [float array with shape (n_features,)] Array of feature-wise var to update with the new data X.

last_n [int with shape (n_features,)] Number of samples seen so far, excluded X.

Returns

means [float array with shape (n_features,)] Updated feature-wise means.

variances [float array with shape (n_features,)] Updated feature-wise variances.

n [int with shape (n_features,)] Updated number of seen samples.

Notes

NaNs are ignored in the algorithm.

6.39.35 sklearn.utils.sparsefuncs.inplace_column_scale

sklearn.utils.sparsefuncs.**inplace_column_scale**(*X*, *scale*)

Inplace column scaling of a CSC/CSR matrix.

Scale each feature of the data matrix by multiplying with specific scale provided by the caller assuming a (n_samples, n_features) shape.

Parameters

X [CSC or CSR matrix with shape (n_samples, n_features)] Matrix to normalize using the variance of the features.

scale [float array with shape (n_features,)] Array of precomputed feature-wise values to use for scaling.

6.39.36 sklearn.utils.sparsefuncs.inplace_row_scale

sklearn.utils.sparsefuncs.**inplace_row_scale**(*X*, *scale*)

Inplace row scaling of a CSR or CSC matrix.

Scale each row of the data matrix by multiplying with specific scale provided by the caller assuming a (n_samples, n_features) shape.

Parameters

X [CSR or CSC sparse matrix, shape (n_samples, n_features)] Matrix to be scaled.

scale [float array with shape (n_features,)] Array of precomputed sample-wise values to use for scaling.

6.39.37 sklearn.utils.sparsefuncs.inplace_swap_row

sklearn.utils.sparsefuncs.**inplace_swap_row**(*X*, *m*, *n*)

Swaps two rows of a CSC/CSR matrix in-place.

Parameters

X [CSR or CSC sparse matrix, shape=(n_samples, n_features)] Matrix whose two rows are to be swapped.

m [int] Index of the row of X to be swapped.

n [int] Index of the row of X to be swapped.

6.39.38 sklearn.utils.sparsefuncs.inplace_swap_column

sklearn.utils.sparsefuncs.**inplace_swap_column**(*X*, *m*, *n*)

Swaps two columns of a CSC/CSR matrix in-place.

Parameters

X [CSR or CSC sparse matrix, shape=(n_samples, n_features)] Matrix whose two columns are to be swapped.

m [int] Index of the column of X to be swapped.

n [int] Index of the column of X to be swapped.

6.39.39 sklearn.utils.sparsefuncs.mean_variance_axis

sklearn.utils.sparsefuncs.**mean_variance_axis**(*X*, *axis*)

Compute mean and variance along an axix on a CSR or CSC matrix

Parameters

X [CSR or CSC sparse matrix, shape (n_samples, n_features)] Input data.

axis [int (either 0 or 1)] Axis along which the axis should be computed.

Returns

means [float array with shape (n_features,)] Feature-wise means

variances [float array with shape (n_features,)] Feature-wise variances

6.39.40 sklearn.utils.sparsefuncs.inplace_csr_column_scale

sklearn.utils.sparsefuncs.**inplace_csr_column_scale**(*X*, *scale*)

Inplace column scaling of a CSR matrix.

Scale each feature of the data matrix by multiplying with specific scale provided by the caller assuming a (n_samples, n_features) shape.

Parameters

X [CSR matrix with shape (n_samples, n_features)] Matrix to normalize using the variance of the features.

scale [float array with shape (n_features,)] Array of precomputed feature-wise values to use for scaling.

6.39.41 sklearn.utils.sparsefuncs_fast.inplace_csr_row_normalize_l1

sklearn.utils.sparsefuncs_fast.**inplace_csr_row_normalize_l1**()

Inplace row normalize using the l1 norm

6.39.42 sklearn.utils.sparsefuncs_fast.inplace_csr_row_normalize_l2

sklearn.utils.sparsefuncs_fast.**inplace_csr_row_normalize_l2**()
Inplace row normalize using the l2 norm

6.39.43 sklearn.utils.random.sample_without_replacement

sklearn.utils.random.**sample_without_replacement**()
Sample integers without replacement.

Select n_samples integers from the set [0, n_population) without replacement.

Parameters

n_population [int,] The size of the set to sample from.

n_samples [int,] The number of integer to sample.

random_state [int, RandomState instance or None, optional (default=None)] 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`.

method ["auto", "tracking_selection", "reservoir_sampling" or "pool"] If method == "auto", the ratio of n_samples / n_population is used to determine which algorithm to use: If ratio is between 0 and 0.01, tracking selection is used. If ratio is between 0.01 and 0.99, numpy.random.permutation is used. If ratio is greater than 0.99, reservoir sampling is used. The order of the selected integers is undefined. If a random order is desired, the selected subset should be shuffled.

If method =="tracking_selection", a set based implementation is used which is suitable for *n_samples* <<< `n_population`.

If method == "reservoir_sampling", a reservoir sampling algorithm is used which is suitable for high memory constraint or when O(*n_samples*) ~ O(`n_population`). The order of the selected integers is undefined. If a random order is desired, the selected subset should be shuffled.

If method == "pool", a pool based algorithm is particularly fast, even faster than the tracking selection method. Hovewer, a vector containing the entire population has to be initialized. If n_samples ~ n_population, the reservoir sampling method is faster.

Returns

out [array of size (n_samples,)] The sampled subsets of integer. The subset of selected integer might not be randomized, see the method argument.

6.39.44 sklearn.utils.validation.check_is_fitted

sklearn.utils.validation.**check_is_fitted**(*estimator*, *attributes*, *msg=None*, *all_or_any=<built-in function all>*)
Perform is_fitted validation for estimator.

Checks if the estimator is fitted by verifying the presence of "all_or_any" of the passed attributes and raises a NotFittedError with the given message.

Parameters

estimator [estimator instance.] estimator instance for which the check is performed.

attributes [attribute name(s) given as string or a list/tuple of strings]

Eg.: `["coef_", "estimator_", ...], "coef_"`

msg [string] The default error message is, "This %(name)s instance is not fitted yet. Call 'fit' with appropriate arguments before using this method."

For custom messages if "%(name)s" is present in the message string, it is substituted for the estimator name.

Eg. : "Estimator, %(name)s, must be fitted before sparsifying".

all_or_any [callable, {all, any}, default all] Specify whether all or any of the given attributes must exist.

Returns

None

Raises

NotFittedError If the attributes are not found.

6.39.45 sklearn.utils.validation.check_memory

sklearn.utils.validation.**check_memory**(*memory*)

Check that `memory` is joblib.Memory-like.

joblib.Memory-like means that `memory` can be converted into a joblib.Memory instance (typically a str denoting the `location`) or has the same interface (has a `cache` method).

Parameters

memory [None, str or object with the joblib.Memory interface]

Returns

memory [object with the joblib.Memory interface]

Raises

ValueError If `memory` is not joblib.Memory-like.

6.39.46 sklearn.utils.validation.check_symmetric

sklearn.utils.validation.**check_symmetric**(*array*, *tol=1e-10*, *raise_warning=True*, *raise_exception=False*)

Make sure that array is 2D, square and symmetric.

If the array is not symmetric, then a symmetrized version is returned. Optionally, a warning or exception is raised if the matrix is not symmetric.

Parameters

array [nd-array or sparse matrix] Input object to check / convert. Must be two-dimensional and square, otherwise a ValueError will be raised.

tol [float] Absolute tolerance for equivalence of arrays. Default = 1E-10.

raise_warning [boolean (default=True)] If True then raise a warning if conversion is required.

raise_exception [boolean (default=False)] If True then raise an exception if array is not symmetric.

Returns

array_sym [ndarray or sparse matrix] Symmetrized version of the input array, i.e. the average of array and array.transpose(). If sparse, then duplicate entries are first summed and zeros are eliminated.

6.39.47 sklearn.utils.validation.column_or_1d

sklearn.utils.validation.**column_or_1d**(*y*, *warn=False*)

Ravel column or 1d numpy array, else raises an error

Parameters

y [array-like]

warn [boolean, default False] To control display of warnings.

Returns

y [array]

6.39.48 sklearn.utils.validation.has_fit_parameter

sklearn.utils.validation.**has_fit_parameter**(*estimator*, *parameter*)

Checks whether the estimator's fit method supports the given parameter.

Parameters

estimator [object] An estimator to inspect.

parameter [str] The searched parameter.

Returns

is_parameter: bool Whether the parameter was found to be a named parameter of the estimator's fit method.

Examples

```
>>> from sklearn.svm import SVC
>>> has_fit_parameter(SVC(), "sample_weight")
True
```

6.39.49 sklearn.utils.testing.assert_in

sklearn.utils.testing.**assert_in**(*member*, *container*, *msg=None*)

Just like self.assertTrue(a in b), but with a nicer default message.

6.39.50 sklearn.utils.testing.assert_not_in

sklearn.utils.testing.**assert_not_in**(*member*, *container*, *msg=None*)

Just like self.assertTrue(a not in b), but with a nicer default message.

6.39.51 sklearn.utils.testing.assert_raise_message

sklearn.utils.testing.**assert_raise_message**(*exceptions*, *message*, *function*, **args*, ***kwargs*)

Helper function to test the message raised in an exception.

Given an exception, a callable to raise the exception, and a message string, tests that the correct exception is raised and that the message is a substring of the error thrown. Used to test that the specific message thrown during an exception is correct.

Parameters

exceptions [exception or tuple of exception] An Exception object.

message [str] The error message or a substring of the error message.

function [callable] Callable object to raise error.

***args** [the positional arguments to *function*.]

****kwargs** [the keyword arguments to *function*.]

6.39.52 sklearn.utils.testing.all_estimators

sklearn.utils.testing.**all_estimators**(*include_meta_estimators=None*, *include_other=None*, *type_filter=None*, *include_dont_test=None*)

Get a list of all estimators from sklearn.

This function crawls the module and gets all classes that inherit from BaseEstimator. Classes that are defined in test-modules are not included. By default meta_estimators such as GridSearchCV are also not included.

Parameters

include_meta_estimators [boolean, default=False] Deprecated, ignored. .. deprecated:: 0.21

`include_meta_estimators` has been deprecated and has no effect in 0.21 and will be removed in 0.23.

include_other [boolean, default=False] Deprecated, ignored. .. deprecated:: 0.21

`include_other` has been deprecated and has not effect in 0.21 and will be removed in 0.23.

type_filter [string, list of string, or None, default=None] Which kind of estimators should be returned. If None, no filter is applied and all estimators are returned. Possible values are 'classifier', 'regressor', 'cluster' and 'transformer' to get estimators only of these specific types, or a list of these to get the estimators that fit at least one of the types.

include_dont_test [boolean, default=False] Deprecated, ignored. .. deprecated:: 0.21

`include_dont_test` has been deprecated and has no effect in 0.21 and will be removed in 0.23.

Returns

estimators [list of tuples] List of (name, class), where `name` is the class name as string and `class` is the actuall type of the class.

Utilities from joblib:

utils.parallel_backend(backend[, n_jobs])	Change the default backend used by Parallel inside a with block.
utils.register_parallel_backend(name, factory)	Register a new Parallel backend factory.

6.39.53 sklearn.utils.parallel_backend

sklearn.utils.**parallel_backend**(*backend*, *n_jobs=-1*, ***backend_params*)

Change the default backend used by Parallel inside a with block.

If backend is a string it must match a previously registered implementation using the register_parallel_backend function.

By default the following backends are available:

- 'loky': single-host, process-based parallelism (used by default),
- 'threading': single-host, thread-based parallelism,
- 'multiprocessing': legacy single-host, process-based parallelism.

'loky' is recommended to run functions that manipulate Python objects. 'threading' is a low-overhead alternative that is most efficient for functions that release the Global Interpreter Lock: e.g. I/O-bound code or CPU-bound code in a few calls to native code that explicitly releases the GIL.

In addition, if the dask and distributed Python packages are installed, it is possible to use the 'dask' backend for better scheduling of nested parallel calls without over-subscription and potentially distribute parallel calls over a networked cluster of several hosts.

Alternatively the backend can be passed directly as an instance.

By default all available workers will be used (n_jobs=-1) unless the caller passes an explicit value for the n_jobs parameter.

This is an alternative to passing a backend='backend_name' argument to the Parallel class constructor. It is particularly useful when calling into library code that uses joblib internally but does not expose the backend argument in its own API.

```
>>> from operator import neg
>>> with parallel_backend('threading'):
...     print(Parallel()(delayed(neg)(i + 1) for i in range(5)))
...
[-1, -2, -3, -4, -5]
```

Warning: this function is experimental and subject to change in a future version of joblib.

New in version 0.10.

6.39.54 sklearn.utils.register_parallel_backend

sklearn.utils.**register_parallel_backend**(*name*, *factory*, *make_default=False*)

Register a new Parallel backend factory.

The new backend can then be selected by passing its name as the backend argument to the Parallel class. Moreover, the default backend can be overwritten globally by setting make_default=True.

The factory can be any callable that takes no argument and return an instance of ParallelBackendBase.

Warning: this function is experimental and subject to change in a future version of joblib.

New in version 0.10.

6.40 Recently deprecated

6.40.1 To be removed in 0.23

utils.Memory(*args, **kwargs)	**Attributes**
utils.Parallel(*args, **kwargs)	**Methods**

sklearn.utils.Memory

Warning: DEPRECATED

class sklearn.utils.**Memory**(**args*, ***kwargs*)

Attributes

cachedir

Methods

cache(self[, func, ignore, verbose, mmap_mode])	Decorates the given function func to only compute its return value for input arguments not cached on disk.
clear(self[, warn])	Erase the complete cache directory.
eval(self, func, *args, **kwargs)	Eval function func with arguments `*args` and `**kwargs`, in the context of the memory.
format(self, obj[, indent])	Return the formatted representation of the object.
reduce_size(self)	Remove cache elements to make cache size fit in `bytes_limit`.

debug	
warn	

__init__(**args*, ***kwargs*)
DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23. Please import this functionality directly from joblib, which can be installed with: pip install joblib.

cache(*self*, *func=None*, *ignore=None*, *verbose=None*, *mmap_mode=False*)
Decorates the given function func to only compute its return value for input arguments not cached on disk.

Parameters

func: callable, optional The function to be decorated

ignore: list of strings A list of arguments name to ignore in the hashing

verbose: integer, optional The verbosity mode of the function. By default that of the memory object is used.

mmap_mode: {None, 'r+', 'r', 'w+', 'c'}, optional The memmapping mode used when loading from cache numpy arrays. See numpy.load for the meaning of the arguments. By default that of the memory object is used.

Returns

decorated_func: MemorizedFunc object The returned object is a MemorizedFunc object, that is callable (behaves like a function), but offers extra methods for cache lookup and management. See the documentation for `joblib.memory.MemorizedFunc`.

clear(*self*, *warn=True*)
Erase the complete cache directory.

eval(*self*, *func*, **args*, ***kwargs*)
Eval function func with arguments `*args` and `**kwargs`, in the context of the memory.

This method works similarly to the builtin *apply*, except that the function is called only if the cache is not up to date.

format(*self*, *obj*, *indent=0*)
Return the formatted representation of the object.

reduce_size(*self*)
Remove cache elements to make cache size fit in `bytes_limit`.

sklearn.utils.Parallel

Warning: DEPRECATED

class sklearn.utils.**Parallel**(**args*, ***kwargs*)

Methods

__call__(self, iterable)	
dispatch_next(self)	Dispatch more data for parallel processing
dispatch_one_batch(self, iterator)	Prefetch the tasks for the next batch and dispatch them.
format(self, obj[, indent])	Return the formatted representation of the object.
print_progress(self)	Display the process of the parallel execution only a fraction of time, controlled by self.verbose.

debug	
retrieve	
warn	

__init__(**args*, ***kwargs*)
DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23. Please import this function-

ality directly from joblib, which can be installed with: pip install joblib.

dispatch_next(*self*)
Dispatch more data for parallel processing

This method is meant to be called concurrently by the multiprocessing callback. We rely on the thread-safety of dispatch_one_batch to protect against concurrent consumption of the unprotected iterator.

dispatch_one_batch(*self*, *iterator*)
Prefetch the tasks for the next batch and dispatch them.

The effective size of the batch is computed here. If there are no more jobs to dispatch, return False, else return True.

The iterator consumption and dispatching is protected by the same lock so calling this function should be thread safe.

format(*self*, *obj*, *indent=0*)
Return the formatted representation of the object.

print_progress(*self*)
Display the process of the parallel execution only a fraction of time, controlled by self.verbose.

utils.cpu_count()	DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23.
utils.delayed(function[, check_pickle])	DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23.
metrics.calinski_harabaz_score(X, labels)	DEPRECATED: Function 'calinski_harabaz_score' has been renamed to 'calinski_harabasz_score' and will be removed in version 0.23.
metrics.jaccard_similarity_score(y_true, y_pred)	Jaccard similarity coefficient score
linear_model.logistic_regression_path(X, y)	DEPRECATED: logistic_regression_path was deprecated in version 0.21 and will be removed in version 0.23.0

sklearn.utils.cpu_count

Warning: DEPRECATED

sklearn.utils.**cpu_count**()
DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23. Please import this functionality directly from joblib, which can be installed with: pip install joblib.

Return the number of CPUs.

sklearn.utils.delayed

Warning: DEPRECATED

sklearn.utils.**delayed**(*function*, *check_pickle=None*)
DEPRECATED: deprecated in version 0.20.1 to be removed in version 0.23. Please import this functionality directly from joblib, which can be installed with: pip install joblib.

Decorator used to capture the arguments of a function.

sklearn.metrics.calinski_harabaz_score

Warning: DEPRECATED

sklearn.metrics.**calinski_harabaz_score**(*X*, *labels*)

DEPRECATED: Function 'calinski_harabaz_score' has been renamed to 'calinski_harabasz_score' and will be removed in version 0.23.

sklearn.metrics.jaccard_similarity_score

Warning: DEPRECATED

sklearn.metrics.**jaccard_similarity_score**(*y_true*, *y_pred*, *normalize=True*, *sample_weight=None*)

Jaccard similarity coefficient score

Deprecated since version 0.21: This is deprecated to be removed in 0.23, since its handling of binary and multiclass inputs was broken. `jaccard_score` has an API that is consistent with precision_score, f_score, etc.

Read more in the *User Guide*.

Parameters

y_true [1d array-like, or label indicator array / sparse matrix] Ground truth (correct) labels.

y_pred [1d array-like, or label indicator array / sparse matrix] Predicted labels, as returned by a classifier.

normalize [bool, optional (default=True)] If `False`, return the sum of the Jaccard similarity coefficient over the sample set. Otherwise, return the average of Jaccard similarity coefficient.

sample_weight [array-like of shape = [n_samples], optional] Sample weights.

Returns

score [float] If `normalize == True`, return the average Jaccard similarity coefficient, else it returns the sum of the Jaccard similarity coefficient over the sample set.

The best performance is 1 with `normalize == True` and the number of samples with `normalize == False`.

See also:

accuracy_score*, *hamming_loss*, *zero_one_loss

Notes

In binary and multiclass classification, this function is equivalent to the `accuracy_score`. It differs in the multilabel classification problem.

References

[1]

sklearn.linear_model.logistic_regression_path

Warning: DEPRECATED

sklearn.linear_model.**logistic_regression_path**(*X*, *y*, *pos_class=None*, *Cs=10*, *fit_intercept=True*, *max_iter=100*, *tol=0.0001*, *verbose=0*, *solver='lbfgs'*, *coef=None*, *class_weight=None*, *dual=False*, *penalty='l2'*, *intercept_scaling=1.0*, *multi_class='warn'*, *random_state=None*, *check_input=True*, *max_squared_sum=None*, *sample_weight=None*, *l1_ratio=None*)

DEPRECATED: logistic_regression_path was deprecated in version 0.21 and will be removed in version 0.23.0

Compute a Logistic Regression model for a list of regularization parameters.

This is an implementation that uses the result of the previous model to speed up computations along the set of solutions, making it faster than sequentially calling LogisticRegression for the different parameters. Note that there will be no speedup with liblinear solver, since it does not handle warm-starting.

Deprecated since version 0.21: `logistic_regression_path` was deprecated in version 0.21 and will be removed in 0.23.

Read more in the *User Guide*.

Parameters

X [array-like or sparse matrix, shape (n_samples, n_features)]

Input data.

y [array-like, shape (n_samples,) or (n_samples, n_targets)] Input data, target values.

pos_class [int, None] The class with respect to which we perform a one-vs-all fit. If None, then it is assumed that the given problem is binary.

Cs [int | array-like, shape (n_cs,)] List of values for the regularization parameter or integer specifying the number of regularization parameters that should be used. In this case, the parameters will be chosen in a logarithmic scale between 1e-4 and 1e4.

fit_intercept [bool] Whether to fit an intercept for the model. In this case the shape of the returned array is (n_cs, n_features + 1).

max_iter [int] Maximum number of iterations for the solver.

tol [float] Stopping criterion. For the newton-cg and lbfgs solvers, the iteration will stop when `max{|g_i | i = 1, ..., n} <= tol` where `g_i` is the i-th component of the gradient.

verbose [int] For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.

solver [{'lbfgs', 'newton-cg', 'liblinear', 'sag', 'saga'}] Numerical solver to use.

coef [array-like, shape (n_features,), default None] Initialization value for coefficients of logistic regression. Useless for liblinear solver.

class_weight [dict or 'balanced', optional] Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

dual [bool] Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.

penalty [str, 'l1', 'l2', or 'elasticnet'] Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is only supported by the 'saga' solver.

intercept_scaling [float, default 1.] Useful only when the solver 'liblinear' is used and self.fit_intercept is set to True. In this case, x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equal to intercept_scaling is appended to the instance vector. The intercept becomes `intercept_scaling * synthetic_feature_weight`.

Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

multi_class [str, {'ovr', 'multinomial', 'auto'}, default: 'ovr'] If the option chosen is 'ovr', then a binary problem is fit for each label. For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. 'multinomial' is unavailable when solver='liblinear'. 'auto' selects 'ovr' if the data is binary, or if solver='liblinear', and otherwise selects 'multinomial'.

New in version 0.18: Stochastic Average Gradient descent solver for 'multinomial' case.

Changed in version 0.20: Default will change from 'ovr' to 'auto' in 0.22.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag' or 'liblinear'.

check_input [bool, default True] If False, the input arrays X and y will not be checked.

max_squared_sum [float, default None] Maximum squared sum of X over samples. Used only in SAG solver. If None, it will be computed, going through all the samples. The value should be precomputed to speed up cross validation.

sample_weight [array-like, shape(n_samples,) optional] Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight.

l1_ratio [float or None, optional (default=None)] The Elastic-Net mixing parameter, with `0 <= l1_ratio <= 1`. Only used if `penalty='elasticnet'`. Setting `l1_ratio=0` is equivalent to using `penalty='l2'`, while setting `l1_ratio=1` is equivalent to using `penalty='l1'`. For `0 < l1_ratio <1`, the penalty is a combination of L1 and L2.

Returns

coefs [ndarray, shape (n_cs, n_features) or (n_cs, n_features + 1)]

List of coefficients for the Logistic Regression model. If fit_intercept is set to True then the second dimension will be n_features + 1, where the last item represents the intercept. For `multiclass='multinomial'`, the shape is (n_classes, n_cs, n_features) or (n_classes, n_cs, n_features + 1).

Cs [ndarray] Grid of Cs used for cross-validation.

n_iter [array, shape (n_cs,)] Actual number of iteration for each Cs.

Notes

You might get slightly different results with the solver liblinear than with the others since this uses LIBLINEAR which penalizes the intercept.

Changed in version 0.19: The "copy" parameter was removed.

ensemble.partial_dependence.partial_dependence(...)	DEPRECATED: The function ensemble.partial_dependence has been deprecated in favour of inspection.partial_dependence in 0.21 and will be removed in 0.23.
ensemble.partial_dependence.plot_partial_dependence(...)	DEPRECATED: The function ensemble.plot_partial_dependence has been deprecated in favour of sklearn.inspection.plot_partial_dependence in 0.21 and will be removed in 0.23.

sklearn.ensemble.partial_dependence.partial_dependence

sklearn.ensemble.partial_dependence.**partial_dependence**(*gbrt*, *target_variables*, *grid=None*, *X=None*, *percentiles=(0.05*, *0.95)*, *grid_resolution=100*)

DEPRECATED: The function ensemble.partial_dependence has been deprecated in favour of inspection.partial_dependence in 0.21 and will be removed in 0.23.

Partial dependence of `target_variables`.

Partial dependence plots show the dependence between the joint values of the `target_variables` and the function represented by the `gbrt`.

Read more in the *User Guide*.

Deprecated since version 0.21: This function was deprecated in version 0.21 in favor of *sklearn.inspection.partial_dependence* and will be removed in 0.23.

Parameters

gbrt [BaseGradientBoosting]

A fitted gradient boosting model.

target_variables [array-like, dtype=int] The target features for which the partial dependency should be computed (size should be smaller than 3 for visual renderings).

grid [array-like, shape=(n_points, len(target_variables))] The grid of `target_variables` values for which the partial dependency should be evaluated (either `grid` or `X` must be specified).

X [array-like, shape=(n_samples, n_features)] The data on which `gbrt` was trained. It is used to generate a `grid` for the `target_variables`. The `grid` comprises `grid_resolution` equally spaced points between the two `percentiles`.

percentiles [(low, high), default=(0.05, 0.95)] The lower and upper percentile used create the extreme values for the `grid`. Only if `X` is not None.

grid_resolution [int, default=100] The number of equally spaced points on the `grid`.

Returns

pdp [array, shape=(n_classes, n_points)]

The partial dependence function evaluated on the `grid`. For regression and binary classification `n_classes==1`.

axes [seq of ndarray or None] The axes with which the grid has been created or None if the grid has been given.

Examples

```
>>> samples = [[0, 0, 2], [1, 0, 0]]
>>> labels = [0, 1]
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> gb = GradientBoostingClassifier(random_state=0).fit(samples, labels)
>>> kwargs = dict(X=samples, percentiles=(0, 1), grid_resolution=2)
>>> partial_dependence(gb, [0], **kwargs)
(array([[-4.52...,  4.52...]]), [array([ 0.,  1.])])
```

sklearn.ensemble.partial_dependence.plot_partial_dependence

sklearn.ensemble.partial_dependence.**plot_partial_dependence**(*gbrt*, *X*, *features*, *feature_names=None*, *label=None*, *n_cols=3*, *grid_resolution=100*, *percentiles=(0.05, 0.95)*, *n_jobs=None*, *verbose=0*, *ax=None*, *line_kw=None*, *contour_kw=None*, ***fig_kw*)

DEPRECATED: The function ensemble.plot_partial_dependence has been deprecated in favour of sklearn.inspection.plot_partial_dependence in 0.21 and will be removed in 0.23.

Partial dependence plots for `features`.

The `len(features)` plots are arranged in a grid with `n_cols` columns. Two-way partial dependence plots are plotted as contour plots.

Read more in the *User Guide*.

Deprecated since version 0.21: This function was deprecated in version 0.21 in favor of *sklearn.inspection.plot_partial_dependence* and will be removed in 0.23.

Parameters

gbrt [BaseGradientBoosting]

A fitted gradient boosting model.

X [array-like, shape=(n_samples, n_features)] The data on which `gbrt` was trained.

features [seq of ints, strings, or tuples of ints or strings] If seq[i] is an int or a tuple with one int value, a one-way PDP is created; if seq[i] is a tuple of two ints, a two-way PDP is created. If feature_names is specified and seq[i] is an int, seq[i] must be < len(feature_names). If seq[i] is a string, feature_names must be specified, and seq[i] must be in feature_names.

feature_names [seq of str] Name of each feature; feature_names[i] holds the name of the feature with index i.

label [object] The class label for which the PDPs should be computed. Only if gbrt is a multi-class model. Must be in `gbrt.classes_`.

n_cols [int] The number of columns in the grid plot (default: 3).

grid_resolution [int, default=100] The number of equally spaced points on the axes.

percentiles [(low, high), default=(0.05, 0.95)] The lower and upper percentile used to create the extreme values for the PDP axes.

n_jobs [int or None, optional (default=None)] `None` means 1 unless in a *joblib.parallel_backend* context. `-1` means using all processors. See *Glossary* for more details.

verbose [int] Verbose output during PD computations. Defaults to 0.

ax [Matplotlib axis object, default None] An axis object onto which the plots will be drawn.

line_kw [dict] Dict with keywords passed to the `matplotlib.pyplot.plot` call. For one-way partial dependence plots.

contour_kw [dict] Dict with keywords passed to the `matplotlib.pyplot.plot` call. For two-way partial dependence plots.

****fig_kw** [dict] Dict with keywords passed to the figure() call. Note that all keywords not recognized above will be automatically included here.

Returns

fig [figure]

The Matplotlib Figure object.

axs [seq of Axis objects] A seq of Axis objects, one for each subplot.

Examples

```
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> X, y = make_friedman1()
>>> clf = GradientBoostingRegressor(n_estimators=10).fit(X, y)
```

```
>>> fig, axs = plot_partial_dependence(clf, X, [0, (0, 1)])
...
```

6.40.2 To be removed in 0.22

covariance.GraphLasso(*args, **kwargs)	Sparse inverse covariance estimation with an l1-penalized estimator.
covariance.GraphLassoCV(*args, **kwargs)	Sparse inverse covariance w/ cross-validated choice of the l1 penalty.
preprocessing.Imputer(*args, **kwargs)	Imputation transformer for completing missing values.
utils.testing.mock_mldata_urlopen(*args, ...)	Object that mocks the urlopen function to fake requests to mldata.

sklearn.covariance.GraphLasso

Warning: DEPRECATED

class sklearn.covariance.**GraphLasso**(**args*, ***kwargs*)

Sparse inverse covariance estimation with an l1-penalized estimator.

This class implements the Graphical Lasso algorithm.

Read more in the *User Guide*.

Parameters

alpha [positive float, default 0.01] The regularization parameter: the higher alpha, the more regularization, the sparser the inverse covariance.

mode [{'cd', 'lars'}, default 'cd'] The Lasso solver to use: coordinate descent or LARS. Use LARS for very sparse underlying graphs, where p > n. Elsewhere prefer cd which is more numerically stable.

tol [positive float, default 1e-4] The tolerance to declare convergence: if the dual gap goes below this value, iterations are stopped.

enet_tol [positive float, optional] The tolerance for the elastic net solver used to calculate the descent direction. This parameter controls the accuracy of the search direction for a given column update, not of the overall parameter estimate. Only used for mode='cd'.

max_iter [integer, default 100] The maximum number of iterations.

verbose [boolean, default False] If verbose is True, the objective function and dual gap are plotted at each iteration.

assume_centered [boolean, default False] If True, data are not centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False, data are centered before computation.

Attributes

covariance_ [array-like, shape (n_features, n_features)] Estimated covariance matrix

precision_ [array-like, shape (n_features, n_features)] Estimated pseudo inverse matrix.

n_iter_ [int] Number of iterations run.

See also:

graph_lasso, *GraphLassoCV*

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the GraphicalLasso model to X.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(**args*, ***kwargs*)
DEPRECATED: The 'GraphLasso' was renamed to 'GraphicalLasso' in version 0.20 and will be removed in 0.22.

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)
Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like, shape = [n_features, n_features]] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True (default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

fit(*self*, *X*, *y=None*)
Fits the GraphicalLasso model to X.

Parameters

X [ndarray, shape (n_samples, n_features)] Data from which to compute the covariance estimate

y [(ignored)]

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

get_precision(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

mahalanobis(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like, shape = [n_samples, n_features]] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

score(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like, shape = [n_samples, n_features]] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***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.

Returns

self

sklearn.covariance.GraphLassoCV

Warning: DEPRECATED

class sklearn.covariance.**GraphLassoCV**(**args*, ***kwargs*)

Sparse inverse covariance w/ cross-validated choice of the l1 penalty.

See glossary entry for *cross-validation estimator*.

This class implements the Graphical Lasso algorithm.

Read more in the *User Guide*.

Parameters

alphas [integer, or list positive float, optional] If an integer is given, it fixes the number of points on the grids of alpha to be used. If a list is given, it gives the grid to be used. See the notes in the class docstring for more details.

n_refinements [strictly positive integer] The number of times the grid is refined. Not used if explicit values of alphas are passed.

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs KFold is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: cv default value if None will change from 3-fold to 5-fold in v0.22.

tol [positive float, optional] The tolerance to declare convergence: if the dual gap goes below this value, iterations are stopped.

enet_tol [positive float, optional] The tolerance for the elastic net solver used to calculate the descent direction. This parameter controls the accuracy of the search direction for a given column update, not of the overall parameter estimate. Only used for mode='cd'.

max_iter [integer, optional] Maximum number of iterations.

mode [{'cd', 'lars'}] The Lasso solver to use: coordinate descent or LARS. Use LARS for very sparse underlying graphs, where number of features is greater than number of samples. Elsewhere prefer cd which is more numerically stable.

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

verbose [boolean, optional] If verbose is True, the objective function and duality gap are printed at each iteration.

assume_centered [Boolean] If True, data are not centered before computation. Useful when working with data whose mean is almost, but not exactly zero. If False, data are centered before computation.

Attributes

covariance_ [numpy.ndarray, shape (n_features, n_features)] Estimated covariance matrix.

precision_ [numpy.ndarray, shape (n_features, n_features)] Estimated precision matrix (inverse covariance).

alpha_ [float] Penalization parameter selected.

cv_alphas_ [list of float] All penalization parameters explored.

grid_scores_ [2D numpy.ndarray (n_alphas, n_folds)] Log-likelihood score on left-out data across folds.

n_iter_ [int] Number of iterations run for the optimal alpha.

See also:

graph_lasso, *GraphLasso*

Notes

The search for the optimal penalization parameter (alpha) is done on an iteratively refined grid: first the cross-validated scores on a grid are computed, then a new refined grid is centered around the maximum, and so on.

One of the challenges which is faced here is that the solvers can fail to converge to a well-conditioned estimate. The corresponding values of alpha then come out as missing values, but the optimum may be close to these missing values.

Methods

error_norm(self, comp_cov[, norm, scaling, ...])	Computes the Mean Squared Error between two covariance estimators.
fit(self, X[, y])	Fits the GraphicalLasso covariance model to X.
get_params(self[, deep])	Get parameters for this estimator.
get_precision(self)	Getter for the precision matrix.
mahalanobis(self, X)	Computes the squared Mahalanobis distances of given observations.
score(self, X_test[, y])	Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.
set_params(self, **params)	Set the parameters of this estimator.

__init__(**args*, ***kwargs*)
DEPRECATED: The 'GraphLassoCV' was renamed to 'GraphicalLassoCV' in version 0.20 and will be removed in 0.22.

error_norm(*self*, *comp_cov*, *norm='frobenius'*, *scaling=True*, *squared=True*)
Computes the Mean Squared Error between two covariance estimators. (In the sense of the Frobenius norm).

Parameters

comp_cov [array-like, shape = [n_features, n_features]] The covariance to compare with.

norm [str] The type of norm used to compute the error. Available error types: - 'frobenius' (default): sqrt(tr(A^t.A)) - 'spectral': sqrt(max(eigenvalues(A^t.A)) where A is the error `(comp_cov - self.covariance_)`.

scaling [bool] If True (default), the squared error norm is divided by n_features. If False, the squared error norm is not rescaled.

squared [bool] Whether to compute the squared error norm or the error norm. If True

(default), the squared error norm is returned. If False, the error norm is returned.

Returns

The Mean Squared Error (in the sense of the Frobenius norm) between

`self` and `comp_cov` covariance estimators.

`fit`(*self*, *X*, *y=None*)
Fits the GraphicalLasso covariance model to X.

Parameters

X [ndarray, shape (n_samples, n_features)] Data from which to compute the covariance estimate

y [(ignored)]

`get_params`(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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.

`get_precision`(*self*)
Getter for the precision matrix.

Returns

precision_ [array-like] The precision matrix associated to the current covariance object.

`mahalanobis`(*self*, *X*)
Computes the squared Mahalanobis distances of given observations.

Parameters

X [array-like, shape = [n_samples, n_features]] The observations, the Mahalanobis distances of the which we compute. Observations are assumed to be drawn from the same distribution than the data used in fit.

Returns

dist [array, shape = [n_samples,]] Squared Mahalanobis distances of the observations.

`score`(*self*, *X_test*, *y=None*)
Computes the log-likelihood of a Gaussian data set with `self.covariance_` as an estimator of its covariance matrix.

Parameters

X_test [array-like, shape = [n_samples, n_features]] Test data of which we compute the likelihood, where n_samples is the number of samples and n_features is the number of features. X_test is assumed to be drawn from the same distribution than the data used in fit (including centering).

y not used, present for API consistence purpose.

Returns

res [float] The likelihood of the data set with `self.covariance_` as an estimator of its covariance matrix.

set_params(*self*, ***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.

Returns

self

sklearn.preprocessing.Imputer

Warning: DEPRECATED

class sklearn.preprocessing.**Imputer**(**args*, ***kwargs*)
Imputation transformer for completing missing values.

Read more in the *User Guide*.

Parameters

missing_values [integer or "NaN", optional (default="NaN")] The placeholder for the missing values. All occurrences of `missing_values` will be imputed. For missing values encoded as np.nan, use the string value "NaN".

strategy [string, optional (default="mean")] The imputation strategy.

- If "mean", then replace missing values using the mean along the axis.
- If "median", then replace missing values using the median along the axis.
- If "most_frequent", then replace missing using the most frequent value along the axis.

axis [integer, optional (default=0)] The axis along which to impute.

- If `axis=0`, then impute along columns.
- If `axis=1`, then impute along rows.

verbose [integer, optional (default=0)] Controls the verbosity of the imputer.

copy [boolean, optional (default=True)] If True, a copy of X will be created. If False, imputation will be done in-place whenever possible. Note that, in the following cases, a new copy will always be made, even if `copy=False`:

- If X is not an array of floating values;
- If X is sparse and `missing_values=0`;
- If `axis=0` and X is encoded as a CSR matrix;
- If `axis=1` and X is encoded as a CSC matrix.

Attributes

statistics_ [array of shape (n_features,)] The imputation fill value for each feature if axis == 0.

Notes

- When `axis=0`, columns which only contained missing values at *`fit`* are discarded upon *`transform`*.
- When `axis=1`, an exception is raised if there are rows for which it is not possible to fill in the missing values (e.g., because they only contain missing values).

Methods

`fit`(self, X[, y])	Fit the imputer on X.
`fit_transform`(self, X[, y])	Fit to data, then transform it.
`get_params`(self[, deep])	Get parameters for this estimator.
`set_params`(self, **params)	Set the parameters of this estimator.
`transform`(self, X)	Impute all missing values in X.

__init__(**args*, ***kwargs*)
DEPRECATED: Imputer was deprecated in version 0.20 and will be removed in 0.22. Import impute.SimpleImputer from sklearn instead.

fit(*self*, *X*, *y=None*)
Fit the imputer on X.

Parameters

X [{array-like, sparse matrix}, shape (n_samples, n_features)] Input data, where `n_samples` is the number of samples and `n_features` is the number of features.

Returns

self [Imputer]

fit_transform(*self*, *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 [numpy array of shape [n_samples, n_features]] Training set.

y [numpy array of shape [n_samples]] Target values.

Returns

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] 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(*self*, ***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.

Returns

self

transform(*self*, *X*)
Impute all missing values in X.

Parameters

X [{array-like, sparse matrix}, shape = [n_samples, n_features]] The input data to complete.

sklearn.utils.testing.mock_mldata_urlopen

Warning: DEPRECATED

class sklearn.utils.testing.**mock_mldata_urlopen**(**args*, ***kwargs*)
Object that mocks the urlopen function to fake requests to mldata.

When requesting a dataset with a name that is in mock_datasets, this object creates a fake dataset in a StringIO object and returns it. Otherwise, it raises an HTTPError.

Deprecated since version 0.20: Will be removed in version 0.22

Parameters

mock_datasets [dict] A dictionary of {dataset_name: data_dict}, or {dataset_name: (data_dict, ordering). `data_dict` itself is a dictionary of {column_name: data_array}, and `ordering` is a list of column_names to determine the ordering in the data set (see `fake_mldata` for details).

Methods

`__call__`(self, urlname)	**Parameters**

__init__(**args*, ***kwargs*)
DEPRECATED: deprecated in version 0.20 to be removed in version 0.22

covariance.graph_lasso(emp_cov, alpha[, ...])	DEPRECATED: The 'graph_lasso' was renamed to 'graphical_lasso' in version 0.20 and will be removed in 0.22.
datasets.fetch_mldata(dataname[, ...])	DEPRECATED: fetch_mldata was deprecated in version 0.20 and will be removed in version 0.22.
datasets.mldata_filename(dataname)	DEPRECATED: mldata_filename was deprecated in version 0.20 and will be removed in version 0.22.

sklearn.covariance.graph_lasso

Warning: DEPRECATED

sklearn.covariance.**graph_lasso**(*emp_cov*, *alpha*, *cov_init=None*, *mode='cd'*, *tol=0.0001*, *enet_tol=0.0001*, *max_iter=100*, *verbose=False*, *return_costs=False*, *eps=2.220446049250313e-16*, *return_n_iter=False*)

DEPRECATED: The 'graph_lasso' was renamed to 'graphical_lasso' in version 0.20 and will be removed in 0.22.

l1-penalized covariance estimator

Read more in the *User Guide*.

Parameters

emp_cov [2D ndarray, shape (n_features, n_features)]

Empirical covariance from which to compute the covariance estimate.

alpha [positive float] The regularization parameter: the higher alpha, the more regularization, the sparser the inverse covariance.

cov_init [2D array (n_features, n_features), optional] The initial guess for the covariance.

mode [{'cd', 'lars'}] The Lasso solver to use: coordinate descent or LARS. Use LARS for very sparse underlying graphs, where p > n. Elsewhere prefer cd which is more numerically stable.

tol [positive float, optional] The tolerance to declare convergence: if the dual gap goes below this value, iterations are stopped.

enet_tol [positive float, optional] The tolerance for the elastic net solver used to calculate the descent direction. This parameter controls the accuracy of the search direction for a given column update, not of the overall parameter estimate. Only used for mode='cd'.

max_iter [integer, optional] The maximum number of iterations.

verbose [boolean, optional] If verbose is True, the objective function and dual gap are printed at each iteration.

return_costs [boolean, optional] If return_costs is True, the objective function and dual gap at each iteration are returned.

eps [float, optional] The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.

return_n_iter [bool, optional] Whether or not to return the number of iterations.

Returns

covariance [2D ndarray, shape (n_features, n_features)]

The estimated covariance matrix.

precision [2D ndarray, shape (n_features, n_features)] The estimated (sparse) precision matrix.

costs [list of (objective, dual_gap) pairs] The list of values of the objective function and the dual gap at each iteration. Returned only if return_costs is True.

n_iter [int] Number of iterations. Returned only if `return_n_iter` is set to True.

Notes

The algorithm employed to solve this problem is the GLasso algorithm, from the Friedman 2008 Biostatistics paper. It is the same algorithm as in the R `glasso` package.

One possible difference with the `glasso` R package is that the diagonal coefficients are not penalized.

sklearn.datasets.fetch_mldata

Warning: DEPRECATED

sklearn.datasets.**fetch_mldata**(*dataname*, *target_name='label'*, *data_name='data'*, *transpose_data=True*, *data_home=None*)

DEPRECATED: fetch_mldata was deprecated in version 0.20 and will be removed in version 0.22. Please use fetch_openml.

Fetch an mldata.org data set

mldata.org is no longer operational.

If the file does not exist yet, it is downloaded from mldata.org .

mldata.org does not have an enforced convention for storing data or naming the columns in a data set. The default behavior of this function works well with the most common cases:

1. data values are stored in the column 'data', and target values in the column 'label'
2. alternatively, the first column stores target values, and the second data values
3. the data array is stored as `n_features x n_samples` , and thus needs to be transposed to match the `sklearn` standard

Keyword arguments allow to adapt these defaults to specific data sets (see parameters `target_name`, `data_name`, `transpose_data`, and the examples below).

mldata.org data sets may have multiple columns, which are stored in the Bunch object with their original name.

Deprecated since version 0.20: Will be removed in version 0.22

Parameters

dataname [str]

Name of the data set on mldata.org, e.g.: "leukemia", "Whistler Daily Snowfall", etc. The raw name is automatically converted to a mldata.org URL .

target_name [optional, default: 'label'] Name or index of the column containing the target values.

data_name [optional, default: 'data'] Name or index of the column containing the data.

transpose_data [optional, default: True] If True, transpose the downloaded data array.

data_home [optional, default: None] Specify another download and cache folder for the data sets. By default all scikit-learn data is stored in '~/scikit_learn_data' subfolders.

Returns

data [Bunch] Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'DESCR', the full description of the dataset, and 'COL_NAMES', the original names of the dataset columns.

sklearn.datasets.mldata_filename

Warning: DEPRECATED

sklearn.datasets.**mldata_filename**(*dataname*)

DEPRECATED: mldata_filename was deprecated in version 0.20 and will be removed in version 0.22. Please use fetch_openml.

Convert a raw name for a data set in a mldata.org filename.

Deprecated since version 0.20: Will be removed in version 0.22

Parameters

dataname [str] Name of dataset

Returns

fname [str] The converted dataname.