OneClassSVM#

***class* sklearn.svm.OneClassSVM(**, *kernel*='rbf', *degree*=3, *gamma*='scale', *coef0*=0.0, *tol*=0.001, *nu*=0.5, *shrinking*=True, *cache_size*=200, *verbose*=False, *max_iter*=-1)[source]**

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*{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'} or callable, default='rbf'*

Specifies the kernel type to be used in the algorithm. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

degree*int, default=3*

Degree of the polynomial kernel function ('poly'). Must be non-negative. Ignored by all other kernels.

gamma*{'scale', 'auto'} or float, default='scale'*

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if gamma='scale' (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features
- if float, must be non-negative.

Changed in version 0.22: The default value of gamma changed from 'auto' to 'scale'.

coef0*float, default=0.0*

Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

tol*float, default=1e-3*

Tolerance for stopping criterion.

nu*float, 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]. By default 0.5 will be taken.

shrinking*bool, default=True*

Whether to use the shrinking heuristic. See the User Guide.

cache_size*float, default=200*

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, default=-1*

Hard limit on iterations within solver, or -1 for no limit.

Attributes:

coef_*ndarray of shape (1, n_features)*

Weights assigned to the features when kernel="linear".

dual_coef_*ndarray of shape (1, n_SV)*

Coefficients of the support vectors in the decision function.

fit_status_*int*

0 if correctly fitted, 1 otherwise (will raise warning)

intercept_*ndarray of shape (1,)*

Constant in the decision function.

n_features_in_*int*

Number of features seen during fit.

Added in version 0.24.

feature_names_in_*ndarray of shape (n_features_in_,)*

Names of features seen during fit. Defined only when X has feature names that are all strings.

Added in version 1.0.

n_iter_*int*

Number of iterations run by the optimization routine to fit the model.

Added in version 1.1.

n_support_*ndarray of shape (n_classes,), dtype=int32*

Number of support vectors for each class.

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.

Added in version 0.20.

shape_fit_*tuple of int of shape (n_dimensions_of_X,)*

Array dimensions of training vector X.

support_*ndarray of shape (n_SV,)*

Indices of support vectors.

support_vectors_*ndarray of shape (n_SV, n_features)*

Support vectors.

See also

sklearn.linear_model.SGDOneClassSVM

Solves linear One-Class SVM using Stochastic Gradient Descent.

sklearn.neighbors.LocalOutlierFactor

Unsupervised Outlier Detection using Local Outlier Factor (LOF).

sklearn.ensemble.IsolationForest

Isolation Forest Algorithm.

Examples

```
>>> from sklearn.svm import OneClassSVM
>>> X = [[0], [0.44], [0.45], [0.46], ]
>>> 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...])
```

***property* coef_**

Weights assigned to the features when kernel="linear".

Returns:

ndarray of shape (n_features, n_classes)

decision_function(*X*)[source]

Signed distance to the separating hyperplane.

Signed distance is positive for an inlier and negative for an outlier.

Parameters:

X*array-like of shape (n_samples, n_features)*

The data matrix.

Returns:

dec*ndarray of shape (n_samples,)*

Returns the decision function of the samples.

fit(*X*, *y*=*None*, *sample_weight*=*None*)[source]

Detect the soft boundary of the set of samples X.

Parameters:

X*{array-like, sparse matrix} of shape (n_samples, n_features)*

Set of samples, where n_samples is the number of samples and n_features is the number of features.

y*Ignored*

Not used, present for API consistency by convention.

sample_weight*array-like of shape (n_samples,), default=None*

Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns:

self*object*

Fitted estimator.

Notes

If X is not a C-ordered contiguous array it is copied.

fit_predict(*X*, *y*=*None*, ***kwargs*)[source]

Perform fit on X and returns labels for X.

Returns -1 for outliers and 1 for inliers.

Parameters:

X*{array-like, sparse matrix} of shape (n_samples, n_features)*

The input samples.

y*Ignored*

Not used, present for API consistency by convention.

****kwargs***dict*

Arguments to be passed to fit.

Added in version 1.4.

Returns:

y*ndarray of shape (n_samples,)*

1 for inliers, -1 for outliers.

get_metadata_routing()[source]

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns:

routing*MetadataRequest*

A **MetadataRequest** encapsulating routing information.

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

Get parameters for this estimator.

Parameters:

deep*bool, default=True*

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns:

params*dict*

Parameter names mapped to their values.

property **n_support_**

Number of support vectors for each class.

predict(*X*)[source]

Perform classification on samples in X.

For a one-class model, +1 or -1 is returned.

Parameters:

X*(array-like, sparse matrix) of shape (n_samples, n_features) or (n_samples_test, n_samples_train)*

For kernel="precomputed", the expected shape of X is (n_samples_test, n_samples_train).

Returns:

y_pred*ndarray of shape (n_samples,)*

Class labels for samples in X.

score_samples(*X*)[source]

Raw scoring function of the samples.

Parameters:

X*array-like of shape (n_samples, n_features)*

The data matrix.

Returns:

score_samples*ndarray of shape (n_samples,)*

Returns the (unshifted) scoring function of the samples.

set_fit_request(**, sample_weight: **bool** / **None** / **str** = '$UNCHANGED$'***) → **OneClassSVM[source]***

Request metadata passed to the fit method.

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

The options for each parameter are:

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

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

Added in version 1.3.

Note

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

Parameters:

sample_weight*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*

Metadata routing for sample_weight parameter in fit.

Returns:

self*object*

The updated object.

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

Set the parameters of this estimator.

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

Parameters:

****params***dict*

Estimator parameters.

Returns:

self*estimator instance*

Estimator instance.

KNNImputer#

class **sklearn.impute.KNNImputer(****, **missing_values**=nan, **n_neighbors**=5, **weights**='uniform', **metric**='nan_euclidean', **copy**=True, **add_indicator**=False, **keep_empty_features**=False***)[source]**

Imputation for completing missing values using k-Nearest Neighbors.

Each sample's missing values are imputed using the mean value from n_neighbors nearest neighbors found in the training set. Two samples are close if the features that neither is missing are close.

Read more in the User Guide.

Added in version 0.22.

Parameters:

missing_values*int, float, str, np.nan or None, default=np.nan*

The placeholder for the missing values. All occurrences of missing_values will be imputed. For pandas' dataframes with nullable integer dtypes with missing values, missing_values should be set to np.nan, since pd.NA will be converted to np.nan.

n_neighbors*int, default=5*

Number of neighboring samples to use for imputation.

weights*{'uniform', 'distance'} or callable, 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.

metric*{'nan_euclidean'} or callable, default='nan_euclidean'*

Distance metric for searching neighbors. Possible values:

- 'nan_euclidean'
- callable : a user-defined function which conforms to the definition of _pairwise_callable(X, Y, metric, **kwds). The function accepts two arrays, X and Y, and a missing_values keyword in kwds and returns a scalar distance value.

copy*bool, default=True*

If True, a copy of X will be created. If False, imputation will be done in-place whenever possible.

add_indicator*bool, default=False*

If True, a **MissingIndicator** transform will stack onto the 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.

keep_empty_features*bool, default=False*

If True, features that consist exclusively of missing values when fit is called are returned in results when transform is called. The imputed value is always 0.

Added in version 1.2.

Attributes:

indicator_*MissingIndicator*

Indicator used to add binary indicators for missing values. None if add_indicator is False.

n_features_in_*int*

Number of features seen during fit.

Added in version 0.24.

feature_names_in_*ndarray of shape (n_features_in_,)*

Names of features seen during fit. Defined only when X has feature names that are all strings.

Added in version 1.0.

See also

SimpleImputer

Univariate imputer for completing missing values with simple strategies.

IterativeImputer

Multivariate imputer that estimates values to impute for each feature with missing values from all the others.

References

- Olga Troyanskaya, Michael Cantor, Gavin Sherlock, Pat Brown, Trevor Hastie, Robert Tibshirani, David Botstein and Russ B. Altman, Missing value estimation methods for DNA microarrays, BIOINFORMATICS Vol. 17 no. 6, 2001 Pages 520-525.

Examples

```
>>> import numpy as np
>>> from sklearn.impute import KNNImputer
>>> X = [[1, 2, np.nan], [3, 4, 3], [np.nan, 6, 5], [8, 8, 7]]
>>> imputer = KNNImputer(n_neighbors=2)
>>> imputer.fit_transform(X)
array([[1. , 2. , 4. ],
       [3. , 4. , 3. ],
       [5.5, 6. , 5. ],
       [8. , 8. , 7. ]])
```

For a more detailed example see Imputing missing values before building an estimator.

fit(*X, y=None*)[source]

Fit the imputer on X.

Parameters:

X*array-like shape of (n_samples, n_features)*

Input data, where n_samples is the number of samples and n_features is the number of features.

y*Ignored*

Not used, present here for API consistency by convention.

Returns:

self*object*

The fitted KNNImputer class instance.

fit_transform(*X, y=None, **fit_params*)[source]

Fit to data, then transform it.

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

Parameters:

X*array-like of shape (n_samples, n_features)*

Input samples.

y*array-like of shape (n_samples,) or (n_samples, n_outputs), default=None*

Target values (None for unsupervised transformations).

****fit_params*dict***

Additional fit parameters.

Returns:

X_new*ndarray array of shape (n_samples, n_features_new)*

Transformed array.

get_feature_names_out(*input_features*=*None*)[source]

Get output feature names for transformation.

Parameters:

input_features*array-like of str or None, default=None*

Input features.

- If input_features is None, then feature_names_in_ is used as feature names in. If feature_names_in_ is not defined, then the following input feature names are generated: ["x0", "x1", ..., "x(n_features_in_ - 1)"].
- If input_features is an array-like, then input_features must match feature_names_in_ if feature_names_in_ is defined.

Returns:

feature_names_out*ndarray of str objects*

Transformed feature names.

get_metadata_routing()[source]

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns:

routing*MetadataRequest*

A **MetadataRequest** encapsulating routing information.

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

Get parameters for this estimator.

Parameters:

deep*bool, default=True*

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns:

params*dict*

Parameter names mapped to their values.

set_output(, transform*=*None*)[source]**

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters:

transform*("default", "pandas", "polars"), default=None*

Configure output of transform and fit_transform.

- "default": Default output format of a transformer
- "pandas": DataFrame output
- "polars": Polars output
- None: Transform configuration is unchanged

Added in version 1.4: "polars" option was added.

Returns:

self*estimator instance*

Estimator instance.

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

Set the parameters of this estimator.

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

Parameters:

****params***dict*

Estimator parameters.

Returns:

self*estimator instance*

Estimator instance.

transform(*X***)[source]**

Impute all missing values in X.

Parameters:

X*array-like of shape (n_samples, n_features)*

The input data to complete.

Returns:

X*array-like of shape (n_samples, n_output_features)*

The imputed dataset. n_output_features is the number of features that is not always missing during fit.

RandomForestClassifier

***class* sklearn.ensemble.RandomForestClassifier(*n_estimators*=100, ***, *criterion*='gini', *max_depth*=None, *min_samples_split*=2, *min_samples_leaf*=1, *min_weight_fraction_leaf*=0.0, *max_features*='sqrt', *max_leaf_nodes*=None, *min_impurity_decrease*=0.0, *bootstrap*=True, *oob_score*=False, *n_jobs*=None, *random_state*=None, *verbose*=0, *warm_start*=False, *class_weight*=None, *ccp_alpha*=0.0, *max_samples*=None, *monotonic_cst*=None)[source]**

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. Trees in the forest use the best split strategy, i.e. equivalent to passing splitter="best" to the underlying **DecisionTreeRegressor**. The sub-sample size is controlled with the max_samples parameter if bootstrap=True (default), otherwise the whole dataset is used to build each tree.

For a comparison between tree-based ensemble models see the example Comparing Random Forests and Histogram Gradient Boosting models.

Read more in the User Guide.

Parameters:

n_estimators*int, default=100*

The number of trees in the forest.

Changed in version 0.22: The default value of n_estimators changed from 10 to 100 in 0.22.

criterion*{"gini", "entropy", "log_loss"}, default="gini"*

The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see Mathematical formulation. Note: This parameter is tree-specific.

max_depth*int, 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 or float, 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 or float, 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, default=0.0*

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*{"sqrt", "log2", None}, int or float, default="sqrt"*

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 max(1, int(max_features * n_features_in_)) features are considered at each split.
- If "sqrt", then max_features=sqrt(n_features).
- If "log2", then max_features=log2(n_features).
- If None, then max_features=n_features.

Changed in version 1.1: The default of max_features changed from "auto" to "sqrt".

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.

max_leaf_nodes*int, 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, default=0.0*

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.

Added in version 0.19.

bootstrap*bool, default=True*

Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score*bool or callable, default=False*

Whether to use out-of-bag samples to estimate the generalization score. By default, **accuracy_score** is used. Provide a callable with signature metric(y_true, y_pred) to use a custom metric. Only available if bootstrap=True.

n_jobs*int, default=None*

The number of jobs to run in parallel. **fit**, **predict**, **decision_path** and **apply** are all parallelized over the trees. 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=None*

Controls both the randomness of the bootstrapping of the samples used when building trees (if bootstrap=True) and the sampling of the features to consider when looking for the best split at each node (if max_features < n_features).

See Glossary for details.

verbose*int, default=0*

Controls the verbosity when fitting and predicting.

warm_start*bool, 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 Glossary and Fitting additional trees for details.

class_weight*{"balanced", "balanced_subsample"}, dict or list of dicts, 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))

The "balanced_subsample" mode is the same as "balanced" except that weights are computed based on the bootstrap sample for every tree grown.

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.

ccp_alpha*non-negative float, default=0.0*

Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed. See Minimal Cost-Complexity Pruning for details.

Added in version 0.22.

max_samples*int or float, default=None*

If bootstrap is True, the number of samples to draw from X to train each base estimator.

- If None (default), then draw X.shape[0] samples.
- If int, then draw max_samples samples.
- If float, then draw max(round(n_samples * max_samples), 1) samples. Thus, max_samples should be in the interval (0.0, 1.0].

Added in version 0.22.

monotonic_cst*array-like of int of shape (n_features), default=None*

Indicates the monotonicity constraint to enforce on each feature.

- 1: monotonic increase
- 0: no constraint
- -1: monotonic decrease

If monotonic_cst is None, no constraints are applied.

Monotonicity constraints are not supported for:

- multiclass classifications (i.e. when n_classes > 2),
- multioutput classifications (i.e. when n_outputs_ > 1),
- classifications trained on data with missing values.

The constraints hold over the probability of the positive class.

Read more in the User Guide.

Added in version 1.4.

Attributes:

estimator_ ***DecisionTreeClassifier***

The child estimator template used to create the collection of fitted sub-estimators.

Added in version 1.2: base_estimator_ was renamed to estimator_.

estimators_ ***list of DecisionTreeClassifier***

The collection of fitted sub-estimators.

classes_ ***ndarray 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).

n_classes_ ***int or list***

The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).

n_features_in_ ***int***

Number of features seen during fit.

Added in version 0.24.

feature_names_in_ ***ndarray of shape (n_features_in_,)***

Names of features seen during fit. Defined only when X has feature names that are all strings.

Added in version 1.0.

n_outputs_ ***int***

The number of outputs when fit is performed.

feature_importances_ ***ndarray of shape (n_features,)***

The impurity-based feature importances.

oob_score_ ***float***

Score of the training dataset obtained using an out-of-bag estimate. This attribute exists only when oob_score is True.

oob_decision_function_ ***ndarray of shape (n_samples, n_classes) or (n_samples, n_classes, n_outputs)***

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. This attribute exists only when oob_score is True.

estimators_samples_*list of arrays*

The subset of drawn samples for each base estimator.

See also

sklearn.tree.DecisionTreeClassifier

A decision tree classifier.

sklearn.ensemble.ExtraTreesClassifier

Ensemble of extremely randomized tree classifiers.

sklearn.ensemble.HistGradientBoostingClassifier

A Histogram-based Gradient Boosting Classification Tree, very fast for big datasets (n_samples >= 10_000).

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, max_features=n_features and bootstrap=False, 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

L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.

Examples

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> 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 = RandomForestClassifier(max_depth=2, random_state=0)
>>> clf.fit(X, y)
RandomForestClassifier(...)
>>> print(clf.predict([[0, 0, 0, 0]]))
```

apply(*X*)[source]

Apply trees in the forest to X, return leaf indices.

Parameters:

X*(array-like, sparse matrix) of 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*ndarray of 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(*X***)[source]**

Return the decision path in the forest.

Added in version 0.18.

Parameters:

X*(array-like, sparse matrix) of 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 matrix of shape (n_samples, n_nodes)*

Return a node indicator matrix where non zero elements indicates that the samples goes through the nodes. The matrix is of CSR format.

n_nodes_ptr*ndarray of shape (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.

property **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.

property **feature_importances_**

The impurity-based feature importances.

The higher, the more important the feature. 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.

Warning: impurity-based feature importances can be misleading for high cardinality features (many unique values).
See **sklearn.inspection.permutation_importance** as an alternative.

Returns:

feature_importances_*ndarray of 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(*X, y, sample_weight=None***)[source]**

Build a forest of trees from the training set (X, y).

Parameters:

X*(array-like, sparse matrix) of shape (n_samples, n_features)*

The training 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 csc_matrix.

y*array-like of shape (n_samples,) or (n_samples, n_outputs)*

The target values (class labels in classification, real numbers in regression).

sample_weight*array-like of shape (n_samples,), default=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*

Fitted estimator.

get_metadata_routing()[source]

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns:

routing*MetadataRequest*

A **MetadataRequest** encapsulating routing information.

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

Get parameters for this estimator.

Parameters:

deep*bool, default=True*

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns:

params*dict*

Parameter names mapped to their values.

predict(*X***)[source]**

Predict class for X.

The predicted class of an input sample is a vote by the trees in the forest, weighted by their probability estimates. That is, the predicted class is the one with highest mean probability estimate across the trees.

Parameters:

X*(array-like, sparse matrix) of 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:

y*ndarray of shape (n_samples,) or (n_samples, n_outputs)*

The predicted classes.

predict_log_proba(*X*)[source]

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 trees in the forest.

Parameters:

X*{array-like, sparse matrix} of 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:

p*ndarray of shape (n_samples, n_classes), or a list of such arrays*

The class probabilities of the input samples. The order of the classes corresponds to that in the attribute classes_.

predict_proba(*X*)[source]

Predict class probabilities for X.

The predicted class probabilities of an input sample are computed as the mean predicted class probabilities of the trees in the forest. The class probability of a single tree is the fraction of samples of the same class in a leaf.

Parameters:

X*{array-like, sparse matrix} of 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:

p*ndarray of shape (n_samples, n_classes), or a list of such arrays*

The class probabilities of the input samples. The order of the classes corresponds to that in the attribute classes_.

score(*X, y, sample_weight*=*None*)[source]

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

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

Parameters:

X*array-like of shape (n_samples, n_features)*

Test samples.

y*array-like of shape (n_samples,) or (n_samples, n_outputs)*

True labels for X.

sample_weight*array-like of shape (n_samples,), default=None*

Sample weights.

Returns:

score*float*

Mean accuracy of self.predict(X) w.r.t. y.

set_fit_request(*, *sample_weight: bool | None | str = '$UNCHANGED$'*) → RandomForestClassifier[source]

Request metadata passed to the fit method.

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

The options for each parameter are:

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

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

Added in version 1.3.

Note

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

Parameters:

sample_weight*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*

Metadata routing for sample_weight parameter in fit.

Returns:

self*object*

The updated object.

set_params(params)[source]**

Set the parameters of this estimator.

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

Parameters:

****params***dict*

Estimator parameters.

Returns:

self*estimator instance*

Estimator instance.

set_score_request(*, *sample_weight: bool / None / str = '$UNCHANGED$'*) → RandomForestClassifier[source]

Request metadata passed to the score method.

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

The options for each parameter are:

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

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

Added in version 1.3.

Note

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

Parameters:

sample_weight*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*

Metadata routing for sample_weight parameter in score.

Returns:

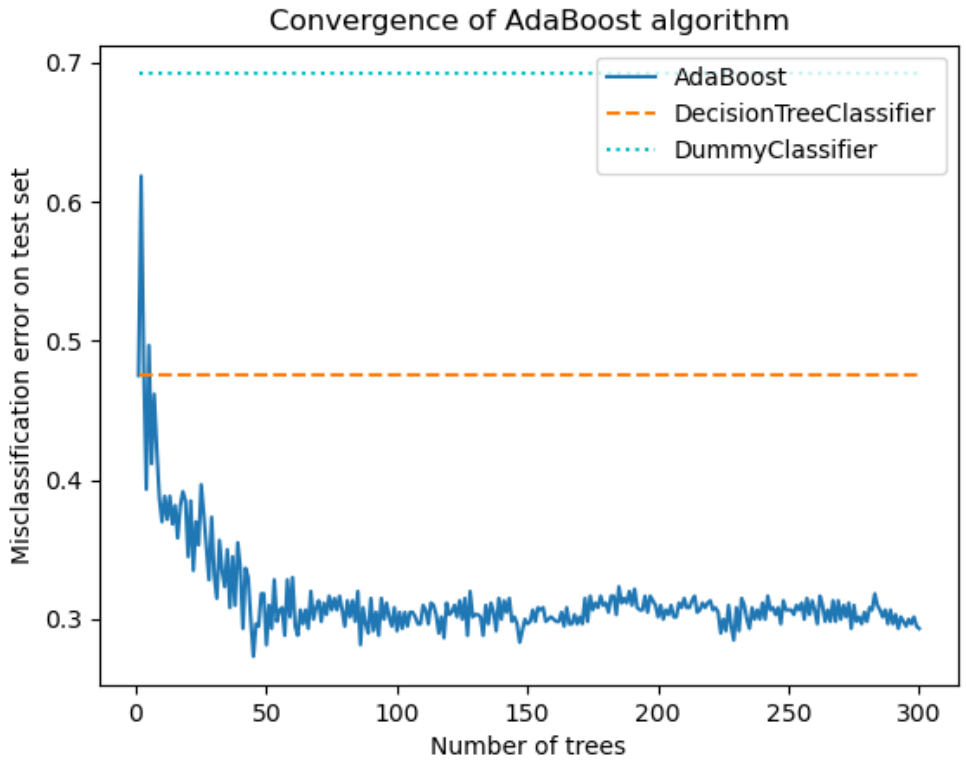
self*object*

The updated object.

1.11.7. AdaBoost

The module **sklearn.ensemble** includes the popular boosting algorithm AdaBoost, introduced in 1995 by Freund and Schapire [FS1995].

The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each so-called boosting iteration consists of applying weights $w1$, $w2$, ..., wN to each of the training samples. Initially, those weights are all set to $wi=1/N$, so that the first step simply trains a weak learner on the original data. For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence [HTF].



AdaBoost can be used both for classification and regression problems:

- For multi-class classification, **AdaBoostClassifier** implements AdaBoost.SAMME [ZZRH2009].
- For regression, **AdaBoostRegressor** implements AdaBoost.R2 [D1997].

1.11.7.1. Usage

The following example shows how to fit an AdaBoost classifier with 100 weak learners:

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import load_iris
>>> from sklearn.ensemble import AdaBoostClassifier

>>> X, y = load_iris(return_X_y=True)
>>> clf = AdaBoostClassifier(n_estimators=100, algorithm="SAMME",)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.9...
```

The number of weak learners is controlled by the parameter n_estimators. The learning_rate parameter controls the contribution of the weak learners in the final combination. By default, weak learners are decision stumps. Different weak learners can be specified through the estimator parameter. The main parameters to tune to obtain good results are n_estimators and the complexity of the base estimators (e.g., its depth max_depth or minimum required number of samples to consider a split min_samples_split).

XGBoost (eXtreme Gradient Boosting) is an open-source software library which provides a regularizing gradient boosting framework for C++, Java, Python, R, Julia, Perl, and Scala. It works on Linux, Microsoft Windows, and macOS. From the project description, it aims to provide a "Scalable, Portable and Distributed Gradient Boosting (GBM, GBRT, GBDT) Library". It runs on a single machine, as well as the distributed processing frameworks Apache Hadoop, Apache Spark, Apache Flink, and Dask.

XGBoost gained much popularity and attention in the mid-2010s as the algorithm of choice for many winning teams of machine learning competitions.

History[edit]

XGBoost initially started as a research project by Tianqi Chen as part of the Distributed (Deep) Machine Learning Community (DMLC) group. Initially, it began as a terminal application which could be configured using a libsvm configuration file. It became well known in the ML competition circles after its use in the winning solution of the Higgs Machine Learning Challenge. Soon after, the Python and R packages were built, and XGBoost now has package implementations for Java, Scala, Julia, Perl, and other languages. This brought the library to more developers and contributed to its popularity among the Kaggle community, where it has been used for a large number of competitions.

It was soon integrated with a number of other packages making it easier to use in their respective communities. It has now been integrated with scikit-learn for Python users and with the caret package for R users. It can also be integrated into Data Flow frameworks like Apache Spark, Apache Hadoop, and Apache Flink using the abstracted Rabit and XGBoost4J. XGBoost is also available on OpenCL for FPGAs. An efficient, scalable implementation of XGBoost has been published by Tianqi Chen and Carlos Guestrin.

While the XGBoost model often achieves higher accuracy than a single decision tree, it sacrifices the intrinsic interpretability of decision trees. For example, following the path that a decision tree takes to make its decision is trivial and self-explained, but following the paths of hundreds or thousands of trees is much harder.