`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, class_weight=None, ccp_alpha=0.0*) [source]

A decision tree classifier.

Read more in the User Guide.

Parameters:

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.

splitter : ***{"best", "random"}, 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, 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 : ***int, float or {"auto", "sqrt", "log2"}, 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`.

Deprecated since version 1.1: The `"auto"` option was deprecated in 1.1 and will be removed in 1.3.

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, default=None*

Controls the randomness of the estimator. The features are always randomly permuted at each split, even if `splitter` is set to `"best"`. When `max_features < n_features`, the algorithm will select `max_features` at random at each split before finding the best split among them. But the best found split may vary across different runs, even if `max_features=n_features`. That is the case, if the improvement of the criterion is identical for several splits and one split has to be selected at random. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed to an integer. See Glossary for details.

max_leaf_nodes : *int, 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, 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.

New in version 0.19.

class_weight : *dict, list of dict or "balanced", default=None*

Weights associated with classes in the form `{class_label: weight}`. If None, 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.

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.

New in version 0.22.

Attributes:

classes_ : ***ndarray of shape (n_classes,) or list of ndarray***

The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

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

Return the feature importances.

max_features_ : ***int***

The inferred value of max_features.

n_classes_ : ***int or list of int***

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***

DEPRECATED: The attribute `n_features_` is deprecated in 1.0 and will be removed in 1.2.

n_features_in_ : ***int***

Number of features seen during fit.

New 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.

New in version 1.0.

n_outputs_ : ***int***

The number of outputs when `fit` is performed.

tree_ : ***Tree instance***

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`

A decision tree regressor.

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 `predict` method operates using the `numpy.argmax` function on the outputs of `predict_proba`. This means that in case the highest predicted probabilities are tied, the classifier will predict the tied class with the lowest index in classes_.

References

1 https://en.wikipedia.org/wiki/Decision_tree_learning

2 L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984.

3 T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical Learning", Springer, 2009.

4 L. Breiman, and A. Cutler, "Random Forests", https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

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(X[, check_input])	Return the index of the leaf that each sample is predicted as.
cost_complexity_pruning_path(X, y[, ...])	Compute the pruning path during Minimal Cost-Complexity Pruning.
decision_path(X[, check_input])	Return the decision path in the tree.
fit(X, y[, sample_weight, check_input])	Build a decision tree classifier from the training set (X, y).
get_depth()	Return the depth of the decision tree.
get_n_leaves()	Return the number of leaves of the decision tree.
get_params([deep])	Get parameters for this estimator.
predict(X[, check_input])	Predict class or regression value for X.
predict_log_proba(X)	Predict class log-probabilities of the input samples X.
predict_proba(X[, check_input])	Predict class probabilities of the input samples X.
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

apply(*X*, *check_input=True*) [source]

Return the index of the leaf that each sample is predicted as.

New in version 0.17.

Parameters:

X : *{array-like, 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, default=True*

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns:

X_leaves : *array-like of 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.

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

Compute the pruning path during Minimal Cost-Complexity Pruning.

See Minimal Cost-Complexity Pruning for details on the pruning process.

Parameters:

X : *{array-like, sparse matrix} of 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 of shape (n_samples,) or (n_samples, n_outputs)*

The target values (class labels) as integers or strings.

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. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

Returns:

ccp_path : *`Bunch`*

Dictionary-like object, with the following attributes.

ccp_alphas : *ndarray*

Effective alphas of subtree during pruning.

impurities : *ndarray*

Sum of the impurities of the subtree leaves for the corresponding alpha value in `ccp_alphas`.

`decision_path`(*X*, *check_input=True*) [source]

Return the decision path in the tree.

New in version 0.18.

Parameters:

X : *{array-like, 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, default=True*

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns:

indicator : *sparse matrix of shape (n_samples, n_nodes)*

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

property **`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.

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,)*

Normalized total reduction of criteria by feature (Gini importance).

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

Build a decision tree classifier from the training set (X, y).

Parameters:

X : *{array-like, sparse matrix} of 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 of shape (n_samples,) or (n_samples, n_outputs)*
The target values (class labels) as integers or strings.

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. Splits are also ignored if they would result in any single class carrying a negative weight in either child node.

check_input : *bool, default=True*
Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns:

self : *DecisionTreeClassifier*
Fitted estimator.

`get_depth`() [source]

Return the depth of the decision tree.

The depth of a tree is the maximum distance between the root and any leaf.

Returns:

self.tree_.max_depth : *int*
The maximum depth of the tree.

`get_n_leaves`() [source]

Return the number of leaves of the decision tree.

Returns:

self.tree_.n_leaves : *int*
Number of leaves.

`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_features_`**
DEPRECATED: The attribute `n_features_` is deprecated in 1.0 and will be removed in 1.2. Use `n_features_in_` instead.

predict(*X*, *check_input=True*) [source]

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, 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, default=True*

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns:

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

The predicted classes, or the predict values.

predict_log_proba(*X*) [source]

Predict class log-probabilities of the input samples X.

Parameters:

X : *{array-like, 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:

proba : *ndarray of shape (n_samples, n_classes) or 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(*X*, *check_input=True*) [source]

Predict class probabilities of the input samples X.

The predicted class probability 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, it will be converted to `dtype=np.float32` and if a sparse matrix is provided to a sparse `csr_matrix`.

check_input : *bool, default=True*

Allow to bypass several input checking. Don't use this parameter unless you know what you do.

Returns:

proba : *ndarray of shape (n_samples, n_classes) or 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(*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)` wrt. `y`.

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

Set the parameters of this estimator.

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

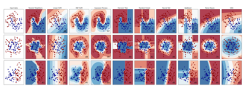
Parameters:

****params** : ***dict***
Estimator parameters.

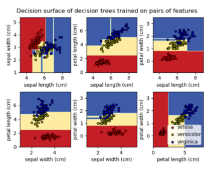
Returns:

self : ***estimator instance***
Estimator instance.

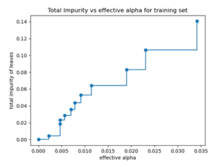
Examples using `sklearn.tree.DecisionTreeClassifier`



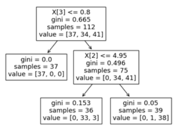
Classifier comparison



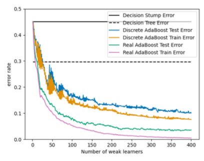
Plot the decision surface of decision trees trained on the iris dataset



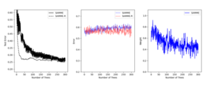
Post pruning decision trees with cost complexity pruning



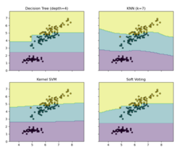
Understanding the decision tree structure



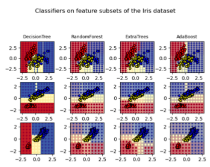
Discrete versus Real AdaBoost



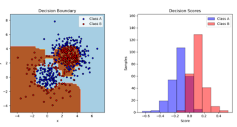
Multi-class AdaBoosted Decision Trees



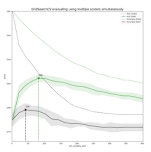
Plot the decision boundaries of a VotingClassifier



Plot the decision surfaces of ensembles of trees



Two-class AdaBoost



Demonstration of multi-metric evaluation on on the iris dataset

cross_val_score and GridSearchCV

Toggle Menu