RandomForestRegressor

.**RandomForestRegressor**(*n_estimators=100, *, criterion='squared_error', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=1.0, 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, ccp_alpha=0.0, max_samples=None*)[source]

A random forest regressor.

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. 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.

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*{"squared_error", "absolute_error", "poisson"}, default="squared_error"*
The function to measure the quality of a split. Supported criteria are "squared_error" for the mean squared error, which is equal to variance reduction as feature selection criterion, "absolute_error" for the mean absolute error, and "poisson" which uses reduction in Poisson deviance to find splits. Training using "absolute_error" is significantly slower than when using "squared_error".

New in version 0.18: Mean Absolute Error (MAE) criterion.
New in version 1.0: Poisson criterion.
Deprecated since version 1.0: Criterion "mse" was deprecated in v1.0 and will be removed in version 1.2. Use `criterion="squared_error"` which is equivalent.
Deprecated since version 1.0: Criterion "mae" was deprecated in v1.0 and will be removed in version 1.2. Use `criterion="absolute_error"` which is equivalent.

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=1.0*
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 "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None or 1.0, then `max_features=n_features`.

Note

The default of 1.0 is equivalent to bagged trees and more randomness can be achieved by setting smaller values, e.g. 0.3.

Changed in version 1.1: The default of `max_features` changed from `"auto"` to 1.0.

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.

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.

New 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, default=False*
Whether to use out-of-bag samples to estimate the generalization score. 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 the Glossary.

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.
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_samples * X.shape[0]` samples. Thus, `max_samples` should be in the interval `(0.0, 1.0]`.

New in version 0.22.
Attributes:
base_estimator_*DecisionTreeRegressor*
The child estimator template used to create the collection of fitted sub-estimators.

estimators_*list of DecisionTreeRegressor*
The collection of fitted sub-estimators.

`feature_importances_`*ndarray of shape (n_features,)*
The impurity-based feature importances.

`n_features_`*int*

DEPRECATED: Attribute `n_features_` was deprecated in version 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.

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_prediction_*ndarray of shape (n_samples,) or (n_samples, n_outputs)*
Prediction computed with out-of-bag estimate on the training set. This attribute exists only when `oob_score` is True.

See also

`sklearn.tree.DecisionTreeRegressor`
A decision tree regressor.
`sklearn.ensemble.ExtraTreesRegressor`
Ensemble of extremely randomized tree regressors.

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.

The default value `max_features="auto"` uses `n_features` rather than `n_features / 3`. The latter was originally suggested in [1], whereas the former was more recently justified empirically in [2].

References

[1]
12. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.

[2]
P. Geurts, D. Ernst., and L. Wehenkel, “Extremely randomized trees”, Machine Learning, 63(1), 3-42, 2006.

Examples

```
>>>
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_features=4, n_informative=2,
...                        random_state=0, shuffle=False)
>>> regr = RandomForestRegressor(max_depth=2, random_state=0)
>>> regr.fit(X, y)
RandomForestRegressor(...)
>>> print(regr.predict([[0, 0, 0, 0]]))
[-8.32987858]
```

Methods

`apply`(X)	Apply trees in the forest to X, return leaf indices.
`decision_path`(X)	Return the decision path in the forest.
`fit`(X, y[, sample_weight])	Build a forest of trees from the training set (X, y).
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict regression target for X.
`score`(X, y[, sample_weight])	Return the coefficient of determination of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

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.

New 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 **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`.

yarray-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_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: Attribute `n_features_` was deprecated in version 1.0 and will be removed in 1.2. Use `n_features_in_` instead.

Number of features when fitting the estimator.

predict(*X*)[source]

Predict regression target for X.

The predicted regression target of an input sample is computed as the mean predicted regression targets 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:

yndarray of shape (n_samples,) or (n_samples, n_outputs)
The predicted values.

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

Return the coefficient of determination of the prediction.

The coefficient of determination R2 is defined as (1−uv), 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 R2 score of 0.0.

Parameters:

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

yarray-like of shape (n_samples,) or (n_samples, n_outputs)
True values for `X`.

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

Returns:

score*float*
R2 of `self.predict(X)` wrt. `y`.

Notes

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

set_params(***params*)[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***

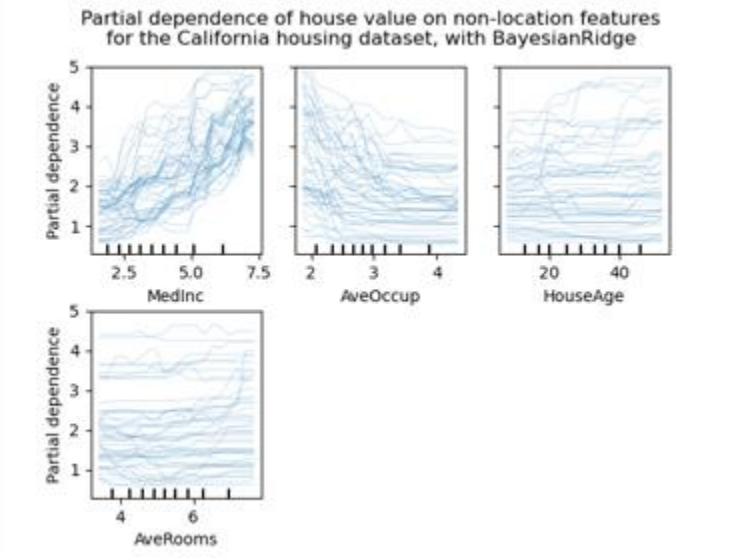
Estimator parameters.

Returns:

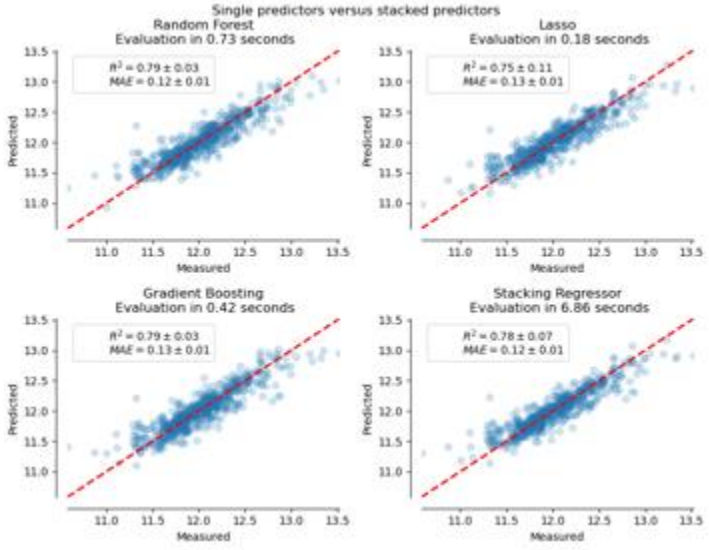
self*estimator instance*

Estimator instance.

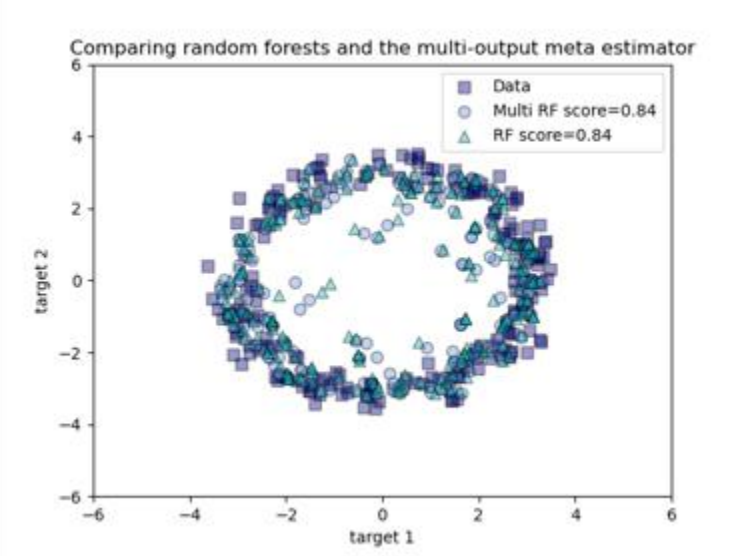
RandomForestRegressor



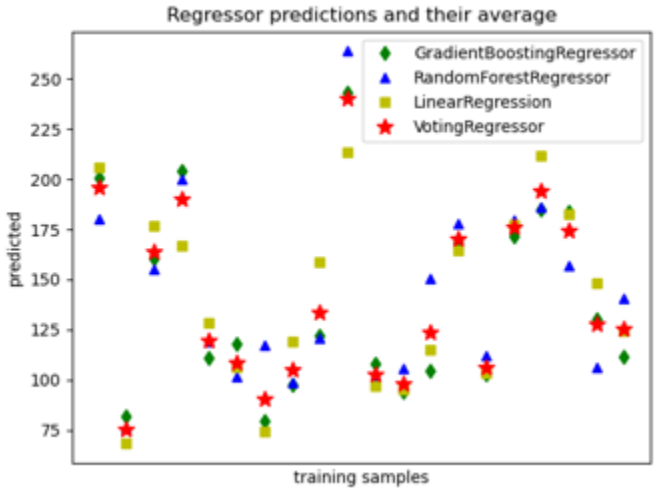
Release Highlights for scikit-learn 0.24



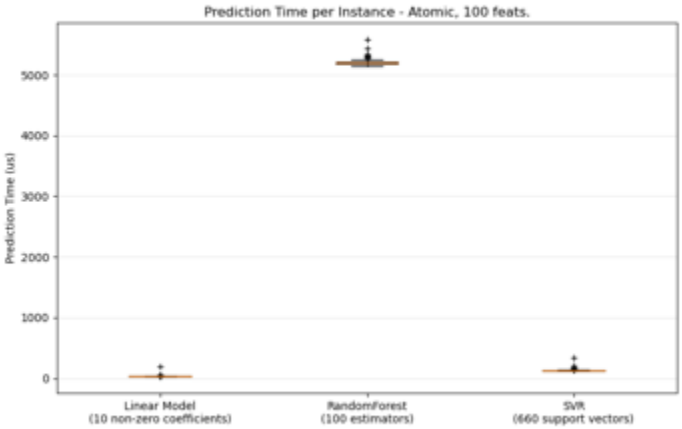
Combine predictors using stacking



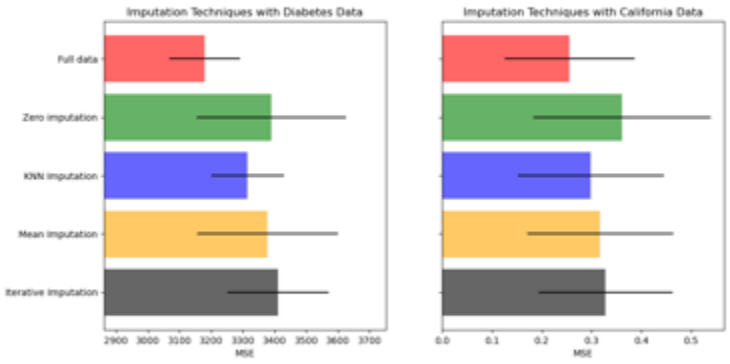
Comparing random forests and the multi-output meta estimaton



Plot individual and voting regression predictions



Prediction Latency



Imputing missing values before building an estimator