Ensemble methods

The goal of **ensemble methods** is to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator.

Two families of ensemble methods are usually distinguished:

- In **averaging methods**, the driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.

  Examples: :ref:`Bagging methods <bagging>`, :ref:`Forests of randomized trees <forest>`, ...

- By contrast, in **boosting methods**, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble.

  Examples: :ref:`AdaBoost <adaboost>`, :ref:`Gradient Tree Boosting <gradient_boosting>`, ...

Bagging meta-estimator

In ensemble algorithms, bagging methods form a class of algorithms which build several instances of a black-box estimator on random subsets of the original training set and then aggregate their individual predictions to form a final prediction. These methods are used as a way to reduce the variance of a base estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it. In many cases, bagging methods constitute a very simple way to improve with respect to a single model, without making it necessary to adapt the underlying base algorithm. As they provide a way to reduce overfitting, bagging methods work best with strong and complex models (e.g., fully developed decision trees), in contrast with boosting methods which usually work best with weak models (e.g., shallow decision trees).

Bagging methods come in many flavours but mostly differ from each other by the way they draw random subsets of the training set:

- When random subsets of the dataset are drawn as random subsets of the samples, then this algorithm is known as Pasting [B1999].
- When samples are drawn with replacement, then the method is known as Bagging [B1996].
- When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces [H1998].
- Finally, when base estimators are built on subsets of both samples and features, then the method is known as Random Patches [LG2012].

In scikit-learn, bagging methods are offered as a unified :class:`BaggingClassifier` meta-estimator (resp. :class:`BaggingRegressor`), taking as input a user-specified base estimator along with parameters specifying the strategy to draw random subsets. In particular, `max_samples` and `max_features` control the size of the subsets (in terms of samples and features), while `bootstrap` and `bootstrap_features` control whether samples and features are drawn with or without replacement. When using a subset of the available samples the generalization accuracy can be estimated with the out-of-bag samples by setting `oob_score=True`. As an example, the snippet below illustrates how to instantiate a bagging ensemble of :class:`KNeighborsClassifier` base estimators, each built on random subsets of 50% of the samples and 50% of the features.

```
>>> from sklearn.ensemble import BaggingClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> bagging = BaggingClassifier(KNeighborsClassifier(),
...                             max_samples=0.5, max_features=0.5)
```

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_bias_variance.py`

References

[**B1999**] L. Breiman, "Pasting small votes for classification in large databases and on-line", Machine Learning, 36(1), 85-103, 1999.

[**B1996**] L. Breiman, "Bagging predictors", Machine Learning, 24(2), 123-140, 1996.

[**H1998**] T. Ho, "The random subspace method for constructing decision forests", Pattern Analysis and Machine Intelligence, 20(8), 832-844, 1998.

[**LG2012**] G. Louppe and P. Geurts, "Ensembles on Random Patches", Machine Learning and Knowledge Discovery in Databases, 346-361, 2012.

Forests of randomized trees

The :mod:`sklearn.ensemble` module includes two averaging algorithms based on randomized :ref:`decision trees <tree>`: the RandomForest algorithm and the Extra-Trees method. Both algorithms are perturb-and-combine techniques [B1998] specifically designed for trees. This means a diverse set of classifiers is created by introducing randomness in the classifier construction. The prediction of the ensemble is given as the averaged prediction of the individual classifiers.

As other classifiers, forest classifiers have to be fitted with two arrays: a sparse or dense array X of shape (`n_samples, n_features`) holding the training samples, and an array Y of shape (`n_samples,`) holding the target values (class labels) for the training samples:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = RandomForestClassifier(n_estimators=10)
>>> clf = clf.fit(X, Y)
```

Like :ref:`decision trees <tree>`, forests of trees also extend to :ref:`multi-output problems <tree_multioutput>` (if Y is an array of shape (`n_samples, n_outputs`)).

Unknown interpreted text role "ref".

Random Forests

In random forests (see :class:`RandomForestClassifier` and :class:`RandomForestRegressor` classes), each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set.

<div style="border:1px solid">

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 130); *backlink*

Unknown interpreted text role "class".

</div>

<div style="border:1px solid">

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 130); *backlink*

Unknown interpreted text role "class".

</div>

Furthermore, when splitting each node during the construction of a tree, the best split is found either from all input features or a random subset of size `max_features`. (See the :ref:`parameter tuning guidelines <random_forest_parameters>`` for more details).

<div style="border:1px solid">

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 135); *backlink*

Unknown interpreted text role "ref".

</div>

The purpose of these two sources of randomness is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit high variance and tend to overfit. The injected randomness in forests yield decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant hence yielding an overall better model.

In contrast to the original publication [B2001], the scikit-learn implementation combines classifiers by averaging their probabilistic prediction, instead of letting each classifier vote for a single class.

Extremely Randomized Trees

In extremely randomized trees (see :class:`ExtraTreesClassifier` and :class:`ExtraTreesRegressor` classes), randomness goes one step further in the way splits are computed. As in random forests, a random subset of candidate features is used, but instead of looking for the most discriminative thresholds, thresholds are drawn at random for each candidate feature and the best of these randomly-generated thresholds is picked as the splitting rule. This usually allows to reduce the variance of the model a bit more, at the expense of a slightly greater increase in bias:

<div style="border:1px solid">

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 156); *backlink*

Unknown interpreted text role "class".

</div>

<div style="border:1px solid">

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 156); *backlink*

Unknown interpreted text role "class".

</div>

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.datasets import make_blobs
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.tree import DecisionTreeClassifier

>>> X, y = make_blobs(n_samples=10000, n_features=10, centers=100,
...     random_state=0)

>>> clf = DecisionTreeClassifier(max_depth=None, min_samples_split=2,
...     random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.98...

>>> clf = RandomForestClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean()
0.999...

>>> clf = ExtraTreesClassifier(n_estimators=10, max_depth=None,
...     min_samples_split=2, random_state=0)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores.mean() > 0.999
True
```

Parameters

The main parameters to adjust when using these methods is `n_estimators` and `max_features`. The former is the number of trees in the forest. The larger the better, but also the longer it will take to compute. In addition, note that results will stop getting significantly better beyond a critical number of trees. The latter is the size of the random subsets of features to consider when splitting a node. The lower the greater the reduction of variance, but also the greater the increase in bias. Empirical good default values are `max_features=1.0` or equivalently `max_features=None` (always considering all features instead of a random subset) for regression problems, and `max_features="sqrt"` (using a random subset of size `sqrt(n_features)`) for classification tasks (where `n_features` is the number of features in the data). The default value of `max_features=1.0` is equivalent to bagged trees and more randomness can be achieved by setting smaller values (e.g. 0.3 is a typical default in the literature). Good results are often achieved when setting `max_depth=None` in combination with `min_samples_split=2` (i.e., when fully developing the trees). Bear in mind though that these values are usually not optimal, and might result in models that consume a lot of RAM. The best parameter values should always be cross-validated. In addition, note that in random forests, bootstrap samples are used by default (`bootstrap=True`) while the default strategy for extra-trees is to use the whole dataset (`bootstrap=False`). When using bootstrap sampling the generalization error can be estimated on the left out or out-of-bag samples. This can be enabled by setting `oob_score=True`.

> **Note**
>
> The size of the model with the default parameters is $O(M*N*log(N))$, where M is the number of trees and N is the number of samples. In order to reduce the size of the model, you can change these parameters: `min_samples_split`, `max_leaf_nodes`, `max_depth` and `min_samples_leaf`.

Parallelization

Finally, this module also features the parallel construction of the trees and the parallel computation of the predictions through the `n_jobs` parameter. If `n_jobs=k` then computations are partitioned into `k` jobs, and run on `k` cores of the machine. If `n_jobs=-1` then all cores available on the machine are used. Note that because of inter-process communication overhead, the speedup might not be linear (i.e., using `k` jobs will unfortunately not be `k` times as fast). Significant speedup can still be achieved though when building a large number of trees, or when building a single tree requires a fair amount of time (e.g., on large datasets).

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_forest_iris.py`

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 250); *backlink***
>
> Unknown interpreted text role "ref".

- :ref:`sphx_glr_auto_examples_ensemble_plot_forest_importances_faces.py`

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 251); *backlink***
>
> Unknown interpreted text role "ref".

- :ref:`sphx_glr_auto_examples_miscellaneous_plot_multioutput_face_completion.py`

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 252); *backlink***
>
> Unknown interpreted text role "ref".

References

[B2001]   L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.

[B1998]   L. Breiman, "Arcing Classifiers", Annals of Statistics 1998.

- P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

Feature importance evaluation

The relative rank (i.e. depth) of a feature used as a decision node in a tree can be used to assess the relative importance of that feature with respect to the predictability of the target variable. Features used at the top of the tree contribute to the final prediction decision of a larger fraction of the input samples. The **expected fraction of the samples** they contribute to can thus be used as an estimate of the **relative importance of the features**. In scikit-learn, the fraction of samples a feature contributes to is combined with the decrease in impurity from splitting them to create a normalized estimate of the predictive power of that feature.

By **averaging** the estimates of predictive ability over several randomized trees one can **reduce the variance** of such an estimate and use it for feature selection. This is known as the mean decrease in impurity, or MDI. Refer to [L2014] for more information on MDI and feature importance evaluation with Random Forests.

> **Warning**
>
> The impurity-based feature importances computed on tree-based models suffer from two flaws that can lead to misleading conclusions. First they are computed on statistics derived from the training dataset and therefore **do not necessarily inform us on which features are most important to make good predictions on held-out dataset**. Secondly, **they favor high cardinality features**, that is features with many unique values.

:ref:`permutation_importance` is an alternative to impurity-based feature importance that does not suffer from these flaws. These two methods of obtaining feature importance are explored in: :ref:`sphx_glr_auto_examples_inspection_plot_permutation_importance.py`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 287);** *backlink*
>
> Unknown interpreted text role "ref".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 287);** *backlink*
>
> Unknown interpreted text role "ref".

The following example shows a color-coded representation of the relative importances of each individual pixel for a face recognition task using a :class:`ExtraTreesClassifier` model.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 298);** *backlink*
>
> Unknown interpreted text role "class".

In practice those estimates are stored as an attribute named `feature_importances_` on the fitted model. This is an array with shape `(n_features,)` whose values are positive and sum to 1.0. The higher the value, the more important is the contribution of the matching feature to the prediction function.

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_forest_importances_faces.py`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 315);** *backlink*
>
> Unknown interpreted text role "ref".

- :ref:`sphx_glr_auto_examples_ensemble_plot_forest_importances.py`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 316);** *backlink*
>
> Unknown interpreted text role "ref".

References

[L2014] G. Louppe, "Understanding Random Forests: From Theory to Practice", PhD Thesis, U. of Liege, 2014.

Totally Random Trees Embedding

:class:`RandomTreesEmbedding` implements an unsupervised transformation of the data. Using a forest of completely random trees, :class:`RandomTreesEmbedding` encodes the data by the indices of the leaves a data point ends up in. This index is then encoded in a one-of-K manner, leading to a high dimensional, sparse binary coding. This coding can be computed very efficiently and can then be used as a basis for other learning tasks. The size and sparsity of the code can be influenced by choosing the number of trees and the maximum depth per tree. For each tree in the ensemble, the coding contains one entry of one. The size of the coding is at most `n_estimators * 2 ** max_depth`, the maximum number of leaves in the forest.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 329);** *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 329);** *backlink*
>
> Unknown interpreted text role "class".

As neighboring data points are more likely to lie within the same leaf of a tree, the transformation performs an implicit, non-parametric density estimation.

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_random_forest_embedding.py`

- :ref:`sphx_glr_auto_examples_manifold_plot_lle_digits.py` compares non-linear dimensionality reduction techniques on handwritten digits.

- :ref:`sphx_glr_auto_examples_ensemble_plot_feature_transformation.py` compares supervised and unsupervised tree based feature transformations.

AdaBoost

The module :mod:`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 consist of applying weights w_1, w_2, ..., w_N to each of the training samples. Initially, those weights are all set to $w_i = 1/N$, 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, :class:`AdaBoostClassifier` implements AdaBoost-SAMME and AdaBoost-SAMME.R [ZZRH2009].

- For regression, :class:`AdaBoostRegressor` implements AdaBoost.R2 [D1997].

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)
>>> 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 `base_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`).

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_adaboost_hastie_10_2.py` compares the classification error of a decision stump, decision tree, and a boosted decision stump using AdaBoost-SAMME and AdaBoost-SAMME.R.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 426**); *backlink*
>
> Unknown interpreted text role "ref".

- :ref:`sphx_glr_auto_examples_ensemble_plot_adaboost_multiclass.py` shows the performance of AdaBoost-SAMME and AdaBoost-SAMME.R on a multi-class problem.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 430**); *backlink*
>
> Unknown interpreted text role "ref".

- :ref:`sphx_glr_auto_examples_ensemble_plot_adaboost_twoclass.py` shows the decision boundary and decision function values for a non-linearly separable two-class problem using AdaBoost-SAMME.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 433**); *backlink*
>
> Unknown interpreted text role "ref".

- :ref:`sphx_glr_auto_examples_ensemble_plot_adaboost_regression.py` demonstrates regression with the AdaBoost.R2 algorithm.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 437**); *backlink*
>
> Unknown interpreted text role "ref".

References

[**FS1995**] Y. Freund, and R. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting", 1997.

[**ZZRH2009**] J. Zhu, H. Zou, S. Rosset, T. Hastie. "Multi-class AdaBoost", 2009.

[**D1997**] H. Drucker. "Improving Regressors using Boosting Techniques", 1997.

[**HTF**] (*1*,*2*,*3*) T. Hastie, R. Tibshirani and J. Friedman, "Elements of Statistical Learning Ed. 2", Springer, 2009.

Gradient Tree Boosting

Gradient Tree Boosting or Gradient Boosted Decision Trees (GBDT) is a generalization of boosting to arbitrary differentiable loss functions. GBDT is an accurate and effective off-the-shelf procedure that can be used for both regression and classification problems in a variety of areas including Web search ranking and ecology.

The module :mod:`sklearn.ensemble` provides methods for both classification and regression via gradient boosted decision trees.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 467**); *backlink*
>
> Unknown interpreted text role "mod".

> **Note**

Scikit-learn 0.21 introduces two new implementations of gradient boosting trees, namely :class:`HistGradientBoostingClassifier` and :class:`HistGradientBoostingRegressor`, inspired by LightGBM (See [LightGBM]).

These histogram-based estimators can be **orders of magnitude faster** than :class:`GradientBoostingClassifier` and :class:`GradientBoostingRegressor` when the number of samples is larger than tens of thousands of samples.

They also have built-in support for missing values, which avoids the need for an imputer.

These estimators are described in more detail below in :ref:`histogram_based_gradient_boosting`.

The following guide focuses on :class:`GradientBoostingClassifier` and :class:`GradientBoostingRegressor`, which might be preferred for small sample sizes since binning may lead to split points that are too approximate in this setting.

The usage and the parameters of :class:`GradientBoostingClassifier` and :class:`GradientBoostingRegressor` are described below. The 2 most important parameters of these estimators are *n_estimators* and *learning_rate*.

Classification

:class:`GradientBoostingClassifier` supports both binary and multi-class classification. The following example shows how to fit a gradient boosting classifier with 100 decision stumps as weak learners:

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> X_train, X_test = X[:2000], X[2000:]
>>> y_train, y_test = y[:2000], y[2000:]

>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.913...
```

The number of weak learners (i.e. regression trees) is controlled by the parameter `n_estimators`; :ref:`The size of each tree <gradient_boosting_tree_size>` can be controlled either by setting the tree depth via `max_depth` or by setting the number of leaf nodes via `max_leaf_nodes`. The `learning_rate` is a hyper-parameter in the range (0.0, 1.0] that controls overfitting via :ref:`shrinkage <gradient_boosting_shrinkage>` .

Note

Classification with more than 2 classes requires the induction of `n_classes` regression trees at each iteration, thus, the total number of induced trees equals `n_classes * n_estimators`. For datasets with a large number of classes we strongly recommend to use :class:`HistGradientBoostingClassifier` as an alternative to :class:`GradientBoostingClassifier` .

Regression

:class:`GradientBoostingRegressor` supports a number of :ref:`different loss functions <gradient_boosting_loss>` for regression which can be specified via the argument `loss`; the default loss function for regression is squared error (`'squared_error'`).

```
>>> import numpy as np
>>> from sklearn.metrics import mean_squared_error
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor

>>> X, y = make_friedman1(n_samples=1200, random_state=0, noise=1.0)
>>> X_train, X_test = X[:200], X[200:]
>>> y_train, y_test = y[:200], y[200:]
>>> est = GradientBoostingRegressor(
...     n_estimators=100, learning_rate=0.1, max_depth=1, random_state=0,
...     loss='squared_error'
... ).fit(X_train, y_train)
>>> mean_squared_error(y_test, est.predict(X_test))
5.00...
```

The figure below shows the results of applying :class:`GradientBoostingRegressor` with least squares loss and 500 base learners to

the diabetes dataset (:func:`sklearn.datasets.load_diabetes`). The plot on the left shows the train and test error at each iteration. The train error at each iteration is stored in the :attr:`~GradientBoostingRegressor.train_score_` attribute of the gradient boosting model. The test error at each iterations can be obtained via the :meth:`~GradientBoostingRegressor.staged_predict` method which returns a generator that yields the predictions at each stage. Plots like these can be used to determine the optimal number of trees (i.e. `n_estimators`) by early stopping.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 563**); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 563**); *backlink*
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 563**); *backlink*
>
> Unknown interpreted text role "attr".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 563**); *backlink*
>
> Unknown interpreted text role "meth".

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_gradient_boosting_regression.py`

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 581**); *backlink*
  >
  > Unknown interpreted text role "ref".

- :ref:`sphx_glr_auto_examples_ensemble_plot_gradient_boosting_oob.py`

  > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 582**); *backlink*
  >
  > Unknown interpreted text role "ref".

Fitting additional weak-learners

Both :class:`GradientBoostingRegressor` and :class:`GradientBoostingClassifier` support `warm_start=True` which allows you to add more estimators to an already fitted model.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 589**); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 589**); *backlink*
>
> Unknown interpreted text role "class".

```
>>> _ = est.set_params(n_estimators=200, warm_start=True)  # set warm_start and new nr of trees
>>> _ = est.fit(X_train, y_train) # fit additional 100 trees to est
>>> mean_squared_error(y_test, est.predict(X_test))
3.84...
```

Controlling the tree size

The size of the regression tree base learners defines the level of variable interactions that can be captured by the gradient boosting model. In general, a tree of depth `h` can capture interactions of order `h`. There are two ways in which the size of the individual regression trees can be controlled.

If you specify `max_depth=h` then complete binary trees of depth `h` will be grown. Such trees will have (at most) $2**h$ leaf nodes and $2**h - 1$ split nodes.

Alternatively, you can control the tree size by specifying the number of leaf nodes via the parameter `max_leaf_nodes`. In this case, trees will be grown using best-first search where nodes with the highest improvement in impurity will be expanded first. A tree with `max_leaf_nodes=k` has `k - 1` split nodes and thus can model interactions of up to order `max_leaf_nodes - 1`.

We found that `max_leaf_nodes=k` gives comparable results to `max_depth=k-1` but is significantly faster to train at the expense of

a slightly higher training error. The parameter `max_leaf_nodes` corresponds to the variable J in the chapter on gradient boosting in [F2001]_ and is related to the parameter `interaction.depth` in R's gbm package where `max_leaf_nodes == interaction.depth + 1`.

Mathematical formulation

We first present GBRT for regression, and then detail the classification case.

Regression

GBRT regressors are additive models whose prediction y_i for a given input x_i is of the following form:

$$\hat{y_i} = F_M(x_i) = \sum_{m=1}^{M} h_m(x_i)$$

where the h_m are estimators called *weak learners* in the context of boosting. Gradient Tree Boosting uses :ref:`decision tree regressors <tree>` of fixed size as weak learners. The constant M corresponds to the *n_estimators* parameter.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 645); *backlink***
>
> Unknown interpreted text role "ref".

Similar to other boosting algorithms, a GBRT is built in a greedy fashion:

$$F_m(x) = F_{m-1}(x) + h_m(x),$$

where the newly added tree h_m is fitted in order to minimize a sum of losses L_m, given the previous ensemble F_{m-1}:

$$h_m = \arg\min_h L_m = \arg\min_h \sum_{i=1}^{n} l(y_i, F_{m-1}(x_i) + h(x_i)),$$

where $l(y_i, F(x_i))$ is defined by the *loss* parameter, detailed in the next section.

By default, the initial model F_0 is chosen as the constant that minimizes the loss: for a least-squares loss, this is the empirical mean of the target values. The initial model can also be specified via the `init` argument.

Using a first-order Taylor approximation, the value of l can be approximated as follows:

$$l(y_i, F_{m-1}(x_i) + h_m(x_i)) \approx l(y_i, F_{m-1}(x_i)) + h_m(x_i) \left[\frac{\partial l(y_i, F(x_i))}{\partial F(x_i)} \right]_{F = F_{m-1}}.$$

> **Note**
>
> Briefly, a first-order Taylor approximation says that $l(z) \approx l(a) + (z - a) \frac{\partial l(a)}{\partial a}$. Here, z corresponds to $F_{m-1}(x_i) + h_m(x_i)$, and a corresponds to $F_{m-1}(x_i)$

The quantity $\left[\frac{\partial l(y_i, F(x_i))}{\partial F(x_i)} \right]_{F = F_{m-1}}$ is the derivative of the loss with respect to its second parameter, evaluated at $F_{m-1}(x)$. It is easy to compute for any given $F_{m-1}(x_i)$ in a closed form since the loss is differentiable. We will denote it by g_i.

Removing the constant terms, we have:

$$h_m \approx \arg\min_h \sum_{i=1}^{n} h(x_i) g_i$$

This is minimized if $h(x_i)$ is fitted to predict a value that is proportional to the negative gradient $-g_i$. Therefore, at each iteration, **the estimator h_m is fitted to predict the negative gradients of the samples**. The gradients are updated at each iteration. This can be considered as some kind of gradient descent in a functional space.

> **Note**
>
> For some losses, e.g. the least absolute deviation (LAD) where the gradients are ±1, the values predicted by a fitted h_m are not accurate enough: the tree can only output integer values. As a result, the leaves values of the tree h_m are modified once the tree is fitted, such that the leaves values minimize the loss L_m. The update is loss-dependent: for the LAD loss, the value of a leaf is updated to the median of the samples in that leaf.

Classification

Gradient boosting for classification is very similar to the regression case. However, the sum of the trees $F_M(x_i) = \sum_m h_m(x_i)$ is not homogeneous to a prediction: it cannot be a class, since the trees predict continuous values.

The mapping from the value $F_M(x_i)$ to a class or a probability is loss-dependent. For the log-loss, the probability that x_i belongs to

the positive class is modeled as $p(y_i = 1|x_i) = \sigma(F_M(x_i))$ where σ is the sigmoid or expit function.

For multiclass classification, K trees (for K classes) are built at each of the M iterations. The probability that x_i belongs to class k is modeled as a softmax of the $F_{M,k}(x_i)$ values.

Note that even for a classification task, the h_m sub-estimator is still a regressor, not a classifier. This is because the sub-estimators are trained to predict (negative) *gradients*, which are always continuous quantities.

Loss Functions

The following loss functions are supported and can be specified using the parameter `loss`:

- Regression
  - Squared error (`'squared_error'`): The natural choice for regression due to its superior computational properties. The initial model is given by the mean of the target values.
  - Least absolute deviation (`'lad'`): A robust loss function for regression. The initial model is given by the median of the target values.
  - Huber (`'huber'`): Another robust loss function that combines least squares and least absolute deviation; use `alpha` to control the sensitivity with regards to outliers (see [F2001]_ for more details).
  - Quantile (`'quantile'`): A loss function for quantile regression. Use `0 < alpha < 1` to specify the quantile. This loss function can be used to create prediction intervals (see :ref:`sphx_glr_auto_examples_ensemble_plot_gradient_boosting_quantile.py`).

    > **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 760); *backlink***
    >
    > Unknown interpreted text role "ref".

- Classification
  - Binary log-loss (`'log-loss'`): The binomial negative log-likelihood loss function for binary classification. It provides probability estimates. The initial model is given by the log odds-ratio.
  - Multi-class log-loss (`'log-loss'`): The multinomial negative log-likelihood loss function for multi-class classification with `n_classes` mutually exclusive classes. It provides probability estimates. The initial model is given by the prior probability of each class. At each iteration `n_classes` regression trees have to be constructed which makes GBRT rather inefficient for data sets with a large number of classes.
  - Exponential loss (`'exponential'`): The same loss function as :class:`AdaBoostClassifier`. Less robust to mislabeled examples than `'log-loss'`; can only be used for binary classification.

    > **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 778); *backlink***
    >
    > Unknown interpreted text role "class".

Shrinkage via learning rate

[F2001]_ proposed a simple regularization strategy that scales the contribution of each weak learner by a constant factor v:

$$F_m(x) = F_{m-1}(x) + v h_m(x)$$

The parameter v is also called the **learning rate** because it scales the step length the gradient descent procedure; it can be set via the `learning_rate` parameter.

The parameter `learning_rate` strongly interacts with the parameter `n_estimators`, the number of weak learners to fit. Smaller values of `learning_rate` require larger numbers of weak learners to maintain a constant training error. Empirical evidence suggests that small values of `learning_rate` favor better test error. [HTF] recommend to set the learning rate to a small constant (e.g. `learning_rate <= 0.1`) and choose `n_estimators` by early stopping. For a more detailed discussion of the interaction between `learning_rate` and `n_estimators` see [R2007].

Subsampling

[F1999] proposed stochastic gradient boosting, which combines gradient boosting with bootstrap averaging (bagging). At each iteration the base classifier is trained on a fraction `subsample` of the available training data. The subsample is drawn without replacement. A typical value of `subsample` is 0.5.

The figure below illustrates the effect of shrinkage and subsampling on the goodness-of-fit of the model. We can clearly see that shrinkage outperforms no-shrinkage. Subsampling with shrinkage can further increase the accuracy of the model. Subsampling without shrinkage, on the other hand, does poorly.

Another strategy to reduce the variance is by subsampling the features analogous to the random splits in :class:`RandomForestClassifier`. The number of subsampled features can be controlled via the `max_features` parameter.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-`**

> **Note**
>
> Using a small `max_features` value can significantly decrease the runtime.

Stochastic gradient boosting allows to compute out-of-bag estimates of the test deviance by computing the improvement in deviance on the examples that are not included in the bootstrap sample (i.e. the out-of-bag examples). The improvements are stored in the attribute :attr:`~GradientBoostingRegressor.oob_improvement_`. `oob_improvement_[i]` holds the improvement in terms of the loss on the OOB samples if you add the i-th stage to the current predictions. Out-of-bag estimates can be used for model selection, for example to determine the optimal number of iterations. OOB estimates are usually very pessimistic thus we recommend to use cross-validation instead and only use OOB if cross-validation is too time consuming.

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_gradient_boosting_regularization.py`

- :ref:`sphx_glr_auto_examples_ensemble_plot_gradient_boosting_oob.py`

- :ref:`sphx_glr_auto_examples_ensemble_plot_ensemble_oob.py`

Interpretation with feature importance

Individual decision trees can be interpreted easily by simply visualizing the tree structure. Gradient boosting models, however, comprise hundreds of regression trees thus they cannot be easily interpreted by visual inspection of the individual trees. Fortunately, a number of techniques have been proposed to summarize and interpret gradient boosting models.

Often features do not contribute equally to predict the target response; in many situations the majority of the features are in fact irrelevant. When interpreting a model, the first question usually is: what are those important features and how do they contributing in predicting the target response?

Individual decision trees intrinsically perform feature selection by selecting appropriate split points. This information can be used to measure the importance of each feature; the basic idea is: the more often a feature is used in the split points of a tree the more important that feature is. This notion of importance can be extended to decision tree ensembles by simply averaging the impurity-based feature importance of each tree (see :ref:`random_forest_feature_importance` for more details).

The feature importance scores of a fit gradient boosting model can be accessed via the `feature_importances_` property:

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X, y)
>>> clf.feature_importances_
array([0.10..., 0.10..., 0.11..., ...
```

Note that this computation of feature importance is based on entropy, and it is distinct from :func:`sklearn.inspection.permutation_importance` which is based on permutation of the features.

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_gradient_boosting_regression.py`

Histogram-Based Gradient Boosting

Scikit-learn 0.21 introduced two new implementations of gradient boosting trees, namely :class:`HistGradientBoostingClassifier` and :class:`HistGradientBoostingRegressor`, inspired by LightGBM (See [LightGBM]).

These histogram-based estimators can be **orders of magnitude faster** than :class:`GradientBoostingClassifier` and :class:`GradientBoostingRegressor` when the number of samples is larger than tens of thousands of samples.

They also have built-in support for missing values, which avoids the need for an imputer.

These fast estimators first bin the input samples X into integer-valued bins (typically 256 bins) which tremendously reduces the number of splitting points to consider, and allows the algorithm to leverage integer-based data structures (histograms) instead of relying on sorted continuous values when building the trees. The API of these estimators is slightly different, and some of the features from :class:`GradientBoostingClassifier` and :class:`GradientBoostingRegressor` are not yet supported, for instance some loss functions.

Examples:

- :ref:`sphx_glr_auto_examples_inspection_plot_partial_dependence.py`

Usage

Most of the parameters are unchanged from :class:`GradientBoostingClassifier` and :class:`GradientBoostingRegressor`. One exception is the `max_iter` parameter that replaces `n_estimators`, and controls the number of iterations of the boosting process:

Unknown interpreted text role "class".

```
>>> from sklearn.ensemble import HistGradientBoostingClassifier
>>> from sklearn.datasets import make_hastie_10_2

>>> X, y = make_hastie_10_2(random_state=0)
>>> X_train, X_test = X[:2000], X[2000:]
>>> y_train, y_test = y[:2000], y[2000:]

>>> clf = HistGradientBoostingClassifier(max_iter=100).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.8965
```

Available losses for regression are 'squared_error', 'absolute_error', which is less sensitive to outliers, and 'poisson', which is well suited to model counts and frequencies. For classification, 'log_loss' is the only option. For binary classification it uses the binary log loss, also kown as binomial deviance or binary cross-entropy. For *n_classes >= 3*, it uses the multi-class log loss function, with multinomial deviance and categorical cross-entropy as alternative names. The appropriate loss version is selected based on :term:`y` passed to :term:`fit`.

The size of the trees can be controlled through the `max_leaf_nodes`, `max_depth`, and `min_samples_leaf` parameters.

The number of bins used to bin the data is controlled with the `max_bins` parameter. Using less bins acts as a form of regularization. It is generally recommended to use as many bins as possible, which is the default.

The `l2_regularization` parameter is a regularizer on the loss function and corresponds to λ in equation (2) of [XGBoost].

Note that **early-stopping is enabled by default if the number of samples is larger than 10,000**. The early-stopping behaviour is controlled via the `early-stopping`, `scoring`, `validation_fraction`, `n_iter_no_change`, and `tol` parameters. It is possible to early-stop using an arbitrary :term:`scorer`, or just the training or validation loss. Note that for technical reasons, using a scorer is significantly slower than using the loss. By default, early-stopping is performed if there are at least 10,000 samples in the training set, using the validation loss.

Missing values support

:class:`HistGradientBoostingClassifier` and :class:`HistGradientBoostingRegressor` have built-in support for missing values (NaNs).

During training, the tree grower learns at each split point whether samples with missing values should go to the left or right child, based on the potential gain. When predicting, samples with missing values are assigned to the left or right child consequently:

```
>>> from sklearn.ensemble import HistGradientBoostingClassifier
>>> import numpy as np

>>> X = np.array([0, 1, 2, np.nan]).reshape(-1, 1)
>>> y = [0, 0, 1, 1]

>>> gbdt = HistGradientBoostingClassifier(min_samples_leaf=1).fit(X, y)
>>> gbdt.predict(X)
array([0, 0, 1, 1])
```

When the missingness pattern is predictive, the splits can be done on whether the feature value is missing or not:

```
>>> X = np.array([0, np.nan, 1, 2, np.nan]).reshape(-1, 1)
>>> y = [0, 1, 0, 0, 1]
```

```
>>> gbdt = HistGradientBoostingClassifier(min_samples_leaf=1,
...                                        max_depth=2,
...                                        learning_rate=1,
...                                        max_iter=1).fit(X, y)
>>> gbdt.predict(X)
array([0, 1, 0, 0, 1])
```

If no missing values were encountered for a given feature during training, then samples with missing values are mapped to whichever child has the most samples.

Sample weight support

:class:`HistGradientBoostingClassifier` and :class:`HistGradientBoostingRegressor` sample support weights during :term:`fit`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1020);** *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1020);** *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1020);** *backlink*
>
> Unknown interpreted text role "term".

The following toy example demonstrates how the model ignores the samples with zero sample weights:

```
>>> X = [[1, 0],
...      [1, 0],
...      [1, 0],
...      [0, 1]]
>>> y = [0, 0, 1, 0]
>>> # ignore the first 2 training samples by setting their weight to 0
>>> sample_weight = [0, 0, 1, 1]
>>> gb = HistGradientBoostingClassifier(min_samples_leaf=1)
>>> gb.fit(X, y, sample_weight=sample_weight)
HistGradientBoostingClassifier(...)
>>> gb.predict([[1, 0]])
array([1])
>>> gb.predict_proba([[1, 0]])[0, 1]
0.99...
```

As you can see, the *[1, 0]* is comfortably classified as *1* since the first two samples are ignored due to their sample weights.

Implementation detail: taking sample weights into account amounts to multiplying the gradients (and the hessians) by the sample weights. Note that the binning stage (specifically the quantiles computation) does not take the weights into account.

Categorical Features Support

:class:`HistGradientBoostingClassifier` and :class:`HistGradientBoostingRegressor` have native support for categorical features: they can consider splits on non-ordered, categorical data.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1055);** *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1055);** *backlink*
>
> Unknown interpreted text role "class".

For datasets with categorical features, using the native categorical support is often better than relying on one-hot encoding (:class:`~sklearn.preprocessing.OneHotEncoder`), because one-hot encoding requires more tree depth to achieve equivalent splits. It is also usually better to rely on the native categorical support rather than to treat categorical features as continuous (ordinal), which happens for ordinal-encoded categorical data, since categories are nominal quantities where order does not matter.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1059);** *backlink*
>
> Unknown interpreted text role "class".

To enable categorical support, a boolean mask can be passed to the *categorical_features* parameter, indicating which feature is categorical. In the following, the first feature will be treated as categorical and the second feature as numerical:

```
>>> gbdt = HistGradientBoostingClassifier(categorical_features=[True, False])
```

Equivalently, one can pass a list of integers indicating the indices of the categorical features:

```
>>> gbdt = HistGradientBoostingClassifier(categorical_features=[0])
```

The cardinality of each categorical feature should be less than the *max_bins* parameter, and each categorical feature is expected to be encoded in *[0, max_bins - 1]*. To that end, it might be useful to pre-process the data with an :class:`~sklearn.preprocessing.OrdinalEncoder` as done in :ref:`sphx_glr_auto_examples_ensemble_plot_gradient_boosting_categorical.py`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1080**); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1080**); *backlink*
>
> Unknown interpreted text role "ref".

If there are missing values during training, the missing values will be treated as a proper category. If there are no missing values during training, then at prediction time, missing values are mapped to the child node that has the most samples (just like for continuous features). When predicting, categories that were not seen during fit time will be treated as missing values.

Split finding with categorical features: The canonical way of considering categorical splits in a tree is to consider all of the $2^{K-1} - 1$ partitions, where K is the number of categories. This can quickly become prohibitive when K is large. Fortunately, since gradient boosting trees are always regression trees (even for classification problems), there exist a faster strategy that can yield equivalent splits. First, the categories of a feature are sorted according to the variance of the target, for each category k. Once the categories are sorted, one can consider *continuous partitions*, i.e. treat the categories as if they were ordered continuous values (see Fisher [Fisher1958] for a formal proof). As a result, only $K - 1$ splits need to be considered instead of $2^{K-1} - 1$. The initial sorting is a $O(K\log(K))$ operation, leading to a total complexity of $O(K\log(K) + K)$, instead of $O(2^K)$.

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_gradient_boosting_categorical.py`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1110**); *backlink*
>
> Unknown interpreted text role "ref".

Monotonic Constraints

Depending on the problem at hand, you may have prior knowledge indicating that a given feature should in general have a positive (or negative) effect on the target value. For example, all else being equal, a higher credit score should increase the probability of getting approved for a loan. Monotonic constraints allow you to incorporate such prior knowledge into the model.

A positive monotonic constraint is a constraint of the form:

$x_1 \leq x_1'$ ⟹ $F(x_1, x_2) \leq F(x_1', x_2)$, where F is the predictor with two features.

Similarly, a negative monotonic constraint is of the form:

$x_1 \leq x_1'$ ⟹ $F(x_1, x_2) \geq F(x_1', x_2)$.

Note that monotonic constraints only constraint the output "all else being equal". Indeed, the following relation **is not enforced** by a positive constraint: $x_1 \leq x_1'$ ⟹ $F(x_1, x_2) \leq F(x_1', x_2')$.

You can specify a monotonic constraint on each feature using the *monotonic_cst* parameter. For each feature, a value of 0 indicates no constraint, while -1 and 1 indicate a negative and positive constraint, respectively:

```
>>> from sklearn.ensemble import HistGradientBoostingRegressor

... # positive, negative, and no constraint on the 3 features
>>> gbdt = HistGradientBoostingRegressor(monotonic_cst=[1, -1, 0])
```

In a binary classification context, imposing a monotonic constraint means that the feature is supposed to have a positive / negative effect on the probability to belong to the positive class. Monotonic constraints are not supported for multiclass context.

> **Note**
>
> Since categories are unordered quantities, it is not possible to enforce monotonic constraints on categorical features.

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_monotonic_constraints.py`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1158**); *backlink*
>
> Unknown interpreted text role "ref".

Low-level parallelism

:class:`HistGradientBoostingClassifier` and :class:`HistGradientBoostingRegressor` have implementations that use OpenMP for

parallelization through Cython. For more details on how to control the number of threads, please refer to our :ref:`parallelism` notes.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1163**); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1163**); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1163**); *backlink*

Unknown interpreted text role "ref".

The following parts are parallelized:

- mapping samples from real values to integer-valued bins (finding the bin thresholds is however sequential)
- building histograms is parallelized over features
- finding the best split point at a node is parallelized over features
- during fit, mapping samples into the left and right children is parallelized over samples
- gradient and hessians computations are parallelized over samples
- predicting is parallelized over samples

Why it's faster

The bottleneck of a gradient boosting procedure is building the decision trees. Building a traditional decision tree (as in the other GBDTs :class:`GradientBoostingClassifier` and :class:`GradientBoostingRegressor`) requires sorting the samples at each node (for each feature). Sorting is needed so that the potential gain of a split point can be computed efficiently. Splitting a single node has thus a complexity of $O(n_{\text{features}} \times n \log(n))$ where n is the number of samples at the node.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1182**); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1182**); *backlink*

Unknown interpreted text role "class".

:class:`HistGradientBoostingClassifier` and :class:`HistGradientBoostingRegressor`, in contrast, do not require sorting the feature values and instead use a data-structure called a histogram, where the samples are implicitly ordered. Building a histogram has a $O(n)$ complexity, so the node splitting procedure has a $O(n_{\text{features}} \times n)$ complexity, much smaller than the previous one. In addition, instead of considering n split points, we here consider only `max_bins` split points, which is much smaller.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1191**); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1191**); *backlink*

Unknown interpreted text role "class".

In order to build histograms, the input data X needs to be binned into integer-valued bins. This binning procedure does require sorting the feature values, but it only happens once at the very beginning of the boosting process (not at each node, like in :class:`GradientBoostingClassifier` and :class:`GradientBoostingRegressor`).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1201**); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1201**); *backlink*

Unknown interpreted text role "class".

Finally, many parts of the implementation of :class:`HistGradientBoostingClassifier` and :class:`HistGradientBoostingRegressor` are parallelized.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-`

References

[F1999] Friedmann, Jerome H., 2007, "Stochastic Gradient Boosting"

[R2007] G. Ridgeway, "Generalized Boosted Models: A guide to the gbm package", 2007

[XGBoost] Tianqi Chen, Carlos Guestrin, :arxiv:`"XGBoost: A Scalable Tree Boosting System" <1603.02754>`

[LightGBM] (*1*,*2*) Ke et. al. "LightGBM: A Highly Efficient Gradient BoostingDecision Tree"

[Fisher1958] Walter D. Fisher. "On Grouping for Maximum Homogeneity"

Voting Classifier

The idea behind the :class:`VotingClassifier` is to combine conceptually different machine learning classifiers and use a majority vote or the average predicted probabilities (soft vote) to predict the class labels. Such a classifier can be useful for a set of equally well performing model in order to balance out their individual weaknesses.

Majority Class Labels (Majority/Hard Voting)

In majority voting, the predicted class label for a particular sample is the class label that represents the majority (mode) of the class labels predicted by each individual classifier.

E.g., if the prediction for a given sample is

- classifier 1 -> class 1
- classifier 2 -> class 1
- classifier 3 -> class 2

the VotingClassifier (with `voting='hard'`) would classify the sample as "class 1" based on the majority class label.

In the cases of a tie, the :class:`VotingClassifier` will select the class based on the ascending sort order. E.g., in the following scenario

- classifier 1 -> class 2
- classifier 2 -> class 1

the class label 1 will be assigned to the sample.

Usage

The following example shows how to fit the majority rule classifier:

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import VotingClassifier

>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, 1:3], iris.target

>>> clf1 = LogisticRegression(random_state=1)
>>> clf2 = RandomForestClassifier(n_estimators=50, random_state=1)
>>> clf3 = GaussianNB()

>>> eclf = VotingClassifier(
...     estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...     voting='hard')

>>> for clf, label in zip([clf1, clf2, clf3, eclf], ['Logistic Regression', 'Random Forest', 'naive Bayes', 'Ens
```

```
...     scores = cross_val_score(clf, X, y, scoring='accuracy', cv=5)
...     print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), label))
Accuracy: 0.95 (+/- 0.04) [Logistic Regression]
Accuracy: 0.94 (+/- 0.04) [Random Forest]
Accuracy: 0.91 (+/- 0.04) [naive Bayes]
Accuracy: 0.95 (+/- 0.04) [Ensemble]
```

Weighted Average Probabilities (Soft Voting)

In contrast to majority voting (hard voting), soft voting returns the class label as argmax of the sum of predicted probabilities.

Specific weights can be assigned to each classifier via the `weights` parameter. When weights are provided, the predicted class probabilities for each classifier are collected, multiplied by the classifier weight, and averaged. The final class label is then derived from the class label with the highest average probability.

To illustrate this with a simple example, let's assume we have 3 classifiers and a 3-class classification problems where we assign equal weights to all classifiers: w1=1, w2=1, w3=1.

The weighted average probabilities for a sample would then be calculated as follows:

classifier	class 1	class 2	class 3
classifier 1	w1 * 0.2	w1 * 0.5	w1 * 0.3
classifier 2	w2 * 0.6	w2 * 0.3	w2 * 0.1
classifier 3	w3 * 0.3	w3 * 0.4	w3 * 0.3
weighted average	0.37	0.4	0.23

Here, the predicted class label is 2, since it has the highest average probability.

The following example illustrates how the decision regions may change when a soft :class:`VotingClassifier` is used based on an linear Support Vector Machine, a Decision Tree, and a K-nearest neighbor classifier:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1324);** *backlink*
>
> Unknown interpreted text role "class".

```
>>> from sklearn import datasets
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.svm import SVC
>>> from itertools import product
>>> from sklearn.ensemble import VotingClassifier

>>> # Loading some example data
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [0, 2]]
>>> y = iris.target

>>> # Training classifiers
>>> clf1 = DecisionTreeClassifier(max_depth=4)
>>> clf2 = KNeighborsClassifier(n_neighbors=7)
>>> clf3 = SVC(kernel='rbf', probability=True)
>>> eclf = VotingClassifier(estimators=[('dt', clf1), ('knn', clf2), ('svc', clf3)],
...                         voting='soft', weights=[2, 1, 2])

>>> clf1 = clf1.fit(X, y)
>>> clf2 = clf2.fit(X, y)
>>> clf3 = clf3.fit(X, y)
>>> eclf = eclf.fit(X, y)
```

Using the *VotingClassifier* with *GridSearchCV*

The :class:`VotingClassifier` can also be used together with :class:`~sklearn.model_selection.GridSearchCV` in order to tune the hyperparameters of the individual estimators:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1360);** *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, **line 1360);** *backlink*
>
> Unknown interpreted text role "class".

```
>>> from sklearn.model_selection import GridSearchCV
>>> clf1 = LogisticRegression(random_state=1)
>>> clf2 = RandomForestClassifier(random_state=1)
>>> clf3 = GaussianNB()
>>> eclf = VotingClassifier(
...     estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...     voting='soft'
... )

>>> params = {'lr__C': [1.0, 100.0], 'rf__n_estimators': [20, 200]}
```

```
>>> grid = GridSearchCV(estimator=eclf, param_grid=params, cv=5)
>>> grid = grid.fit(iris.data, iris.target)
```

Usage

In order to predict the class labels based on the predicted class-probabilities (scikit-learn estimators in the VotingClassifier must support `predict_proba` method):

```
>>> eclf = VotingClassifier(
...     estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...     voting='soft'
... )
```

Optionally, weights can be provided for the individual classifiers:

```
>>> eclf = VotingClassifier(
...     estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
...     voting='soft', weights=[2,5,1]
... )
```

Voting Regressor

The idea behind the :class:`VotingRegressor` is to combine conceptually different machine learning regressors and return the average predicted values. Such a regressor can be useful for a set of equally well performing models in order to balance out their individual weaknesses.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 1402); *backlink*
>
> Unknown interpreted text role "class".

Usage

The following example shows how to fit the VotingRegressor:

```
>>> from sklearn.datasets import load_diabetes
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.ensemble import VotingRegressor

>>> # Loading some example data
>>> X, y = load_diabetes(return_X_y=True)

>>> # Training classifiers
>>> reg1 = GradientBoostingRegressor(random_state=1)
>>> reg2 = RandomForestRegressor(random_state=1)
>>> reg3 = LinearRegression()
>>> ereg = VotingRegressor(estimators=[('gb', reg1), ('rf', reg2), ('lr', reg3)])
>>> ereg = ereg.fit(X, y)
```

Examples:

- :ref:`sphx_glr_auto_examples_ensemble_plot_voting_regressor.py`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 1435); *backlink*
>
> Unknown interpreted text role "ref".

Stacked generalization

Stacked generalization is a method for combining estimators to reduce their biases [W1992] [HTF]. More precisely, the predictions of each individual estimator are stacked together and used as input to a final estimator to compute the prediction. This final estimator is trained through cross-validation.

The :class:`StackingClassifier` and :class:`StackingRegressor` provide such strategies which can be applied to classification and regression problems.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 1448); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 1448); *backlink*
>
> Unknown interpreted text role "class".

The *estimators* parameter corresponds to the list of the estimators which are stacked together in parallel on the input data. It should be given as a list of names and estimators:

```
>>> from sklearn.linear_model import RidgeCV, LassoCV
>>> from sklearn.neighbors import KNeighborsRegressor
>>> estimators = [('ridge', RidgeCV()),
...               ('lasso', LassoCV(random_state=42)),
...               ('knr', KNeighborsRegressor(n_neighbors=20,
...                                           metric='euclidean'))]
```

The *final_estimator* will use the predictions of the *estimators* as input. It needs to be a classifier or a regressor when using :class:`StackingClassifier` or :class:`StackingRegressor`, respectively:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst,` line 1462); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst,` line 1462); *backlink*
>
> Unknown interpreted text role "class".

```
>>> from sklearn.ensemble import GradientBoostingRegressor
>>> from sklearn.ensemble import StackingRegressor
>>> final_estimator = GradientBoostingRegressor(
...     n_estimators=25, subsample=0.5, min_samples_leaf=25, max_features=1,
...     random_state=42)
>>> reg = StackingRegressor(
...     estimators=estimators,
...     final_estimator=final_estimator)
```

To train the *estimators* and *final_estimator*, the *fit* method needs to be called on the training data:

```
>>> from sklearn.datasets import load_diabetes
>>> X, y = load_diabetes(return_X_y=True)
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
...                                                     random_state=42)
>>> reg.fit(X_train, y_train)
StackingRegressor(...)
```

During training, the *estimators* are fitted on the whole training data *X_train*. They will be used when calling *predict* or *predict_proba*. To generalize and avoid over-fitting, the *final_estimator* is trained on out-samples using :func:`sklearn.model_selection.cross_val_predict` internally.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst,` line 1486); *backlink*
>
> Unknown interpreted text role "func".

For :class:`StackingClassifier`, note that the output of the `estimators` is controlled by the parameter *stack_method* and it is called by each estimator. This parameter is either a string, being estimator method names, or *'auto'* which will automatically identify an available method depending on the availability, tested in the order of preference: *predict_proba*, *decision_function* and *predict*.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst,` line 1491); *backlink*
>
> Unknown interpreted text role "class".

A :class:`StackingRegressor` and :class:`StackingClassifier` can be used as any other regressor or classifier, exposing a *predict*, *predict_proba*, and *decision_function* methods, e.g.:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst,` line 1498); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst,` line 1498); *backlink*
>
> Unknown interpreted text role "class".

```
>>> y_pred = reg.predict(X_test)
>>> from sklearn.metrics import r2_score
>>> print('R2 score: {:.2f}'.format(r2_score(y_test, y_pred)))
R2 score: 0.53
```

Note that it is also possible to get the output of the stacked *estimators* using the *transform* method:

```
>>> reg.transform(X_test[:5])
array([[142..., 138..., 146...],
       [179..., 182..., 151...],
       [139..., 132..., 158...],
       [286..., 292..., 225...],
       [126..., 124..., 164...]])
```

In practice, a stacking predictor predicts as good as the best predictor of the base layer and even sometimes outperforms it by combining the different strengths of the these predictors. However, training a stacking predictor is computationally expensive.

> **Note**
>
> For :class:`StackingClassifier`, when using *stack_method_='predict_proba'*, the first column is dropped when the problem is a binary classification problem. Indeed, both probability columns predicted by each estimator are perfectly collinear.
>
> > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 1523); *backlink*
> >
> > Unknown interpreted text role "class".

> **Note**
>
> Multiple stacking layers can be achieved by assigning *final_estimator* to a :class:`StackingClassifier` or :class:`StackingRegressor`:
>
> > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 1529); *backlink*
> >
> > Unknown interpreted text role "class".
>
> > **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 1529); *backlink*
> >
> > Unknown interpreted text role "class".
>
> ```
> >>> final_layer_rfr = RandomForestRegressor(
> ...     n_estimators=10, max_features=1, max_leaf_nodes=5,random_state=42)
> >>> final_layer_gbr = GradientBoostingRegressor(
> ...     n_estimators=10, max_features=1, max_leaf_nodes=5,random_state=42)
> >>> final_layer = StackingRegressor(
> ...     estimators=[('rf', final_layer_rfr),
> ...                 ('gbrt', final_layer_gbr)],
> ...     final_estimator=RidgeCV()
> ...     )
> >>> multi_layer_regressor = StackingRegressor(
> ...     estimators=[('ridge', RidgeCV()),
> ...                 ('lasso', LassoCV(random_state=42)),
> ...                 ('knr', KNeighborsRegressor(n_neighbors=20,
> ...                                             metric='euclidean'))],
> ...     final_estimator=final_layer
> ... )
> >>> multi_layer_regressor.fit(X_train, y_train)
> StackingRegressor(...)
> >>> print('R2 score: {:.2f}'
> ...       .format(multi_layer_regressor.score(X_test, y_test)))
> R2 score: 0.53
> ```

References

[W1992] Wolpert, David H. "Stacked generalization." Neural networks 5.2 (1992): 241-259.

Docutils System Messages

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 622); *backlink*
>
> Unknown target name: "f2001".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 756); *backlink*
>
> Unknown target name: "f2001".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\modules\(scikit-learn-main)(doc)(modules)ensemble.rst`, line 788); *backlink*
>
> Unknown target name: "f2001".