10
Decision Trees and Random Forests

In this chapter, we will learn about two new classes of machine learning models: decision trees and random forests. We will see how decision trees learn rules from data that encodes non-linear relationships between the input and the output variables. We will illustrate how to train a decision tree and use it for prediction for regression and classification problems, visualize and interpret the rules learned by the model, and tune the model's hyperparameters to optimize the bias-variance tradeoff and prevent overfitting. Decision trees are not only important standalone models but are also frequently used as components in other models.

In the second part of this chapter, we will introduce ensemble models that combine multiple individual models to produce a single aggregate prediction with lower prediction-error variance. We will illustrate bootstrap aggregation, often called bagging, as one of several methods to randomize the construction of individual models and reduce the correlation of the prediction errors made by an ensemble's components.

Boosting is a very powerful alternative method that merits its own chapter to address a range of recent developments. We will illustrate how bagging effectively reduces the variance, and learn how to configure, train, and tune random forests. We will see how random forests as an ensemble of a large number of decision trees, can dramatically reduce prediction errors, at the expense of some loss in interpretation.

In short, in this chapter, we will cover the following:

- How to use decision trees for regression and classification
- How to gain insights from decision trees and visualize the decision rules learned from the data
- Why ensemble models tend to deliver superior results
- How bootstrap aggregation addresses the overfitting challenges of decision trees
- How to train, tune, and interpret random forests

Decision trees

Decision trees are a machine learning algorithm that predicts the value of a target variable based on decision rules learned from training data. The algorithm can be applied to both regression and classification problems by changing the objective function that governs how the tree learns the decision rules.

We will discuss how decision trees use rules to make predictions, how to train them to predict (continuous) returns as well as (categorical) directions of price movements, and how to interpret, visualize, and tune them effectively.

How trees learn and apply decision rules

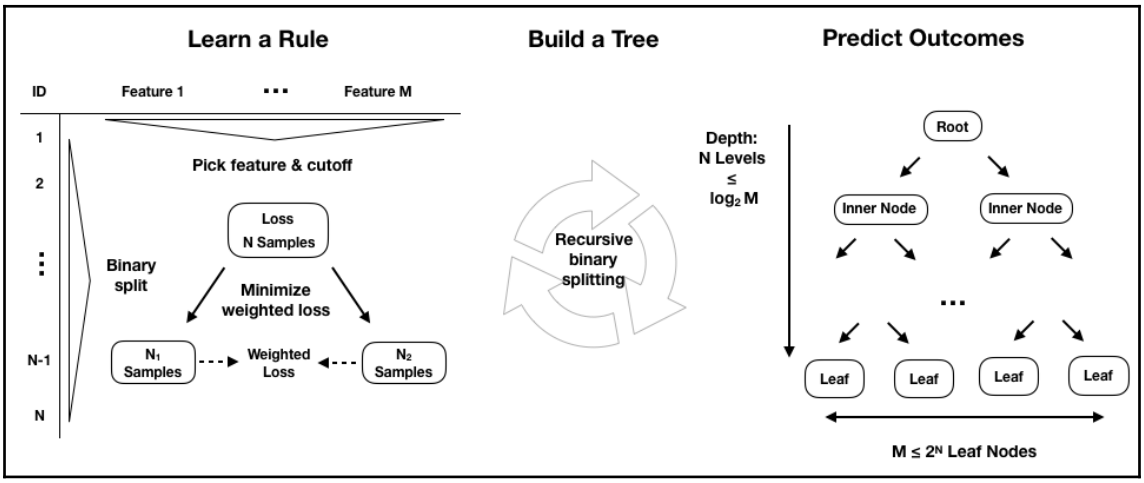
The linear models we studied in `Chapters 7`, *Linear Models* and `Chapter 8`, *Time Series Models*, learn a set of parameters to predict the outcome using a linear combination of the input variables, possibly after transformation by an S-shaped link function in the case of logistic regression.

Decision trees take a different approach: they learn and sequentially apply a set of rules that split data points into subsets and then make one prediction for each subset. The predictions are based on the outcome values for the subset of training samples that result from the application of a given sequence of rules. As we will see in more detail further, classification trees predict a probability estimated from the relative class frequencies or the value of the majority class directly, whereas regression models compute prediction from the mean of the outcome values for the available data points.

Each of these rules relies on one particular feature and uses a threshold to split the samples into two groups with values either below or above the threshold with respect to this feature. A binary tree naturally represents the logic of the model: the root is the starting point for all samples, nodes represent the application of the decision rules, and the data moves along the edges as it is split into smaller subsets until arriving at a leaf node where the model makes a prediction.

For a linear model, the parameter values allow for an interpretation of the impact of the input variables on the output and the model's prediction. In contrast, for a decision tree, the path from the root to the leaves creates transparency about how the features and their values lead to specific decisions by the model.

The following figure highlights how the model learns a rule. During training, the algorithm scans the features and, for each feature, seeks to find a cutoff that splits the data to minimize the loss that results from predictions made using the subsets that would result from the split, weighted by the number of samples in each subset:



To build an entire tree during training, the learning algorithm repeats this process of dividing the feature space, that is, the set of possible values for the p input variables, X_1, X_2, ..., X_p, into mutually-exclusive and collectively-exhaustive regions, each represented by a leaf node. Unfortunately, the algorithm will not be able to evaluate every possible partition of the feature space given the explosive number of possible combinations of sequences of features and thresholds. Tree-based learning takes a top-down, greedy approach, known as recursive binary splitting to overcome this computational limitation.

This process is recursive because it uses subsets of data resulting from prior splits. It is top-down because it begins at the root node of the tree, where all observations still belong to a single region and then successively creates two new branches of the tree by adding one more split to the predictor space. It is greedy because the algorithm picks the best rule in the form of a feature-threshold combination based on the immediate impact on the objective function rather than looking ahead and evaluating the loss several steps ahead. We will return to the splitting logic in the more specific context of regression and classification trees because this represents the major difference.

The number of training samples continues to shrink as recursive splits add new nodes to the tree. If rules split the samples evenly, resulting in a perfectly balanced tree with an equal number of children for every node, then there would be 2^n nodes at level n, each containing a corresponding fraction of the total number of observations. In practice, this is unlikely, so the number of samples along some branches may diminish rapidly, and trees tend to grow to different levels of depth along different paths.

To arrive at a prediction for a new observation, the model uses the rules that it inferred during training to decide which leaf node the data point should be assigned to, and then uses the mean (for regression) or the mode (for classification) of the training observations in the corresponding region of the feature space. A smaller number of training samples in a given region of the feature space, that is, in a given leaf node, reduces the confidence in the prediction and may reflect overfitting.

Recursive splitting would continue until each leaf node contains only a single sample and the training error has been reduced to zero. We will introduce several criteria to limit splits and prevent this natural tendency of decision trees to produce extreme overfitting.

How to use decision trees in practice

In this section, we illustrate how to use tree-based models to gain insight and make predictions. To demonstrate regression trees we predict returns, and for the classification case, we return to the example of positive and negative asset price moves. The code examples for this section are in the notebook `decision_trees` unless stated otherwise.

How to prepare the data

We use a simplified version of the data set constructed in `Chapter 4`, *Alpha Factor Research*. It consists of daily stock prices provided by Quandl for the 2010-2017 period and various engineered features. The details can be found in the `data_prep` notebook in the GitHub repo for this chapter. The decision tree models in this chapter are not equipped to handle missing or categorical variables, so we will apply dummy encoding to the latter after dropping any of the former.

How to code a custom cross-validation class

We also construct a custom cross-validation class tailored to the format of the data just created, which has pandas MultiIndex with two levels, one for the ticker and one for the data:

```
class OneStepTimeSeriesSplit:
    """Generates tuples of train_idx, test_idx pairs
    Assumes the index contains a level labeled 'date'"""

    def __init__(self, n_splits=3, test_period_length=1, shuffle=False):
        self.n_splits = n_splits
        self.test_period_length = test_period_length
        self.shuffle = shuffle
        self.test_end = n_splits * test_period_length

    @staticmethod
    def chunks(l, chunk_size):
        for i in range(0, len(l), chunk_size):
            yield l[i:i + chunk_size]

    def split(self, X, y=None, groups=None):
        unique_dates = (X.index
                        .get_level_values('date')
                        .unique()
                        .sort_values(ascending=False)[:self.test_end])

        dates = X.reset_index()[['date']]
        for test_date in self.chunks(unique_dates,
self.test_period_length):
            train_idx = dates[dates.date < min(test_date)].index
            test_idx = dates[dates.date.isin(test_date)].index
            if self.shuffle:
                np.random.shuffle(list(train_idx))
            yield train_idx, test_idx
```

`OneStepTimeSeriesSplit` ensures a split of training and validation sets that avoids a lookahead bias by training models using only data up to period *T-1* for each stock when validating using data for month *T*. We will only use one-step-ahead forecasts.

How to build a regression tree

Regression trees make predictions based on the mean outcome value for the training samples assigned to a given node and typically rely on the mean-squared error to select optimal rules during recursive binary splitting.

Given a training set, the algorithm iterates over the predictors, X_1, X_2, ..., X_p, and possible cutpoints, s_1, s_1, ..., s_N, to find an optimal combination. The optimal rule splits the feature space into two regions, $\{X \mid X_i < s_j\}$ and $\{X \mid X_i > s_j\}$, with values for the X_i feature either below or above the s_j threshold so that predictions based on the training subsets maximize the reduction of the squared residuals relative to the current node.

Let's start with a simplified example to facilitate visualization and only use two months of lagged returns to predict the following month, in the vein of an AR(2) model from the last chapter:

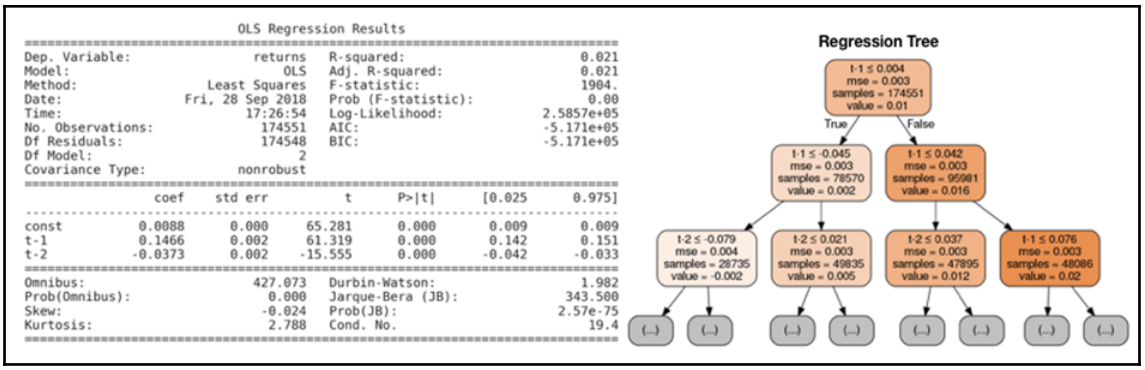
$$r_t = f(r_{t-1}, r_{t-2})$$

Using `sklearn`, configuring and training a regression tree is very straightforward:

```
from sklearn.tree import DecisionTreeRegressor

# configure regression tree
regression_tree = DecisionTreeRegressor(criterion='mse', # default
                                        max_depth=4,     # up to 4 splits
                                        random_state=42)
# Create training data
y = data.returns
X = data.drop('returns', axis=1)
X2 = X.loc[:, ['t-1', 't-2']]

# fit model
regression_tree.fit(X=X2, y=y)

# fit OLS model
ols_model = sm.OLS(endog=y, exog=sm.add_constant(X2)).fit()
```

The OLS summary and a visualization of the first two levels of the decision tree reveal the striking differences between the model. The OLS model provides three parameters for the intercepts and the two features in line with the linear assumption this model makes about the *f* function.

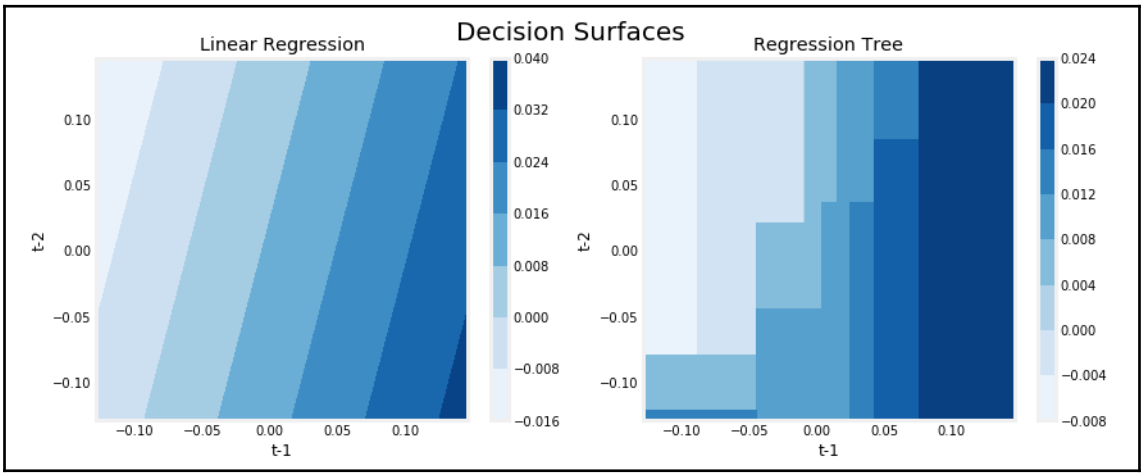
In contrast, the regression tree chart displays, for each node of the first two levels, the feature and threshold used to split the data (note that features can be used repeatedly), as well as the current value of the **mean-squared error** (**MSE**), the number of samples, and predicted value based on these training samples:



The regression tree chart

The tree chart also highlights the uneven distribution of samples across the nodes as the numbers vary between 28,000 and 49,000 samples after only two splits.

To further illustrate the different assumptions about the functional form of the relationships between the input variables and the output, we can visualize current return predictions as a function of the feature space, that is, as a function of the range of values for the lagged returns. The following figure shows the current period return as a function of returns one and two periods ago for linear regression and the regression tree:



The linear-regression model result on the right side underlines the linearity of the relationship between lagged and current returns, whereas the regression tree chart on the left illustrates the non-linear relationship encoded in the recursive partitioning of the feature space.

How to build a classification tree

A classification tree works just like the regression version, except that categorical nature of the outcome requires a different approach to making predictions and measuring the loss. While a regression tree predicts the response for an observation assigned to a leaf node using the mean outcome of the associated training samples, a classification tree instead uses the mode, that is, the most common class among the training samples in the relevant region. A classification tree can also generate probabilistic predictions based on relative class frequencies.

How to optimize for node purity

When growing a classification tree, we also use recursive binary splitting but, instead of evaluating the quality of a decision rule using the reduction of the mean-squared error, we can use the classification error rate, which is simply the fraction of the training samples in a given (leave) node that do not belong to the most common class.

However, the alternative measures, Gini Index or Cross-Entropy, are preferred because they are more sensitive to node purity than the classification error rate. Node purity refers to the extent of the preponderance of a single class in a node. A node that only contains samples with outcomes belonging to a single class is pure and imply successful classification for this particular region of the feature space. They are calculated as follows for a classification outcome taking on *K* values, *0,1,...,K-1*, for a given node, *m*, that represents a region, $R_{m,}$ of the feature space and where p_{mk} is the proportion of outcomes of the *k* class in the *m* node:

$$\text{Gini Impurity} = \sum_{k} p_{mk}(1 - p_{mk})$$

$$\text{Cross-Entropy} = -\sum_{k} p_{mk} log(p_{mk})$$

Both the Gini Impurity and the Cross-Entropy measure take on smaller values when the class proportions approach zero or one, that is, when the child nodes become pure as a result of the split and are highest when the class proportions are even or 0.5 in the binary case. The chart at the end of this section visualizes the values assumed by these two measures and the misclassification error rates across the [0, 1] interval of proportions.

How to train a classification tree

We will now train, visualize, and evaluate a classification tree with up to 5 consecutive splits using 80% of the samples for training to predict the remaining 20%. We are taking a shortcut here to simplify the illustration and use the built-in `train_test_split`, which does not protect against lookahead bias, as our custom iterator. The tree configuration implies up to 2^5=32 leaf nodes that, on average in the balanced case, would contain over 4,300 of the training samples. Take a look at the following code:

```
# randomize train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y_binary,
test_size=0.2, random_state=42)

# configure & train tree learner
classifier = DecisionTreeClassifier(criterion='gini',
                                    max_depth=5,
                                    random_state=42)
classifier.fit(X=X_train, y=y_train)

# Output:
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=42,
            splitter='best')
```

The output after training the model displays all the `DecisionTreeClassifier` parameters that we will address in more detail in the next section when we discuss parameter-tuning.

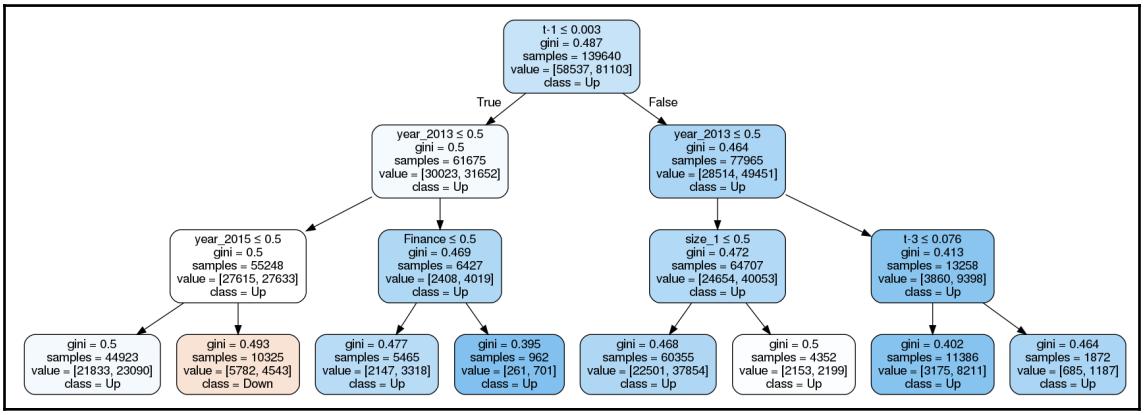
How to visualize a decision tree

You can visualize the tree using the `graphviz` library (see GitHub for installation instructions) because `sklearn` can output a description of the tree using the `.dot` language used by that library. You can configure the output to include feature and class labels and limit the number of levels to keep the chart readable, as follows:

```
dot_data = export_graphviz(classifier,
                           out_file=None, # opt. save to file and convert
to png
                           feature_names=X.columns,
                           class_names=['Down', 'Up'],
                           max_depth=3,
                           filled=True,
```

```
                                rounded=True,
                                special_characters=True)

graphviz.Source(dot_data)
```

The result shows that the model uses a variety of different features and indicates the split rules for both continuous and categorical (dummy) variables. The chart displays, under the label **value**, the number of samples from each class and, under the label **class**, the most common class (there were more up months during the sample period):



How to evaluate decision tree predictions

To evaluate the predictive accuracy of our first classification tree, we will use our test set to generate predicted class probabilities, as follows:

```
y_score = classifier.predict_proba(X=X_test)[:, 1] # only keep
probabilities for pos. class
```

The `.predict_proba()` method produces one probability for each class. In the binary class, these probabilities are complementary and sum to 1, so we only need the value for the positive class. To evaluate the generalization error, we will use the area under the curve based on the receiver-operating characteristic that we introduced in `Chapter 6`, *The Machine Learning Process*. The result indicates a significant improvement above and beyond the baseline value of 0.5 for a random prediction:

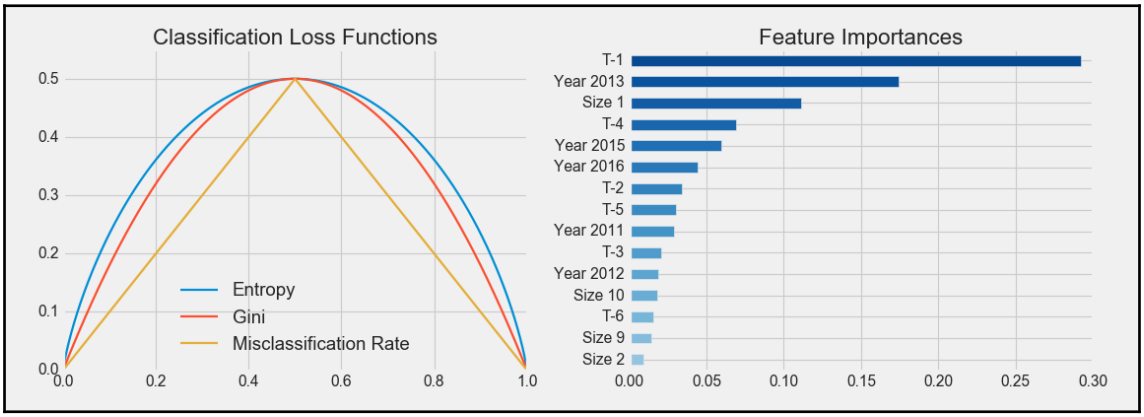
```
roc_auc_score(y_score=y_score, y_true=y_test)
0.5941
```

Feature importance

Decision trees can not only be visualized to inspect the decision path for a given feature, but also provide a summary measure of the contribution of each feature to the model fit to the training data.

The feature importance captures how much the splits produced by the feature helped to optimize the model's metric used to evaluate the split quality, which in our case is the Gini Impurity index. A feature's importance is computed as the (normalized) total reduction of this metric and takes into account the number of samples affected by a split. Hence, features used earlier in the tree where the nodes tend to contain more samples typically are considered of higher importance.

The following chart shows the feature importance for the top 15 features:



Overfitting and regularization

Decision trees have a strong tendency to overfit, especially when a dataset has a large number of features relative to the number of samples. As discussed in previous chapters, overfitting increases the prediction error because the model does not only learn the signal contained in the training data, but also the noise.

There are several ways to address the risk of overfitting:

- **Dimensionality reduction** (`Chapter 12`, *Unsupervised Learning*) improves the feature-to-sample ratio by representing the existing features with fewer, more informative, and less noisy features.

- **Ensemble models**, such as random forests, combine multiple trees while randomizing the tree construction, as we will see in the second part of this chapter.
- Decision trees provide several **regularization** hyperparameters to limit the growth of a tree and the associated complexity. While every split increases the number of nodes, it also reduces the number of samples available per node to support a prediction. For each additional level, twice the number of samples is needed to populate the new nodes with the same sample density.
- **Tree-pruning** is an additional tool to reduce the complexity of a tree by eliminating nodes or entire parts of a tree that add little value but increase the model's variance. Cost-complexity-pruning, for instance, starts with a large tree and recursively reduces its size by replacing nodes with leaves, essentially running the tree construction in reverse. The various steps produce a sequence of trees that can then be compared using cross-validation to select the ideal size.

How to regularize a decision tree

The following table lists key parameters available for this purpose in the sklearn decision tree implementation. After introducing the most important parameters, we will illustrate how to use cross-validation to optimize the hyperparameter settings with respect to the bias-variance tradeoff and lower prediction errors:

Parameter	Default	Options	Description
`max_depth`	None	int	Maximum number of levels: split nodes until reaching `max_depth` or all leaves are pure or contain fewer than `min_samples_split` samples.
`max_features`	None	None: all features; int float: fraction auto, sqrt: sqrt(n_features) log2: log2(n_features)	Number of features to consider for a split.
`max_leaf_nodes`	None	None: unlimited number of leaf nodes int	Split nodes until creating this many leaves.
`min_impurity_decrease`	0	float	Split node if impurity decreases by at least this value.
`min_samples_leaf`	1	int; float (as a percentage of N)	Minimum number of samples to be at a leaf node. A split will only be considered if there are at least `min_samples_leaf` training samples in each of the left and right branches. May smoothen the model, especially for regression.
`min_samples_split`	2	int; float (percent of N)	The minimum number of samples required to split an internal node:
`min_weight_fraction_leaf`	0		The minimum weighted fraction of the sum total of all sample weights needed at a leaf node. Samples have equal weight unless `sample_weight` provided in fit method.

The `max_depth` parameter imposes a hard limit on the number of consecutive splits and represents the most straightforward way to cap the growth of a tree.

The `min_samples_split` and `min_samples_leaf` parameters are alternative, data-driven ways to limit the growth of a tree. Rather than imposing a hard limit on the number of consecutive splits, these parameters control the minimum number of samples required to further split the data. The latter guarantees a certain number of samples per leaf, while the former can create very small leaves if a split results in a very uneven distribution. Small parameter values facilitate overfitting, while a high number may prevent the tree from learning the signal in the data. The default values are often quite low, and you should use cross-validation to explore a range of potential values. You can also use a float to indicate a percentage as opposed to an absolute number.

The sklearn documentation contains additional details about how to use the various parameters for different use cases; see GitHub references.

Decision tree pruning

Recursive binary-splitting will likely produce good predictions on the training set but tends to overfit the data and produce poor generalization performance because it leads to overly complex trees, reflected in a large number of leaf nodes or partitioning of the feature space. Fewer splits and leaf nodes imply an overall smaller tree and often lead to better predictive performance as well as interpretability.

One approach to limit the number of leaf nodes is to avoid further splits unless they yield significant improvements of the objective metric. The downside of this strategy, however, is that sometimes splits that result in small improvements enable more valuable splits later on as the composition of the samples keeps changing.

Tree-pruning, in contrast, starts by growing a very large tree before removing or pruning nodes to reduce the large tree to a less complex and overfit subtree. Cost-complexity-pruning generates a sequence of subtrees by adding a penalty for adding leaf nodes to the tree model and a regularization parameter, similar to the lasso and ridge linear-regression models, that modulates the impact of the penalty. Applied to the large tree, an increasing penalty will automatically produce a sequence of subtrees. Cross-validation of the regularization parameter can be used to identify the optimal, pruned subtree.

This method is not yet available in sklearn; see references on GitHub for further details and ways to manually implement pruning.

How to tune the hyperparameters

Decision trees offer an array of hyperparameters to control and tune the training result. Cross-validation is the most important tool to obtain an unbiased estimate of the generalization error, which in turn permits an informed choice among the various configuration options. sklearn offers several tools to facilitate the process of cross-validating numerous parameter settings, namely the `GridSearchCV` convenience class that we will illustrate in the next section. Learning curves also allow for diagnostics that evaluate potential benefits of collecting additional data to reduce the generalization error.

GridsearchCV for decision trees

`sklearn` provides a method to define ranges of values for multiple hyperparameters. It automates the process of cross-validating the various combinations of these parameter values to identify the optimal configuration. Let's walk through the process of automatically tuning your model.

The first step is to instantiate a model object and define a dictionary where the keywords name the hyperparameters, and the values list the parameter settings to be tested:

```
clf = DecisionTreeClassifier(random_state=42)
param_grid = {'max_depth': range(10, 20),
              'min_samples_leaf': [250, 500, 750],
              'max_features': ['sqrt', 'auto']
              }
```

Then, instantiate the `GridSearchCV` object, providing the estimator object and parameter grid, as well as a scoring method and cross-validation choice to the initialization method. We'll use an object of our custom `OneStepTimeSeriesSplit` class, initialized to use ten folds for the `cv` parameter, and set the scoring to the `roc_auc` metric. We can parallelize the search using the `n_jobs` parameter and automatically obtain a trained model that uses the optimal hyperparameters by setting `refit=True`.

With all settings in place, we can fit `GridSearchCV` just like any other model:

```
gridsearch_clf = GridSearchCV(estimator=clf,
                              param_grid=param_grid,
                              scoring='roc_auc',
                              n_jobs=-1,
                              cv=cv,  # custom OneStepTimeSeriesSplit
                              refit=True,
                              return_train_score=True)

gridsearch_clf.fit(X=X, y=y_binary)
```

The training process produces some new attributes for our `GridSearchCV` object, most importantly the information about the optimal settings and the best cross-validation score (now using the proper setup that avoids lookahead bias).

Setting `max_depth` to `13`, `min_samples_leaf` to `500`, and randomly selecting only a number corresponding to the square root of the total number of features when deciding on a split, produces the best results, with an AUC of `0.5855`:

```
gridsearch_clf.best_params_
{'max_depth': 13, 'max_features': 'sqrt', 'min_samples_leaf': 500}

gridsearch_clf.best_score_
0.5855
```

The automation is quite convenient, but we also would like to inspect how the performance evolves for different parameter values. Upon completion of this process, the `GridSearchCV` object makes available detailed cross-validation results to gain more insights.

How to inspect the tree structure

The notebook also illustrates how to run cross-validation more manually to obtain custom tree attributes, such as the total number of nodes or leaf nodes associated with certain hyperparameter settings. The following function accesses the internal `.tree_` attribute to retrieve information about the total node count, and how many of these nodes are leaf nodes:

```
def get_leaves_count(tree):
    t = tree.tree_
    n = t.node_count
    leaves = len([i for i in range(t.node_count) if t.children_left[i]==
-1])
    return leaves
```

We can combine this information with the train and test scores to gain detailed knowledge about the model behavior throughout the cross-validation process, as follows:

```
train_scores, val_scores, leaves = {}, {}, {}
for max_depth in range(1, 26):
    print(max_depth, end=' ', flush=True)
    clf = DecisionTreeClassifier(criterion='gini',
                                 max_depth=max_depth,
                                 min_samples_leaf=500,
                                 max_features='auto',
                                 random_state=42)
```

```
    train_scores[max_depth], val_scores[max_depth], leaves[max_depth] = [],
[], []
    for train_idx, test_idx in cv.split(X):
        X_train, y_train,  = X.iloc[train_idx], y_binary.iloc[train_idx]
        X_test, y_test = X.iloc[test_idx], y_binary.iloc[test_idx]
        clf.fit(X=X_train, y=y_train)

        train_pred = clf.predict_proba(X=X_train)[:, 1]
        train_score = roc_auc_score(y_score=train_pred, y_true=y_train)
        train_scores[max_depth].append(train_score)

        test_pred = clf.predict_proba(X=X_test)[:, 1]
        val_score = roc_auc_score(y_score=test_pred, y_true=y_test)
        val_scores[max_depth].append(val_score)
        leaves[max_depth].append(get_leaves_count(clf))
```

The result is shown on the left panel of the following chart. It highlights the in- and out-of-sample performance across the range of `max_depth` settings, alongside a confidence interval around the error metrics. It also shows the number of leaf nodes on the right-hand log scale and indicates the best-performing setting at 13 consecutive splits, as indicated by the vertical black line.

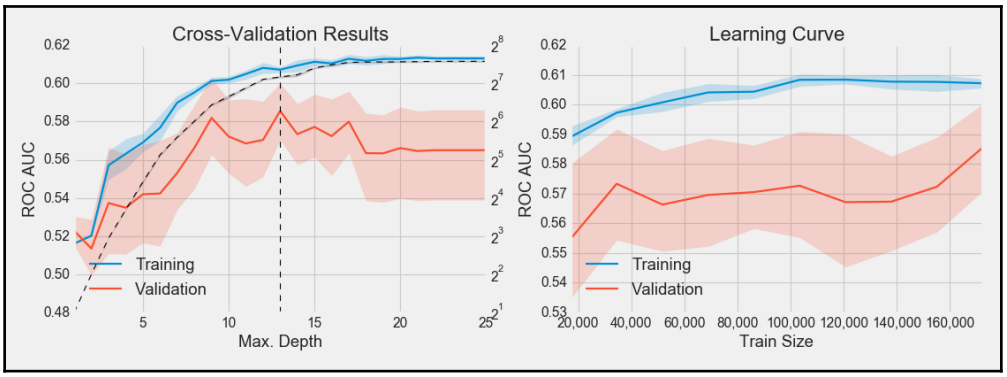
Learning curves

A learning curve is a useful tool that displays how the validation and training score evolve as the number of training samples evolves.

The purpose of the learning curve is to find out whether and how much the model would benefit from using more data during training. It is also useful to diagnose whether the model's generalization error is more likely driven by bias or variance.

If, for example, both the validation score and the training score converge to a similarly low value despite an increasing training set size, the error is more likely due to bias, and additional training data is unlikely to help.

Take a look at the following visualization:



Strengths and weaknesses of decision trees

Regression and classification trees take a very different approach to prediction when compared to the linear models we have explored so far. How do you decide which model is more suitable to the problem at hand? Consider the following:

- If the relationship between the outcome and the features is approximately linear (or can be transformed accordingly), then linear regression will likely outperform a more complex method, such as a decision tree that does not exploit this linear structure.
- If the relationship appears highly non-linear and more complex, decision trees will likely outperform the classical models.

Several advantages have made decision trees very popular:

- They are fairly straightforward to understand and to interpret, not least because they can be easily visualized and are thus more accessible to a non-technical audience. Decision trees are also referred to as white-box models given the high degree of transparency about how they arrive at a prediction. Black-box models, such as ensembles and neural networks may deliver better prediction accuracy but the decision logic is often much more challenging to understand and interpret.
- Decision trees require less data preparation than models that make stronger assumptions about the data or are more sensitive to outliers and require data standardization (such as regularized regression).

- Some decision tree implementations handle categorical input, do not require the creation of dummy variables (improving memory efficiency), and can work with missing values, as we will see in `Chapter 11`, *Gradient Boosting Machines*, but this is not the case for sklearn.
- Prediction is fast because it is logarithmic in the number of leaf nodes (unless the tree becomes extremely unbalanced).
- It is possible to validate the model using statistical tests and account for its reliability (see GitHub references).

Decision trees also have several key disadvantages:

- Decision trees have a built-in tendency to overfit to the training set and produce a high generalization error. Key steps to address this weakness are pruning (not yet supported by sklearn) as well as regularization using the various early-stopping criteria outlined in the previous section.
- Closely related is the high variance of decision trees that results from their ability to closely adapt to a training set so that minor variations in the data can produce wide swings in the structure of the decision trees and, consequently, the predictions the model generates. The key mechanism to address the high variance of decision trees is the use of an ensemble of randomized decision trees that have low bias and produce uncorrelated prediction errors.
- The greedy approach to decision-tree learning optimizes based on local criteria, that is, to reduce the prediction error at the current node and does not guarantee a globally optimal outcome. Again, ensembles consisting of randomized trees help to mitigate this problem.
- Decision trees are also sensitive to unbalanced class weights and may produce biased trees. One option is to oversample the underrepresented or under-sample the more frequent class. It is typically better, though, to use class weights and directly adjust the objective function.

Random forests

Decision trees are not only useful for their transparency and interpretability but are also fundamental building blocks for much more powerful ensemble models that combine many individual trees with strategies to randomly vary their design to address the overfitting and high variance problems discussed in the preceding section.

Ensemble models

Ensemble learning involves combining several machine learning models into a single new model that aims to make better predictions than any individual model. More specifically, an ensemble integrates the predictions of several base estimators trained using one or more given learning algorithms to reduce the generalization error that these models may produce on their own.

For ensemble learning to achieve this goal, the individual models must be:

- **Accurate:** They outperform a naive baseline (such as the sample mean or class proportions)
- **Independent:** Their predictions are generated differently to produce different errors

Ensemble methods are among the most successful machine learning algorithms, in particular for standard numerical data. Large ensembles are very successful in machine learning competitions and may consist of many distinct individual models that have been combined by hand or using another machine learning algorithm.

There are several disadvantages to combining predictions made by different models. These include reduced interpretability, and higher complexity and cost of training, prediction, and model maintenance. As a result, in practice (outside of competitions), the small gains in accuracy from large-scale ensembling may not be worth the added costs.

There are two groups of ensemble methods that are typically distinguished depending on how they optimize the constituent models and then integrate the results for a single ensemble prediction:

- **Averaging methods** train several base estimators independently and then average their predictions. If the base models are not biased and make different prediction errors that are not highly correlated, then the combined prediction may have lower variance and can be more reliable. This resembles the construction of a portfolio from assets with uncorrelated returns to reduce the volatility without sacrificing the return.

- **Boosting methods**, in contrast, train base estimators sequentially with the specific goal to reduce the bias of the combined estimator. The motivation is to combine several weak models into a powerful ensemble.

We will focus on automatic averaging methods in the remainder of this chapter, and boosting methods in `Chapter 11`, *Gradient Boosting Machines*.

How bagging lowers model variance

We saw that decision trees are likely to make poor predictions due to high variance, which implies that the tree structure is quite sensitive to the composition of the training sample. We have also seen that a model with low variance, such as linear regression, produces similar estimates despite different training samples as long as there are sufficient samples given the number of features.

For a given a set of independent observations, each with a variance of σ^2, the standard error of the sample mean is given by σ/n. In other words, averaging over a larger set of observations reduces the variance. A natural way to reduce the variance of a model and its generalization error would thus be to collect many training sets from the population, train a different model on each dataset, and average the resulting predictions.

In practice, we do not typically have the luxury of many different training sets. This is where bagging, short for bootstrap aggregation, comes in. Bagging is a general-purpose method to reduce the variance of a machine learning model, which is particularly useful and popular when applied to decision trees.

Bagging refers to the aggregation of bootstrap samples, which are random samples with replacement. Such a random sample has the same number of observations as the original dataset but may contain duplicates due to replacement.

Bagging increases predictive accuracy but decreases model interpretability because it's no longer possible to visualize the tree to understand the importance of each feature. As an ensemble algorithm, bagging methods train a given number of base estimators on these bootstrapped samples and then aggregate their predictions into a final ensemble prediction.

Bagging reduces the variance of the base estimators by randomizing how, for example, each tree is grown and then averages the predictions to reduce their generalization error. It is often a straightforward approach to improve on a given model without the need to change the underlying algorithm. It works best with complex models that have low bias and high variance, such as deep decision trees, because its goal is to limit overfitting. Boosting methods, in contrast, work best with weak models, such as shallow decision trees.

There are several bagging methods that differ by the random sampling process they apply to the training set:

- Pasting draws random samples from the training data without replacement, whereas bagging samples with replacement
- Random subspaces randomly sample from the features (that is, the columns) without replacement
- Random patches train base estimators by randomly sampling both observations and features

Bagged decision trees

To apply bagging to decision trees, we create bootstrap samples from our training data by repeatedly sampling with replacement, then train one decision tree on each of these samples, and create an ensemble prediction by averaging over the predictions of the different trees.

Bagged decision trees are usually grown large, that is, have many levels and leaf nodes and are not pruned so that each tree has low bias but high variance. The effect of averaging their predictions then aims to reduce their variance. Bagging has been shown to substantially improve predictive performance by constructing ensembles that combine hundreds or even thousands of trees trained on bootstrap samples.

To illustrate the effect of bagging on the variance of a regression tree, we can use the `BaggingRegressor` meta-estimator provided by `sklearn`. It trains a user-defined base estimator based on parameters that specify the sampling strategy:

- `max_samples` and `max_features` control the size of the subsets drawn from the rows and the columns, respectively
- `bootstrap` and `bootstrap_features` determine whether each of these samples is drawn with or without replacement

The following example uses an exponential function to generate training samples for a single `DecisionTreeRegressor` and a `BaggingRegressor` ensemble that consists of ten trees, each grown ten levels deep. Both models are trained on the random samples and predict outcomes for the actual function with added noise.

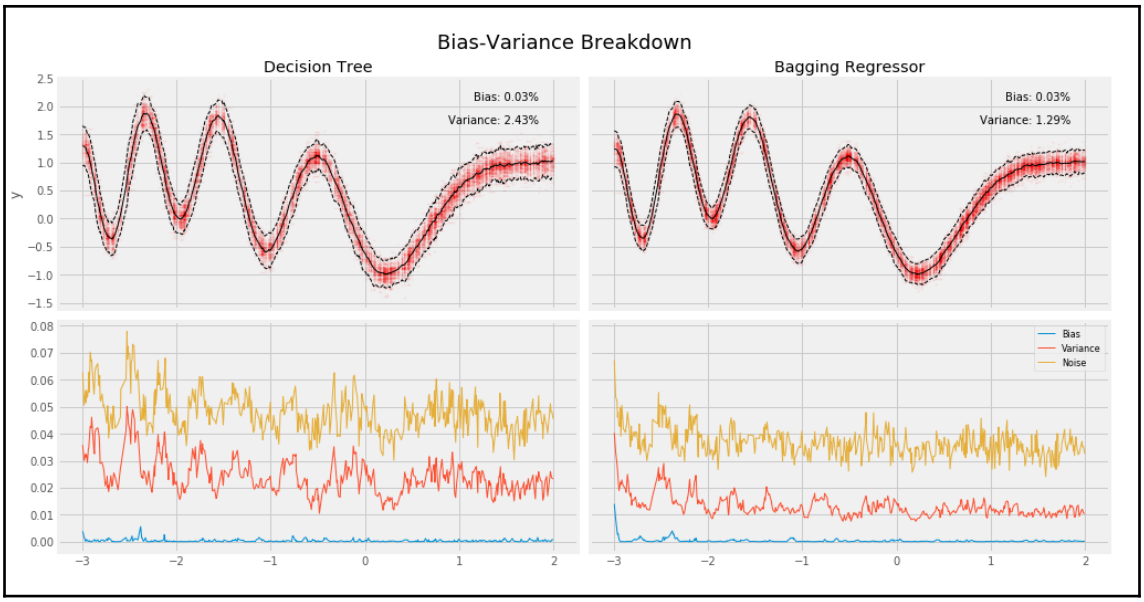
Since we know the true function, we can decompose the mean-squared error into bias, variance, and noise, and compare the relative size of these components for both models according to the following breakdown:

$$E\left[y_0 - \hat{f}\left(x_0\right)\right]^2 = \mathrm{Var}(\hat{f}\left(x_0\right)) + \left[\mathrm{Bias}(\hat{f}\left(x_0\right))\right]^2 + \mathrm{Var}(\epsilon)$$

For 100 repeated random training and test samples of 250 and 500 observations each, we find that the variance of the predictions of the individual decision tree is almost twice as high as that for the small ensemble of `10` bagged trees based on bootstrapped samples:

```
noise = .5  # noise relative to std(y)
noise = y.std() * noise_to_signal

X_test = choice(x, size=test_size, replace=False)

max_depth = 10
n_estimators=10

tree = DecisionTreeRegressor(max_depth=max_depth)
bagged_tree = BaggingRegressor(base_estimator=tree,
n_estimators=n_estimators)
learners = {'Decision Tree': tree, 'Bagging Regressor': bagged_tree}

predictions = {k: pd.DataFrame() for k, v in learners.items()}
for i in range(reps):
    X_train = choice(x, train_size)
    y_train = f(X_train) + normal(scale=noise, size=train_size)
    for label, learner in learners.items():
        learner.fit(X=X_train.reshape(-1, 1), y=y_train)
        preds = pd.DataFrame({i: learner.predict(X_test.reshape(-1, 1))},
index=X_test)
        predictions[label] = pd.concat([predictions[label], preds], axis=1)
```

For each model, the following plot shows the mean prediction and a band of two standard deviations around the mean for both models in the upper panel, and the bias-variance-noise breakdown based on the values for the true function in the bottom panel:



See the notebook `random_forest` for implementation details.

How to build a random forest

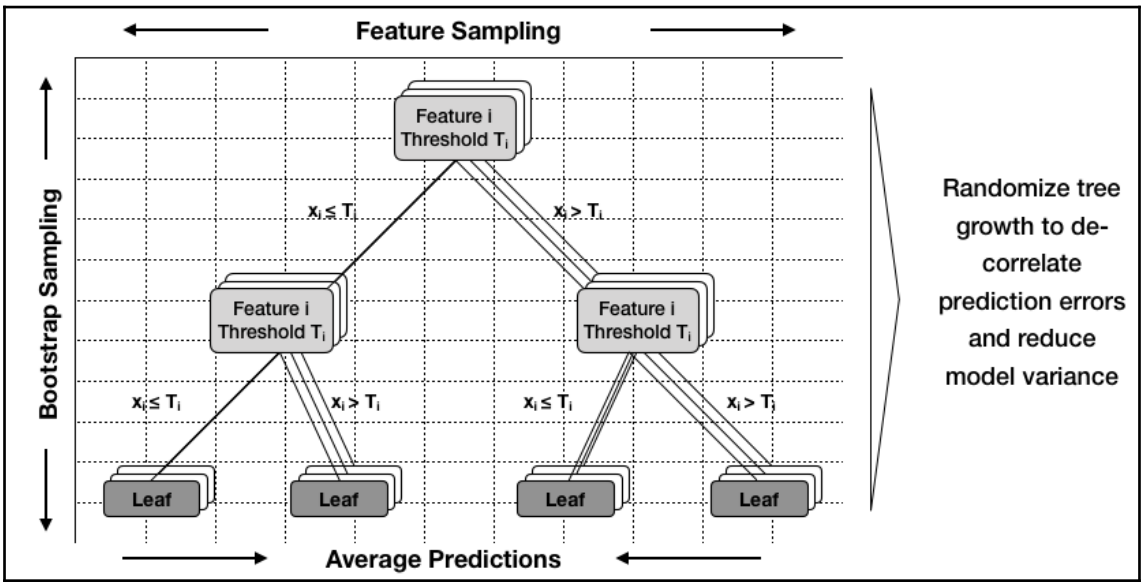
The random forest algorithm expands on the randomization introduced by the bootstrap samples generated by bagging to reduce variance further and improve predictive performance.

In addition to training each ensemble member on bootstrapped training data, random forests also randomly sample from the features used in the model (without replacement). Depending on the implementation, the random samples can be drawn for each tree or each split. As a result, the algorithm faces different options when learning new rules, either at the level of a tree or for each split.

The sizes of the feature samples differ for regression and classification trees:

- For **classification**, the sample size is typically the square root of the number of features.
- For **regression**, it can be anywhere from one-third to all features and should be selected based on cross-validation.

The following diagram illustrates how random forests randomize the training of individual trees and then aggregate their predictions into an ensemble prediction:



The goal of randomizing the features in addition to the training observations is to further de-correlate the prediction errors of the individual trees. All features are not created equal, and a small number of highly relevant features will be selected much more frequently and earlier in the tree-construction process, making decision trees more alike across the ensemble. However, the less the generalization errors of individual trees correlate, the more the overall variance will be reduced.

How to train and tune a random forest

The key configuration parameters include the various hyperparameters for the individual decision trees introduced in the section *How to tune the hyperparameters*. The following tables lists additional options for the two `RandomForest` classes:

Keyword	Default	Description
`bootstrap`	`True`	Bootstrap samples during training.
`n_estimators`	`10`	Number of trees in the forest.
`oob_score`	`False`	Uses out-of-bag samples to estimate the R^2 on unseen data.

The `bootstrap` parameter activates in the preceding bagging algorithm outline, which in turn enables the computation of the out-of-bag score (`oob_score`) that estimates the generalization accuracy using samples not included in the bootstrap sample used to train a given tree (see next section for detail).

The `n_estimators` parameter defines the number of trees to be grown as part of the forest. Larger forests perform better, but also take more time to build. It is important to monitor the cross-validation error as a function of the number of base learners to identify when the marginal reduction of the prediction error declines and the cost of additional training begins to outweigh the benefits.

The `max_features` parameter controls the size of the randomly selected feature subsets available when learning a new decision rule and split a node. A lower value reduces the correlation of the trees and, thus, the ensemble's variance, but may also increase the bias. Good starting values are `n_features` (the number of training features) for regression problems and `sqrt(n_features)` for classification problems, but will depend on the relationships among features and should be optimized using cross-validation.

Random forests are designed to contain deep fully-grown trees, which can be created using `max_depth=None` and `min_samples_split=2`. However, these values are not necessarily optimal, especially for high-dimensional data with many samples and, consequently, potentially very deep trees that can become very computationally-, and memory-, intensive.

The `RandomForest` class provided by `sklearn` support parallel training and prediction by setting the `n_jobs` parameter to the `k` number of jobs to run on different cores. The `-1` value uses all available cores. The overhead of interprocess communication may limit the speedup from being linear so that *k* jobs may take more than *1/k* the time of a single job. Nonetheless, the speedup is often quite significant for large forests or deep individual trees that may take a meaningful amount of time to train when the data is large, and split evaluation becomes costly.

As always, the best parameter configuration should be identified using cross-validation. The following steps illustrate the process:

1. We will use `GridSearchCV` to identify an optimal set of parameters for an ensemble of classification trees:

```
rf_clf = RandomForestClassifier(n_estimators=10,
                                criterion='gini',
                                max_depth=None,
                                min_samples_split=2,
                                min_samples_leaf=1,
```

```
                                        min_weight_fraction_leaf=0.0,
                                        max_features='auto',
                                        max_leaf_nodes=None,
                                        min_impurity_decrease=0.0,
                                        min_impurity_split=None,
                                        bootstrap=True, oob_score=False,
                                        n_jobs=-1, random_state=42)
```

2. We will use 10-fold custom cross-validation and populate the parameter grid with values for the key configuration settings:

```
cv = OneStepTimeSeriesSplit(n_splits=10)
clf = RandomForestClassifier(random_state=42, n_jobs=-1)
param_grid = {'n_estimators': [200, 400],
              'max_depth': [10, 15, 20],
              'min_samples_leaf': [50, 100]}
```

3. Configure `GridSearchCV` using the preceding input:

```
gridsearch_clf = GridSearchCV(estimator=clf,
                              param_grid=param_grid,
                              scoring='roc_auc',
                              n_jobs=-1,
                              cv=cv,
                              refit=True,
                              return_train_score=True,
                              verbose=1)
```

4. Train the multiple ensemble models defined by the parameter grid:

```
gridsearch_clf.fit(X=X, y=y_binary)
```

5. Obtain the best parameters as follows:

```
gridsearch_clf.bestparams
{'max_depth': 15,
 'min_samples_leaf': 100,
 'n_estimators': 400}
```

6. The best score is a small but significant improvement over the single-tree baseline:

```
gridsearch_clf.bestscore_
0.6013
```

Feature importance for random forests

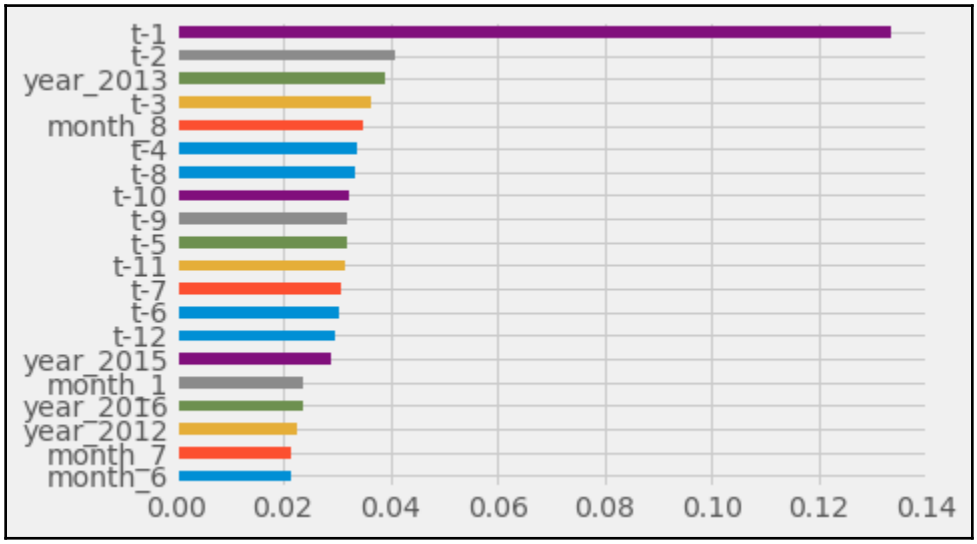
A random forest ensemble may contain hundreds of individual trees, but it is still possible to obtain an overall summary measure of feature importance from bagged models.

For a given feature, the importance score is the total reduction in the objective function's value, which results from splits based on this feature, averaged over all trees. Since the objective function takes into account how many features are affected by a split, this measure is implicitly a weighted average so that features used near the top of a tree will get higher scores due to the larger number of observations contained in the much smaller number of available nodes. By averaging over many trees grown in a randomized fashion, the feature importance estimate loses some variance and becomes more accurate.

The computation differs for classification and regression trees based on the different objectives used to learn the decision rules and is measured in terms of the mean square error for regression trees and the Gini index or entropy for classification trees.

`sklearn` further normalizes the feature-importance measure so that it sums up to `1`. Feature importance thus computed is also used for feature selection as an alternative to the mutual information measures we saw in `Chapter 6`, *The Machine Learning Process* (see `SelectFromModel` in the `sklearn.feature_selection` module).

In our example, the importance values for the top-20 features are as shown here:



Feature-importance values

Out-of-bag testing

Random forests offer the benefit of built-in cross-validation because individual trees are trained on bootstrapped versions of the training data. As a result, each tree uses on average only two-thirds of the available observations. To see why, consider that a bootstrap sample has the same size, n, as the original sample, and each observation has the same probability, $1/n$, to be drawn. Hence, the probability of not entering a bootstrap sample at all is $(1-1/n)^n$, which converges (quickly) to $1/e$, or roughly one-third.

This remaining one-third of the observations that are not included in the training set used to grow a bagged tree is called **out-of-bag** (**OOB**) observations and can serve as a validation set. Just as with cross-validation, we predict the response for an OOB sample for each tree built without this observation, and then average the predicted responses (if regression is the goal) or take a majority vote or predicted probability (if classification is the goal) for a single ensemble prediction for each OOB sample. These predictions produce an unbiased estimate of the generalization error, conveniently computed during training.

The resulting OOB error is a valid estimate of the generalization error for this observation because the prediction is produced using decision rules learned in the absence of this observation. Once the random forest is sufficiently large, the OOB error closely approximates the leave-one-out cross-validation error. The OOB approach to estimate the test error is very efficient for large datasets where cross-validation can be computationally costly.

Pros and cons of random forests

Bagged ensemble models have both advantages and disadvantages. The advantages of random forests include:

- The predictive performance can compete with the best supervised learning algorithms
- They provide a reliable feature importance estimate
- They offer efficient estimates of the test error without incurring the cost of repeated model training associated with cross-validation