11
Random Forests – A Long-Short Strategy for Japanese Stocks

In this chapter, we will learn how to use two new classes of machine learning models for trading: **decision trees** and **random forests**. We will see how decision trees learn rules from data that encode nonlinear relationships between the input and the output variables. We will illustrate how to train a decision tree and use it for prediction with regression and classification problems, visualize and interpret the rules learned by the model, and tune the model's hyperparameters to optimize the bias-variance trade-off 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. 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 (potentially large) number of decision trees, can dramatically reduce prediction errors, at the expense of some loss in interpretation.

Then, we will proceed and build a long-short trading strategy that uses a random forest to generate profitable signals for large-cap Japanese equities over the last 3 years. We will source and prepare the stock price data, tune the hyperparameters of a random forest model, and backtest trading rules based on the model's signals. The resulting long-short strategy uses machine learning rather than the cointegration relationship we saw in *Chapter 9*, *Time-Series Models for Volatility Forecasts and Statistical Arbitrage*, to identify and trade baskets of securities whose prices will likely move in opposite directions over a given investment horizon.

In short, after reading this chapter, you will be able to:

- Use decision trees for regression and classification
- Gain insights from decision trees and visualize the rules learned from the data
- Understand why ensemble models tend to deliver superior results
- Use bootstrap aggregation to address the overfitting challenges of decision trees
- Train, tune, and interpret random forests
- Employ a random forest to design and evaluate a profitable trading strategy

> You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

Decision trees – learning rules from data

A decision tree is a machine learning algorithm that predicts the value of a target variable based on **decision rules learned from data**. The algorithm can be applied to both regression and classification problems by changing the objective function that governs how the tree learns the 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. See Rokach and Maimon (2008) and Hastie, Tibshirani, and Friedman (2009) for additional details and further background information.

How trees learn and apply decision rules

The **linear models** we studied in *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*, and *Chapter 9*, *Time-Series Models for Volatility Forecasts and Statistical Arbitrage*, learn a set of parameters to predict the outcome using a linear combination of the input variables, possibly after being transformed 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 val-

ues for the subset of training samples that result from the application of a given sequence of rules. **Classification trees** predict a probability estimated from the relative class frequencies or the value of the majority class directly, whereas **regression trees** 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 for 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 it arrives at a leaf node, where the model makes a prediction.

For a linear model, the parameter values allow an interpretation of the impact of the input variables on the output and the model's prediction. In contrast, for a decision tree, the various possible paths from the root to the leaves determine how the features and their values lead to specific decisions by the model. As a consequence, decision trees are **capable of capturing interdependence** among features that linear models cannot capture "out of the box."

The following diagram 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. It does so using the subsets that would result from the split, weighted by the number of samples in each subset:

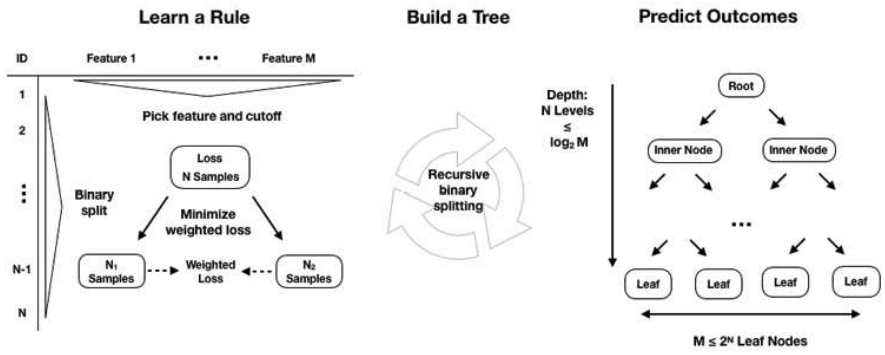


Figure 11.1: How a decision tree learns rules from data

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 between them.

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.

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 methods to limit splits and prevent this natural tendency of decision trees to produce extreme overfitting.

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.

Decision trees in practice

In this section, we will 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.

The data – monthly stock returns and features

We will select a subset of the Quandl US equity dataset covering the period 2006-2017 and follow a process similar to our first feature engineering example in *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*. We will compute monthly returns and 25 (hopefully) predictive features for the 500 most-traded stocks based on the 5-year moving average of their dollar volume, yielding 56,756 observations. The features include:

- **Historical returns** for the past 1, 3, 6, and 12 months.
- **Momentum indicators** that relate the most recent 1- or 3-month returns to those for longer horizons.
- **Technical indicators** designed to capture volatility like the (normalized) average true range (NATR and ATR) and momentum like the **relative strength index** (**RSI**).
- **Factor loadings** for the five Fama-French factors based on rolling OLS regressions.
- **Categorical variables** for year and month, as well as sector.

Figure 11.2 displays the mutual information between these features and the monthly returns we use for regression (left panel) and their binarized classification counterpart, which represents positive or negative price moves for the same period. It shows that, on a univariate basis, there appear to be substantial differences in the signal content regarding both outcomes across the features.

More details can be found in the `data_prep` notebook in the GitHub repository for this chapter. The decision tree models in this chapter are not equipped to handle missing or categorical variables, so we will drop the former and apply dummy encoding (see *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors* and *Chapter 6*, *The Machine Learning Process*) to the categorical sector variable:

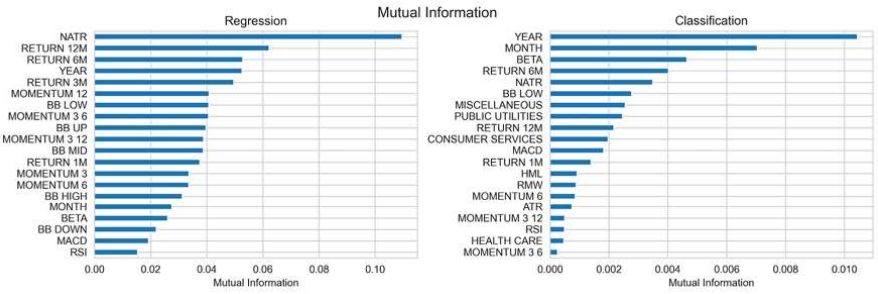


Figure 11.2: Mutual information for features and returns or price move direction

Building a regression tree with time-series data

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 p predictors, X_1, X_2, ..., X_p, and n possible cutpoints, s_1, s_2, ..., s_n, to find an optimal combination. The optimal rule splits the feature space into two regions, $\{X|X_i < s_j\}$ and $\{X|X_i > s_j\}$, 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 also demonstrate how we can use time-series data with a decision tree. We will only use 2 months of lagged returns to predict the following month, in the vein of an AR(2) model from the previous chapter:

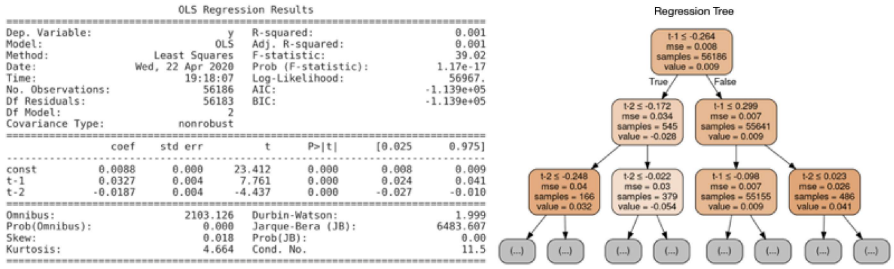
$$r_t = f(r_{t-1}, r_{t-2})$$

Using scikit-learn, configuring and training a regression tree is very straightforward:

```
from sklearn.tree import DecisionTreeRegressor
# configure regression tree
regression_tree = DecisionTreeRegressor(criterion='mse',
                                        max_depth=6,
                                        min_samples_leaf=50)
# Create training data
y = data.target
X = data.drop(target, 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 models (see *Figure 11.3*). The OLS model provides three parameters for the intercepts and the two features in line with the linear assumption this model makes about the 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 the predicted value based on these training samples. Also, note that training the decision tree takes 58 milliseconds compared to 66 microseconds for the linear regression. While both models run fast with only two features, the difference is a factor of 1,000:



OLS Regression Results

Dep. Variable:	y	R-squared:	0.001
Model:	OLS	Adj. R-squared:	0.001
Method:	Least Squares	F-statistic:	39.02
Date:	Wed, 22 Apr 2020	Prob (F-statistic):	1.17e-17
Time:	19:18:07	Log-Likelihood:	56967.
No. Observations:	56186	AIC:	-1.139e+05
Df Residuals:	56183	BIC:	-1.139e+05
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.0088	0.000	23.412	0.000	0.008	0.009
t-1	0.0327	0.004	7.761	0.000	0.024	0.041
t-2	-0.0187	0.004	-4.437	0.000	-0.027	-0.010

Omnibus:	2103.126	Durbin-Watson:	1.999
Prob(Omnibus):	0.000	Jarque-Bera (JB):	6483.607
Skew:	0.018	Prob(JB):	0.00
Kurtosis:	4.664	Cond. No.	11.5

Figure 11.3: OLS results and regression tree

The tree chart also highlights the uneven distribution of samples across the nodes as the numbers vary between 545 and 55,000 samples after the first splits.

To further illustrate the different assumptions about the functional form of the relationships between the input variables and the output, we can visualize the 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 image shows the current monthly return as a function of returns one and two periods ago for linear regression (left panel) and the regression tree:

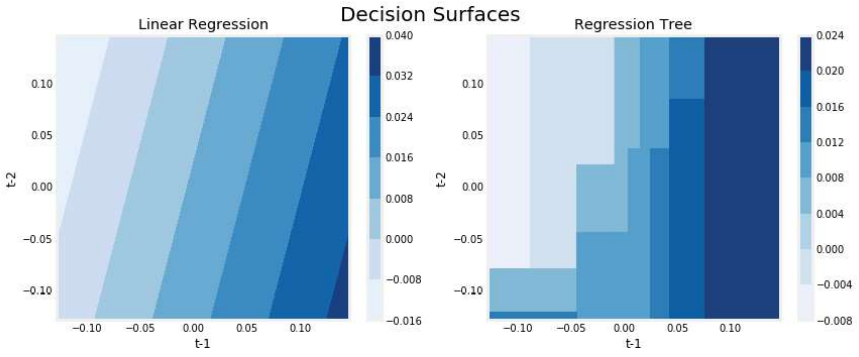


Figure 11.4: Decision surfaces for linear regression and the regression tree

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

Building a classification tree

A classification tree works just like the regression version, except that the 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 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 (leaf) node that do not belong to the most common class.

However, the alternative measures, either **Gini impurity** or **cross-entropy**, are preferred because they are more sensitive to node purity than the classification error rate, as you can see in *Figure 11.5*. **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 implies successful classification for this particular region of the feature space.

Let's see how to compute these measures for a classification outcome with K categories 0,1,..., K-1 (with K=2, in the binary case). For a given node m, let p_{mk} be the proportion of samples from the k^{th} class:

$$\text{Gini impurity} = \sum_k p_{mk}(1 - p_{mk})$$

$$\text{cross entropy} = -\sum_k p_{mk}\, log(p_{mk})$$

The following plot shows that both the Gini impurity and cross-entropy measures are maximized over the [0, 1] interval when the class proportions are even, or 0.5 in the binary case. Both measures decline when the class proportions approach zero or one and the child nodes tend toward purity as a result of a split. At the same time, they imply a higher penalty for node impurity than the classification error rate:

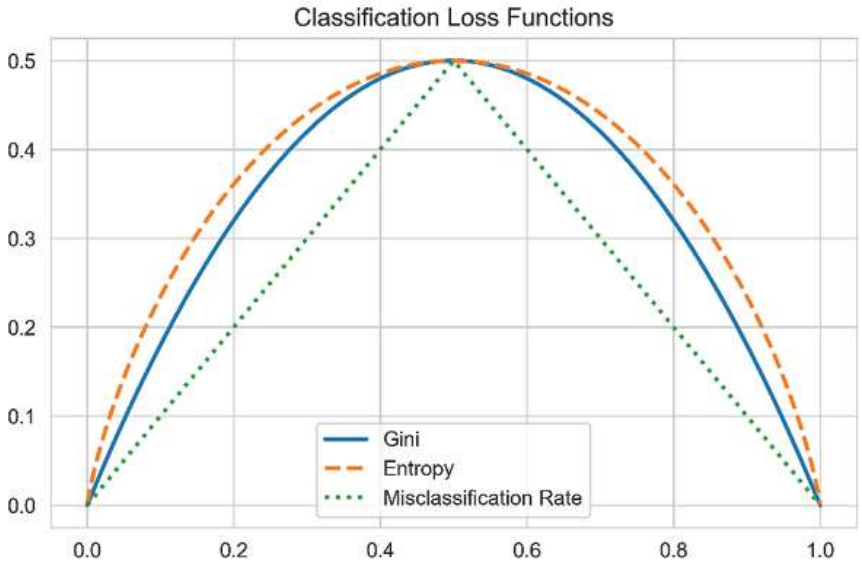


Figure 11.5: Classification loss functions

Note that cross-entropy takes almost 20 times as long to compute as the Gini measure (see the notebook for details).

How to train a classification tree

We will now train, visualize, and evaluate a classification tree with up to five consecutive splits using 80 percent of the samples for training to predict the remaining 20 percent. We will take a shortcut here to simplify the illustration and use the built-in `train_test_split`, which does not protect against lookahead bias, as the custom `MultipleTimeSeriesCV` iterator we introduced in *Chapter 6*, *The Machine Learning Process* and will use later in this chapter.

The tree configuration implies up to 2^5=32 leaf nodes that, on average, in the balanced case, would contain over 1,400 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_
# configure & train tree learner
clf = DecisionTreeClassifier(criterion='gini',
```

```
                              max_depth=5,
                              random_state=42)
clf.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. We will address these in more detail in the *Hyperparameter tuning* section.

Visualizing a decision tree

You can visualize the tree using the Graphviz library (see GitHub for installation instructions) because scikit-learn 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, # 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 following diagram shows how the model uses different features and indicates the split rules for both continuous and categorical (dummy) variables. Under the label value for each node, the chart shows 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):

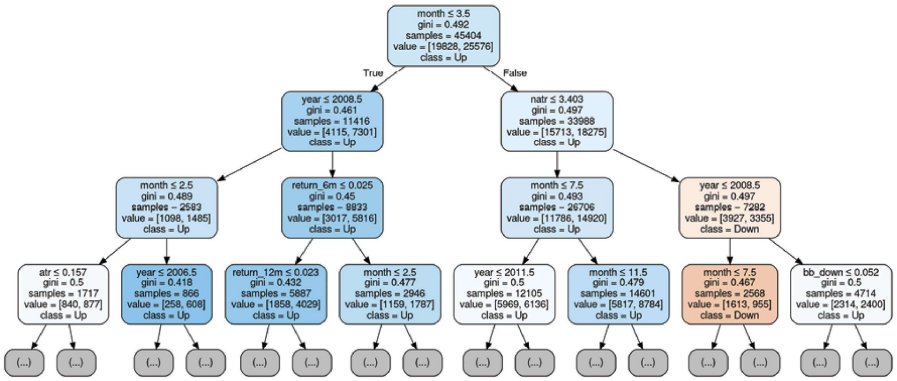


Figure 11.6: Visualization of a classification tree

Evaluating 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:

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

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, which 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 (but keep in mind that the cross-validation method here does not respect the time-series nature of the data):

```
roc_auc_score(y_score=y_score, y_true=y_test)
0.6341
```

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 multiple ways to **address the risk of overfitting**, including:

- **Dimensionality reduction** (see *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with 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. It does so 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 the key parameters available for this purpose in the scikit-learn 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 trade-off and lower prediction errors:

Parameter	Description	Default	Options
`max_depth`	The maximum number of levels: split the nodes until `max_depth` has been reached. All leaves are pure or contain fewer samples than `min_samples_split`.	None	int
`max_features`	Number of features to consider for a	None	None: all features `int` : # features

	split.		`float` : fraction `auto` , `sqrt` : sqrt(`n_features`) `log2` : log2(`n_features`)
`max_leaf_nodes`	Split nodes until creating this many leaves.	None	None: unlimited `int`
`min_impurity_decrease`	Split node if impurity decreases by at least this value.	0	`float`
`min_samples_leaf`	A split will only be considered if there are at least `min_samples_leaf` training samples in each of the left and right branches.	1	`int` ; `float` (as a percent of *N*)
`min_samples_split`	The minimum number of samples required to split an internal node.	2	`int` ; `float` (percent of *N*)
`min_weight_fraction_leaf`	The minimum weighted fraction of the sum total of all sample weights needed at a leaf node. Samples have equal weight unless `sample_weight` is provided in the fit method.	0	

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 scikit-learn documentation contains additional details about how to use the various parameters for different use cases; see the resources linked on GitHub for more information.

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. This is because it leads to overly complex trees, which are 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 in the objective metric. The downside of this strategy, however, is that sometimes, splits that result in small improvements enable more valuable splits later 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 was introduced in scikit-learn version 0.22; see Esposito et al. (1997) for a survey of how various methods work and perform.

Hyperparameter tuning

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. scikit-learn offers several tools to facilitate the process of cross-validating numerous parameter settings, namely the `GridSearchCV` convenience class, which we will illustrate in the next section. Learning curves also allow diagnostics that evaluate potential benefits of collecting additional data to reduce the generalization error.

Using GridsearchCV with a custom metric

As highlighted in *Chapter 6*, *The Machine Learning Process*, scikit-learn 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:

```
reg_tree = DecisionTreeRegressor(random_state=42)
param_grid = {'max_depth': [2, 3, 4, 5, 6, 7, 8, 10, 12, 15],
              'min_samples_leaf': [5, 25, 50, 100],
              '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 set our custom `MultipleTimeSeriesSplit` class to train the model for 60 months, or 5 years, of data and to validate performance using the subsequent 6 months, repeating the process over 10 folds to cover an out-of-sample period of 5 years:

```
cv = MultipleTimeSeriesCV(n_splits=10,
                          train_period_length=60,
                          test_period_length=6,
                          lookahead=1)
```

We use the `roc_auc` metric to score the classifier, and define a custom information coefficient (IC) metric using scikit-learn's `make_scorer` function for the regression model:

```
def rank_correl(y, y_pred):
    return spearmanr(y, y_pred)[0]
ic = make_scorer(rank_correl)
```

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

```
gridsearch_reg = GridSearchCV(estimator=reg_tree,
                              param_grid=param_grid,
                              scoring=ic,
                              n_jobs=-1,
                              cv=cv,  # custom MultipleTimeSeriesSplit
                              refit=True,
                              return_train_score=True)
gridsearch_reg.fit(X=X, y=y)
```

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, which avoids lookahead bias).

The following table lists the parameters and scores for the best regression and classification model, respectively. With a shallower tree and more regularized leaf nodes, the regression tree achieves an IC of 0.083, while the classifier's AUC score is 0.525:

Parameter	Regression	Classification
max_depth	6	12
max_features	sqrt	sqrt
min_samples_leaf	50	5
Score	0.0829	0.5250

The automation is quite convenient, but we also would like to inspect how the performance evolves for different parameter values. Upon com-

pletion of this process, the `GridSearchCV` object makes detailed cross-validation results available so that we can 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, as well as 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=10,
                                 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, = X.iloc[train_idx],
        y_train  = 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 following plot displays how the number of leaf nodes increases with the depth of the tree. Due to the sample size of each cross-validation fold containing 60 months with around 500 data points each, the number of leaf nodes is limited to around 3,000 when limiting the number of `min_samples_leaf` to 10 samples:

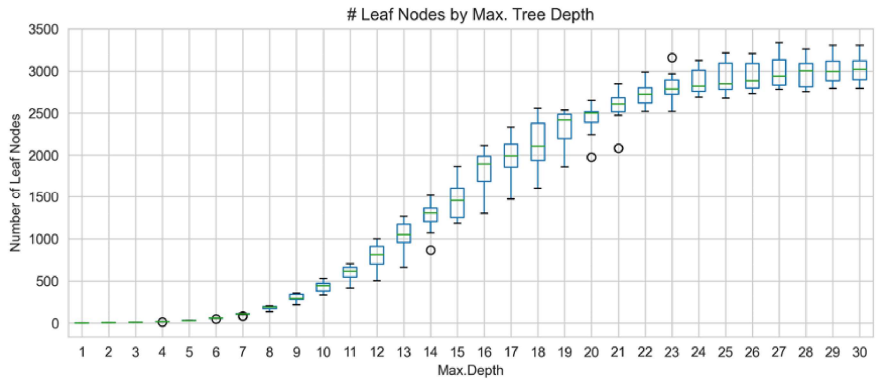


Figure 11.7: Visualization of a classification tree

Comparing regression and classification performance

To take a closer look at the performance of the models, we will show the cross-validation performance for various levels of depth, while maintaining the other parameter settings that produced the best grid search results. *Figure 11.8* displays the train and the validation scores and highlights the degree of overfitting for deeper trees. This is because the training scores steadily increase, whereas validation performance remains flat or decreases.

Note that, for the classification tree, the grid search suggested 12 levels for the best predictive accuracy. However, the plot shows similar AUC scores for less complex trees, with three or seven levels. We would prefer a shallower tree that promises comparable generalization performance while reducing the risk of overfitting:

Figure 11.8: Train and validation scores for both models

Diagnosing training set size with learning curves

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

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 also helps to diagnose whether the model's generalization error is more likely driven by bias or variance.

If the training score meets performance expectations and the validation score exhibits significant improvement as the training sample grows, training for a longer lookback period or obtaining more data might add value. If, on the other hand, both the validation and the training score converge to a similarly poor value, despite an increasing training set size, the error is more likely due to bias, and additional training data is unlikely to help.

The following image depicts the learning curves for the best regression and classification models:

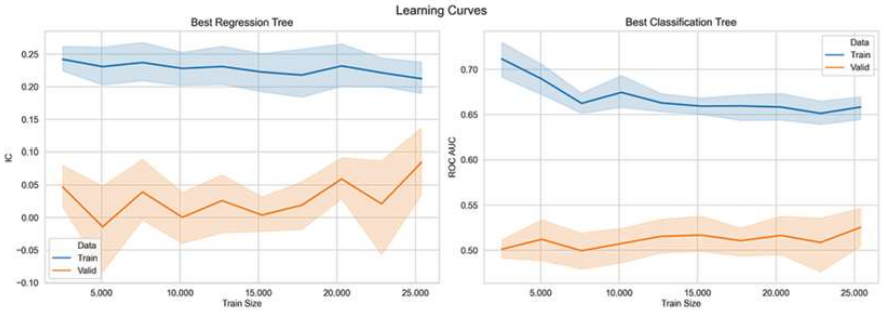


Figure 11.9: Learning curves for the best version of each model

Especially for the regression model, the validation performance improves with a larger training set. This suggests that a longer training period may yield better results. Try it yourself to see if it works!

Gaining insight from feature importance

Decision trees can not only be visualized to inspect the decision path for a given feature, but can also summarize the contribution of each feature to the rules learned by the model to fit the training data.

Feature importance captures how much the splits produced by each feature help optimize the model's metric used to evaluate the split quality, which in our case is the Gini impurity. A feature's importance is com-

puted 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 are typically considered of higher importance.

Figure 11.10 shows the plots for feature importance for the top 15 features of each model. Note how the order of features differs from the univariate evaluation based on the mutual information scores given at the beginning of this section. Clearly, the ability of decision trees to capture interdependencies, such as between time periods and other features, can alter the value of each feature:

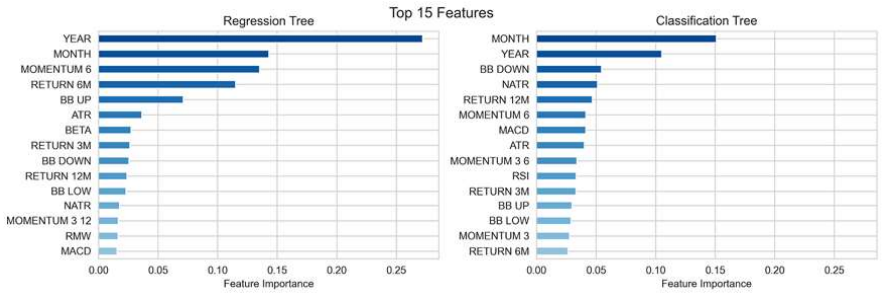


Figure 11.10: Feature importance for the best regression and classification models

Strengths and weaknesses of decision trees

Regression and classification trees approach making predictions very differently from the linear models we have explored in the previous chapters. How do you **decide which model is more suitable** for 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 nonlinear and more complex, decision trees will likely outperform the classical models. Keep in mind that the complexity of the relationship needs to be systematic or "real," rather than driven by noise, which leads more complex models to overfit.

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 accessi-

ble 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 12*, *Boosting Your Trading Strategy*, but this is not the case for scikit-learn.
- 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 the references for more details).

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. The key steps to address this weakness are pruning and regularization using the early-stopping criteria that limits tree growth, as outlined in this section.
- Decision trees are also sensitive to unbalanced class weights and may produce biased trees. One option is to oversample the underrepresented classes or undersample the more frequent class. It is typically better, though, to use class weights and directly adjust the objective function.
- The high variance of decision trees is tied to their ability to closely adapt to a training set. As a result, minor variations in the data can produce wide swings in the tree's structure and, consequently, the model's predictions. A key prevention mechanism 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 local criteria that reduce the prediction error at the current node and do not guarantee a globally optimal outcome. Again, ensembles consisting of randomized trees help to mitigate this problem.

We will now turn to the ensemble method of mitigating the risk of overfitting that's inherent when using decision trees.

Random forests – making trees more reliable

Decision trees are not only useful for their transparency and interpretability. They are also fundamental building blocks for more powerful ensemble models that combine many individual trees, while randomly varying their design to address the overfitting problems we just discussed.

Why ensemble models perform better

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 learning algorithms, to reduce the generalization error that these models produce on their own.

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

- **Accurate**: Outperform a naive baseline (such as the sample mean or class proportions)
- **Independent**: 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 between, 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 of reducing 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 12*, *Boosting Your Trading Strategy*.

Bootstrap aggregation

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 available 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 $\sigma/\sqrt{n}$. 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 that's used to reduce the variance of a machine learning model, which is particularly useful and popular when applied to decision trees.

We will first explain how this technique mitigates overfitting and then show how to apply it to decision trees.

How bagging lowers model variance

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 to reduce their generalization error by:

1. Randomizing how each tree is grown
2. Averaging their predictions

It is often a straightforward approach to improve on a given model without the need to change the underlying algorithm. This technique 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, we train one decision tree on each of these samples and create an ensemble prediction by averaging over the predictions of the different trees. You can find the code for this example in the notebook `bagged_decision_trees`, unless otherwise noted.

Bagged decision trees are usually grown large, that is, they have many levels and leaf nodes and are not pruned so that each tree has a 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 scikit-learn. 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 10 trees, each grown 10 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:

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

We will draw 100 random samples of 250 training and 500 test observations each to train each model and collect the predictions:

```
noise = .5  # noise relative to std(y)
noise = y.std() * noise
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 plots in *Figure 11.11* show:

- The mean prediction and a band of two standard deviations around the mean (upper panel)
- The bias-variance-noise breakdown based on the values for the true function (bottom panel)

We find that the variance of the predictions of the individual decision tree (left side) is almost twice as high as that for the small ensemble of 10 bagged trees, based on bootstrapped samples:

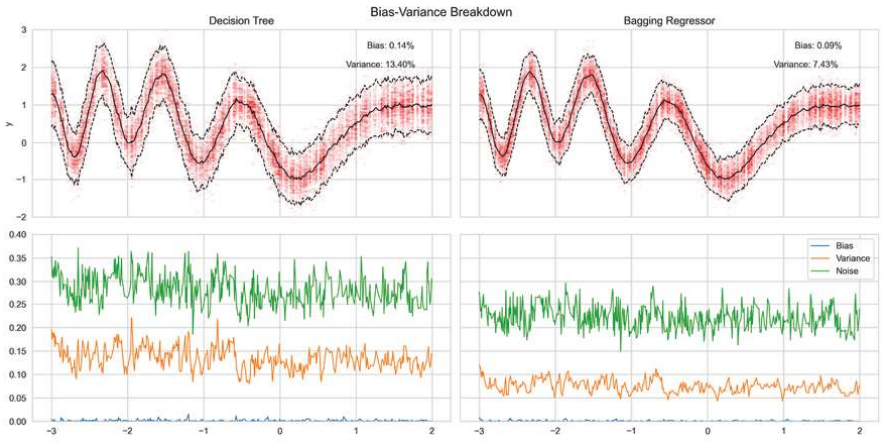


Figure 11.11: Bias-variance breakdown for individual and bagged decision trees

See the notebook `bagged_decision_trees` for implementation details.

How to build a random forest

The random forest algorithm builds on the randomization introduced by bagging to further reduce variance 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 **sample size for the features** differs between 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:

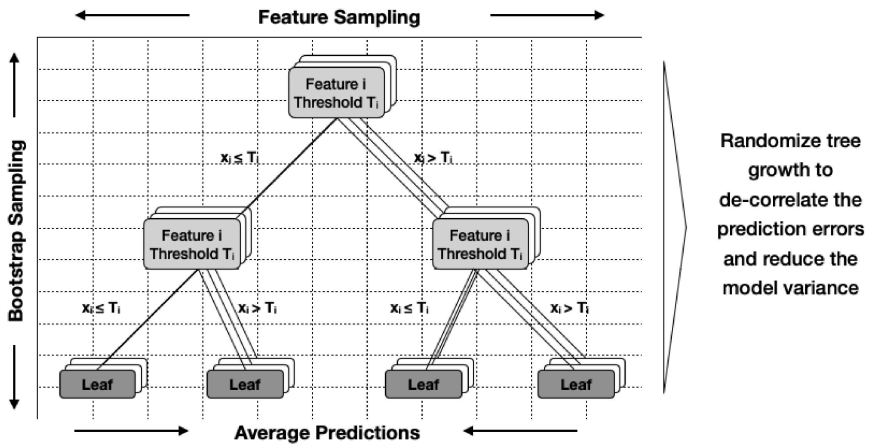


Figure 11.12: How a random forest grows individual trees

The goal of randomizing the features in addition to the training observations is to further **decorrelate 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 *How to tune the hyperparameters* section. The following table 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

The `bootstrap` parameter activates the bagging algorithm just described. Bagging, in turn, enables the computation of the out-of-bag score (`oob_score`), which estimates the generalization accuracy from samples not included in the bootstrap sample used to train a given tree (see the *Out-of-bag testing* section).

The parameter `n_estimators` 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 the number of base learners grows. The goal is to identify when the rising cost of training an additional tree outweighs the benefit of reducing the validation error, or when the latter starts to increase again.

The `max_features` parameter controls the size of the randomly selected feature subsets available when learning a new decision rule and to split a node. A lower value reduces the correlation of the trees and, thus, the ensemble's variance, but may also increase the bias. As pointed out at the beginning of this section, good starting values are the number of training features for regression problems and the square root of this number 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 scikit-learn supports 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. The code for this example is in the notebook `random_forest_tuning`.

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

```
rf_clf = RandomForestClassifier(n_estimators=100,
                                criterion='gini',
                                max_depth=None,
                                min_samples_split=2,
                                min_samples_leaf=1,
                                min_weight_fraction_leaf=0.0,
                                max_features='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)
```

We use the same 10-fold custom cross-validation as in the decision tree example previously and populate the parameter grid with values for the key configuration settings:

```
cv = MultipleTimeSeriesCV(n_splits=10, train_period_length=60,
                          test_period_length=6, lookahead=1)
clf = RandomForestClassifier(random_state=42, n_jobs=-1)
param_grid = {'n_estimators': [50, 100, 250],
              'max_depth': [5, 15, None],
              'min_samples_leaf': [5, 25, 100]}
```

Configure `GridSearchCV` using the preceding as 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)
```

We run our grid search as before and find the following result for the best-performing regression and classification models. A random forest regression model does better with shallower trees compared to the classifier but otherwise uses the same settings:

Parameter	Regression	Classification

max_depth	5	15
min_samples_leaf	5	5
n_estimators	100	100
Score	0.0435	0.5205

However, both models underperform their individual decision tree counterparts, highlighting that more complex models do not necessarily outperform simpler approaches, especially when the data is noisy and the risk of overfitting is high.

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 due to splits on this feature and is averaged over all trees. Since the objective function takes into account how many features are affected by a split, features used near the top of a tree will get higher scores due to the larger number of observations contained in the 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 score is measured in terms of the mean-squared error for regression trees and the Gini impurity or entropy for classification trees. scikit-learn further normalizes feature importance so that it sums up to 1. Feature importance thus computed is also popular 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).

Figure 11.13 shows the values for the top 15 features for both models. The regression model relies much more on time periods than the better-performing decision tree:

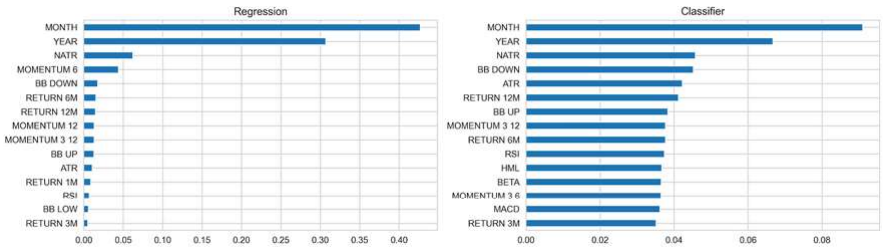


Figure 11.13: Random forest feature importance

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 is used to grow a bagged tree 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, which is conveniently computed during training.

The resulting OOB error is a valid estimate of the generalization error for this observation. This is 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.

However, the same caveats apply as for cross-validation: you need to take care to avoid a lookahead bias that would ensue if OOB observations could be selected *out-of-order*. In practice, this makes it very difficult to use OOB testing with time-series data, where the validation set needs to be selected subject to the sequential nature of the data.

Pros and cons of random forests

Bagged ensemble models have both advantages and disadvantages.

The **advantages** of random forests include:

- Depending on the use case, a random forest can perform on par with the best supervised learning algorithms.
- Random forests 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.

On the other hand, the **disadvantages** of random forests include:

- An ensemble model is inherently less interpretable than an individual decision tree.
- Training a large number of deep trees can have high computational costs (but can be parallelized) and use a lot of memory.
- Predictions are slower, which may create challenges for applications that require low latency.

Let's now take a look at how we can use a random forest for a trading strategy.

Long-short signals for Japanese stocks

In *Chapter 9*, *Time-Series Models for Volatility Forecasts and Statistical Arbitrage*, we used cointegration tests to identify pairs of stocks with a long-term equilibrium relationship in the form of a common trend to which their prices revert.

In this chapter, we will use the predictions of a machine learning model to identify assets that are likely to go up or down so we can enter market-neutral long and short positions, accordingly. The approach is similar to our initial trading strategy that used linear regression in *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*, and *Chapter 8*, *The ML4T Workflow – From Model to Strategy Backtesting*.

Instead of the scikit-learn random forest implementation, we will use the LightGBM package, which has been primarily designed for gradient boosting. One of several advantages is LightGBM's ability to efficiently encode categorical variables as numeric features rather than using one-hot dummy encoding (Fisher 1958). We'll provide a more detailed introduc-

tion in the next chapter, but the code samples should be easy to follow as the logic is similar to the scikit-learn version.

The data – Japanese equities

We are going to design a strategy for a universe of Japanese stocks, using data provided by Stooq, a Polish data provider that currently offers interesting datasets for various asset classes, markets, and frequencies, which we also relied upon in *Chapter 9*, *Time-Series Models for Volatility Forecasts and Statistical Arbitrage*.

While there is little transparency regarding the sourcing and quality of the data, it has the powerful advantage of currently being free of charge. In other words, we get to experiment with data on stocks, bonds, commodities, and FX at daily, hourly, and 5-minute frequencies, but should take the results with a large grain of salt.

The `create_datasets` notebook in the data directory of this book's GitHub repository contains instructions for downloading the data and storing them in HDF5 format. For this example, we are using price data on some 3,000 Japanese stocks for the 2010-2019 period. The last 2 years will serve as the out-of-sample test period, while the prior years will serve as our cross-validation sample for model selection.

Please refer to the notebook `japanese_equity_features` for the code samples in this section. We remove tickers with more than five consecutive missing values and only keep the 1,000 most-traded stocks.

The features – lagged returns and technical indicators

We'll keep it relatively simple and combine historical returns for 1, 5, 10, 21, and 63 trading days with several technical indicators provided by TA-Lib (see *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*).

More specifically, we compute for each stock:

- **Percentage price oscillator** (**PPO**): A normalized version of the **moving average convergence/divergence** (**MACD**) indicator that measures the difference between the 14-day and the 26-day exponential moving average to capture differences in momentum across assets.
- **Normalized average true range** (**NATR**): Measures price volatility in a way that can be compared across assets.
- **Relative strength index** (**RSI**): Another popular momentum indicator (see *Chapter 4*, *Financial Feature Engineering – How to Research Alpha*

Factors for details).
- **Bollinger Bands**: Ratios of the moving average to the moving standard deviations used to identify opportunities for mean reversion.

We will also include markers for the time periods year, month, and weekday, and rank stocks on a scale from 1 to 20 with respect to their latest return for each of the six intervals on each trading day.

The outcomes – forward returns for different horizons

To test the predictive ability of a random forest given these features, we generate forward returns for the same intervals up to 21 trading days (1 month).

The leads and lags implied by the historical and forward returns cause some loss of data that increases with the investment horizon. We end up with 2.3 million observations on 18 features and 4 outcomes for 941 stocks.

The ML4T workflow with LightGBM

We will now embark on selecting a random forest model that produces tradeable signals. Several studies have done so successfully; see, for instance, Krauss, Do, and Huck (2017) and Rasekhschaffe and Jones (2019) and the resources referenced there.

We will use the fast and memory-efficient LightGBM implementation that's open sourced by Microsoft and most popular for gradient boosting, which is the topic of the next chapter, where we will take a closer look at the various LightGBM features.

We will begin by discussing key experimental design decisions, then build and evaluate a predictive model whose signals will drive the trading strategy that we will design and evaluate in the final step. Please refer to the notebook `random_forest_return_signals` for the code samples in this section, unless otherwise stated.

From universe selection to hyperparameter tuning

To develop a trading strategy that uses a machine learning model, we need to make several decisions on the scope and design of the model, including:

- **Lookback period**: How many historical trading days to use for training

- **Lookahead period**: How many days into the future to predict returns
- **Test period**: For how many consecutive days to make predictions with the same model
- **Hyperparameters**: Which parameters and configurations to evaluate
- **Ensembling**: Whether to rely on a single model or some combination of multiple models

To evaluate the options of interest, we also need to select a **universe** and **time period** for cross-validation, as well as an out-of-sample test period and universe. More specifically, we cross-validate several options for the period up to 2017 on a subset of our sample of Japanese stocks.

Once we've settled on a model, we'll define trading rules and backtest the strategy that uses the signals of our model **out-of-sample** over the last 2 years on the complete universe to validate its performance.

For the time-series cross-validation, we'll rely on the `MultipleTimeSeriesCV` that we developed in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, to parameterize the length of the training and test period while avoiding lookahead bias. This custom CV class permits us to:

- Train the model on a consecutive sample containing `train_length` days for each ticker.
- Validate its performance during a subsequent period containing `test_length` days and `lookahead` number of days, apart from the training period, to avoid data leakage.
- Repeat for a given number of `n_splits` while rolling the train and validation periods forward for `test_length` number of days each time.

We'll work on the model selection step in this section and on strategy backtesting in the following one.

Sampling tickers to speed up cross-validation

Training a random forest takes quite a bit longer than linear regression and depends on the configuration, where the number of trees and their depth are the main drivers.

To keep our experiments manageable, we'll select the 250 most-traded stocks over the 2010-17 period to evaluate the performance of different outcomes and model configurations, as follows:

```
DATA_DIR = Path('..', 'data')
prices = (pd.read_hdf(DATA_DIR / 'assets.h5', 'stooq/jp/tse/stocks/prices')
          .loc[idx[:, '2010': '2017'], :])
dollar_vol = prices.close.mul(prices.volume)
dollar_vol_rank = dollar_vol.groupby(level='date').rank(ascending=False)
universe = dollar_vol_rank.groupby(level='symbol').mean().nsmallest(250).index
```

Defining lookback, lookahead, and roll-forward periods

Running our strategy requires training models on a rolling basis, using a certain number of trading days (the lookback period) from our universe to learn the model parameters and predict the outcome for a certain number of future days. In our example, we'll consider 63, 126, 252, 756, and 1,260 trading days for training while rolling forward and predicting for 5, 21, or 63 days during each iteration.

We will pair the parameters in a list for easy iteration and optional sampling and/or shuffling, as follows:

```
train_lengths = [1260, 756, 252, 126, 63]
test_lengths = [5, 21, 63]
test_params = list(product(train_lengths, test_lengths))
n = len(test_params)
test_param_sample = np.random.choice(list(range(n)),
                                     size=int(n),
                                     replace=False)
test_params = [test_params[i] for i in test_param_sample]
```

Hyperparameter tuning with LightGBM

The LightGBM model accepts a large number of parameters, as the documentation explains in detail (see **https://lightgbm.readthedocs.io/** and the next chapter). For our purposes, we just need to enable the random forest algorithm by defining `boosting_type`, setting `bagging_freq` to a positive number, and setting `objective` to `regression`:

```
base_params = dict(boosting_type='rf',
                   objective='regression',
                   bagging_freq=1)
```

Next, we select the hyperparameters most likely to affect the predictive accuracy, namely:

- The number of trees to grow for the model (`num_boost_round`)

- The share of rows (`bagging_fraction`) and columns (`feature_fraction`) used for bagging
- The minimum number of samples required in a leaf (`min_data_in_leaf`) to control for overfitting

Another benefit of LightGBM is that we can evaluate a trained model for a subset of its trees (or continue training after a certain number of evaluations), which allows us to test multiple `num_iteration` values during a single training session.

Alternatively, you can enable `early_stopping` to interrupt training when the loss metric for a validation set no longer improves. However, the cross-validation performance estimates will be biased upward as the model uses information on the outcome that will not be available under realistic circumstances.

We'll use the following values for the hyperparameters, which control the bagging method and tree growth:

```
bagging_fraction_opts = [.5, .75, .95]
feature_fraction_opts = [.75, .95]
min_data_in_leaf_opts = [250, 500, 1000]
cv_params = list(product(bagging_fraction_opts,
                         feature_fraction_opts,
                         min_data_in_leaf_opts))
n_cv_params = len(cv_params)
```

Cross-validating signals over various horizons

To evaluate a model for a given set of hyperparameters, we will generate predictions using the lookback, lookahead, and roll-forward periods.

First, we will identify categorical variables because LightGBM does not require one-hot encoding; instead, it sorts the categories according to the outcome, which delivers better results for regression trees, according to Fisher (1958). We'll create variables to identify different periods:

```
categoricals = ['year', 'weekday', 'month']
for feature in categoricals:
    data[feature] = pd.factorize(data[feature], sort=True)[0]
```

To this end, we will create the binary LightGBM Dataset and configure `MultipleTimeSeriesCV` using the given `train_length` and

`test_length`, which determine the number of splits for our 2-year validation period:

```
for train_length, test_length in test_params:
    n_splits = int(2 * YEAR / test_length)
    cv = MultipleTimeSeriesCV(n_splits=n_splits,
                              test_period_length=test_length,
                              lookahead=lookahead,
                              train_period_length=train_length)
    label = label_dict[lookahead]
    outcome_data = data.loc[:, features + [label]].dropna()
    lgb_data = lgb.Dataset(data=outcome_data.drop(label, axis=1),
                           label=outcome_data[label],
                           categorical_feature=categoricals,
                           free_raw_data=False)
```

Next, we take the following steps:

1. Select the hyperparameters for this iteration.
2. Slice the binary LightGM Dataset we just created into train and test sets.
3. Train the model.
4. Generate predictions for the validation set for a range of `num_iteration` settings:

```
    for p, (bagging_fraction, feature_fraction, min_data_in_leaf) \
            in enumerate(cv_params_):
        params = base_params.copy()
        params.update(dict(bagging_fraction=bagging_fraction,
                           feature_fraction=feature_fraction,
                           min_data_in_leaf=min_data_in_leaf))
        start = time()
        cv_preds, nrounds = [], []
        for i, (train_idx, test_idx) in \
                enumerate(cv.split(X=outcome_data)):
            lgb_train = lgb_data.subset(train_idx.tolist()).construct()
            lgb_test = lgb_data.subset(test_idx.tolist()).construct()
            model = lgb.train(params=params,
                              train_set=lgb_train,
                              num_boost_round=num_boost_round,
                              verbose_eval=False)
            test_set = outcome_data.iloc[test_idx, :]
            X_test = test_set.loc[:, model.feature_name()]
            y_test = test_set.loc[:, label]
            y_pred = {str(n): model.predict(X_test, num_iteration=n)
                      for n in num_iterations}
            cv_preds.append(y_test.to_frame('y_test')
```

```
                                .assign(**y_pred).assign(i=i))
        nrounds.append(model.best_iteration)
```

5. To evaluate the validation performance, we compute the IC for the complete set of predictions, as well as on a daily basis, for a range of numbers of iterations:

```
df = [by_day.apply(lambda x: spearmanr(x.y_test,
                                       x[str(n)])[0]).to_frame(n)
      for n in num_iterations]
ic_by_day = pd.concat(df, axis=1)
daily_ic.append(ic_by_day.assign(bagging_fraction=bagging_fraction,
                                 feature_fraction=feature_fraction,
                                 min_data_in_leaf=min_data_in_leaf))
cv_ic = [spearmanr(cv_preds.y_test, cv_preds[str(n)])[0]
         for n in num_iterations]
ic.append([bagging_fraction, feature_fraction,
           min_data_in_leaf, lookahead] + cv_ic)
```

Now, we need to assess the signal content of the predictions to select a model for our trading strategy.

Analyzing cross-validation performance

First, we'll take a look at the distribution of the IC for the various train and test windows, as well as prediction horizons across all hyperparameter settings. Then, we'll take a closer look at the impact of the hyperparameter settings on the model's predictive accuracy.

IC for different lookback, roll-forward, and lookahead periods

The following image illustrates the distribution and quantiles of the daily mean IC for four prediction horizons and five training windows, as well as the best-performing 21-day test window. Unfortunately, it does not yield conclusive insights into whether shorter or longer windows do better, but rather illustrates the degree of noise in the data due to the range of model configurations we tested and the resulting lack of consistency in outcomes:

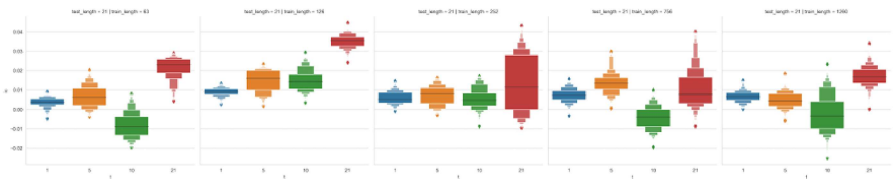


Figure 11.14: Distribution of the daily mean information coefficient for various model configurations

OLS regression of random forest configuration parameters

To understand in more detail how the parameters of our experiment affect the outcome, we can run an OLS regression of these parameters on the daily mean IC. *Figure 11.15* shows the coefficients and confidence intervals for the 1- and 5-day lookahead periods.

All variables are one-hot encoded and can be interpreted relative to the smallest category of each that is captured by the constant. The results differ across the horizons; the longest training period works best for the 1-day prediction but yields the worst performance for 5 days, with no clear patterns. Longer training appears to improve the 1-day model up to a certain point, but this is less clear for the 5-day model. The only somewhat consistent result seems to suggest a lower bagging fraction and higher minimum sample settings:

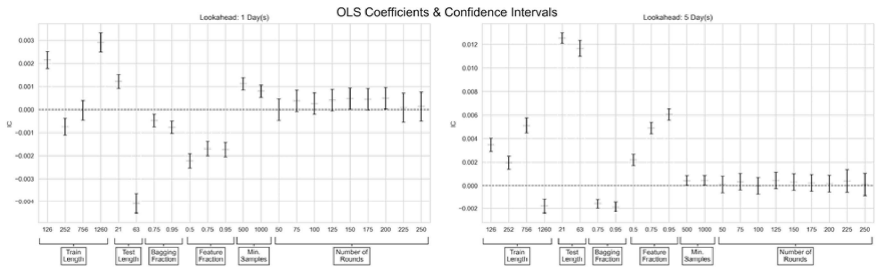


Figure 11.15: OLS coefficients and confidence intervals for the various random forest configuration parameters

Ensembling forecasts – signal analysis using Alphalens

Ultimately, we care about the signal content of the model predictions regarding our investment universe and holding period. To this end, we'll evaluate the return spread produced by equal-weighted portfolios invested in different quantiles of the predicted returns using Alphalens.

As discussed in *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*, Alphalens computes and visualizes various metrics that summarize the predictive performance of an Alpha Factor. The notebook `alphalens_signals_quality` illustrates how to combine the model predictions with price data in the appropriate format using the utility function `get_clean_factor_and_forward_returns`.

To address some of the noise inherent in the CV predictions, we select the top three 1-day models according to their mean daily IC and average their results.

When we provide the resulting signal to Alphalens, we find the following for a 1-day holding period:

- Annualized alpha of 0.081 and beta of 0.083
- A mean period-wise spread between top and bottom quintile returns of 5.16 basis points

The following image visualizes the mean period-wise returns by factor quintile and the cumulative daily forward returns associated with the stocks in each quintile:

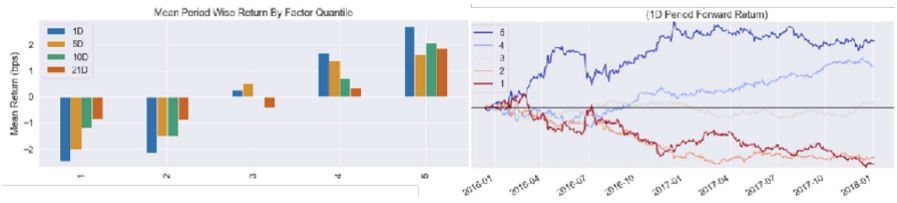


Figure 11.16: Alphalens factor signal evaluation

The preceding image shows that the 1-day ahead predictions appear to contain useful trading signals over a short horizon based on the return spread of the top and bottom quintiles. We'll now move on and develop and backtest a strategy that uses predictions generated by the top ten 1-day lookahead models that produced the results shown here for the validation period.

The strategy – backtest with Zipline

To design and backtest a trading strategy using Zipline, we need to generate predictions for our universe for the test period, ingest the Japanese equity data and load the signal into Zipline, set up a pipeline, and define rebalancing rules to trigger trades accordingly.

Ingesting Japanese Equities into Zipline

We follow the process described in *Chapter 8*, *The ML4T Workflow – From Model to Strategy Backtesting*, to convert our Stooq equity OHLCV data into a Zipline bundle. The directory `custom_bundle` contains the preprocessing module that creates the asset IDs and metadata, defines an ingest function that does the heavy lifting, and registers the bundle with an extension.

The folder contains a `README` with additional instructions.

Running an in- and out-of-sample strategy backtest

The notebook `random_forest_return_signals` shows how to select the hyperparameters that produced the best validation IC performance and generate forecasts accordingly.

We will use our 1-day model predictions and apply some simple logic: we will enter long and short positions for the 25 assets with the highest positive and lowest negative predicted returns. We will trade every day, as long as there are at least 15 candidates on either side, and close out all positions that are not among the current top forecasts.

This time, we will also include a small trading commission of $0.05 per share but will not use slippage since we are trading the most liquid Japanese stocks with a relatively modest capital base.

The results – evaluation with pyfolio

The left panel shown in *Figure 11.17* shows the in-sample (2016-17) and out-of-sample (2018-19) performance of the strategy relative to the Nikkei 225, which was mostly flat throughout the period.

The strategy earns 10.4 percent for in-sample and 5.5 percent for out-of-sample on an annualized basis.

The right panel shows the 3-month rolling Sharpe ratio, which reaches 0.96 in-sample and 0.61 out-of-sample:

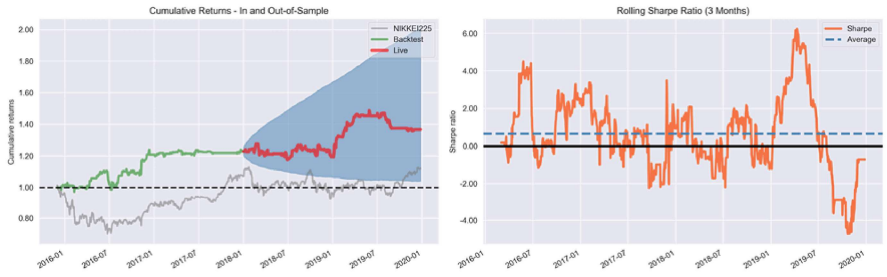


Figure 11.17: Pyfolio strategy evaluation

The overall performance statistics highlight cumulative returns of 36.6 percent after the (low) transaction costs of $0.05 cent per share, implying an out-of-sample alpha of 0.06 and a beta of 0.08 (relative to the NIKKEI 225). The maximum drawdown was 11.0 percent in-sample and 8.7 percent out-of-sample:

	All	In-sample	Out-of-sample

# Months	48	25	23
Annual return	8.00%	10.40%	5.50%
Cumulative returns	36.60%	22.80%	11.20%
Annual volatility	10.20%	10.90%	9.60%
Sharpe ratio	0.8	0.96	0.61
Calmar ratio	0.72	0.94	0.63
Stability	0.82	0.82	0.64
Max drawdown	-11.00%	-11.00%	-8.70%
Sortino ratio	1.26	1.53	0.95
Daily value at risk	-1.30%	-1.30%	-1.20%
Alpha	0.08	0.11	0.06
Beta	0.06	0.04	0.08

The pyfolio tearsheets contain lots of additional details regarding exposure, risk profile, and other aspects.

Summary

In this chapter, we learned about a new class of model capable of capturing a non-linear relationship, in contrast to the classical linear models we had explored so far. We saw how decision trees learn rules to partition the feature space into regions that yield predictions, and thus segment the input data into specific regions.

Decision trees are very useful because they provide unique insights into the relationships between features and target variables, and we saw how to visualize the sequence of decision rules encoded in the tree structure.

Unfortunately, a decision tree is prone to overfitting. We learned that ensemble models and the bootstrap aggregation method manage to overcome some of the shortcomings of decision trees and render them useful as components of much more powerful composite models.

In the next chapter, we will explore another ensemble model, boosting, which has come to be considered one of the most important machine learning algorithms.