12
Boosting Your Trading Strategy

In the previous chapter, we saw how **random forests** improve on the predictions of a decision tree by combining many trees into an ensemble. The key to reducing the high variance of an individual tree is the use of **bagging**, short for **bootstrap aggregation**, which introduces randomness into the process of growing individual trees. More specifically, bagging samples from the data with replacements so that each tree is trained on a different but equal-sized random subset, with some observations repeating. In addition, a random forest randomly selects a subset of the features so that both the rows and the columns of the training set for each tree are random versions of the original data. The ensemble then generates predictions by averaging over the outputs of the individual trees.

Individual random forest trees are usually grown deep to ensure low bias while relying on the randomized training process to produce different, uncorrelated prediction errors that have a lower variance when aggregated than individual tree predictions. In other words, the randomized training aims to decorrelate (think *diversify*) the errors of individual trees. It does this so that the ensemble is less susceptible to overfitting, has a lower variance, and thus generalizes better to new data.

This chapter explores **boosting**, an alternative ensemble algorithm for decision trees that often produces even better results. The key difference is that boosting modifies the training data for each new tree based on the cumulative errors made by the model so far. In contrast to random forests that train many trees independently using samples of the training set, boosting proceeds sequentially using reweighted versions of the data. State-of-the-art boosting implementations also adopt the randomization strategies of random forests.

Over the last three decades, boosting has become one of the most successful **machine learning** (**ML**) algorithms, dominating many ML competitions for structured, tabular data (as opposed to high-dimensional image or speech data with a more complex input-out relationship where deep learning excels). We will show how boosting works, introduce several high-performance implementations, and apply boosting to **high-frequency data** and backtest an **intraday trading strategy**.

More specifically, after reading this chapter, you will be able to:

- Understand how boosting differs from bagging and how gradient boosting evolved from adaptive boosting.
- Design and tune adaptive boosting and gradient boosting models with scikit-learn.
- Build, tune, and evaluate gradient boosting models on large datasets using the state-of-the-art implementations XGBoost, LightGBM, and

CatBoost.
- Interpret and gain insights from gradient boosting models.
- Use boosting with high-frequency data to design an intraday 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.

Getting started – adaptive boosting

Like bagging, boosting is an ensemble learning algorithm that combines base learners (typically decision trees) into an ensemble. Boosting was initially developed for classification problems, but can also be used for regression, and has been called one of the most potent learning ideas introduced in the last 20 years (Hastie, Tibshirani, and Friedman 2009). Like bagging, it is a general method or metamethod that can be applied to many statistical learning methods.

The motivation behind boosting was to find a method that **combines** the outputs of **many weak models**, where "weak" means they perform only slightly better than a random guess, into a highly **accurate**, **boosted joint prediction** (Schapire and Freund 2012).

In general, boosting learns an additive hypothesis, H_M, of a form similar to linear regression. However, each of the m= 1,..., M elements of the summation is a weak base learner, called h_t, which itself requires training. The following formula summarizes this approach:

$$H_M(x) = \sum_{m=1}^{M} \underbrace{h_t(x)}_{\text{weak learner}}$$

As discussed in the previous chapter, bagging trains base learners on different random samples of the data. Boosting, in contrast, proceeds sequentially by training the base learners on data that it repeatedly modifies to reflect the cumulative learning. The goal is to ensure that the next base learner compensates for the shortcomings of the current ensemble. We will see in this chapter that boosting algorithms differ in how they define shortcomings. The ensemble makes predictions using a weighted average of the predictions of the weak models.

The first boosting algorithm that came with a mathematical proof that it enhances the performance of weak learners was developed by Robert Schapire and Yoav Freund around 1990. In 1997, a practical solution for classification problems emerged in the form of the **adaptive boosting** (**AdaBoost**) algorithm, which won the Göedel Prize in 2003 (Freund and Schapire 1997). About another 5 years later, this algorithm was extended to arbitrary objective functions when Leo Breiman (who invented ran-

dom forests) connected the approach to gradient descent, and Jerome Friedman came up with **gradient boosting** in 1999 (Friedman 2001).

Numerous optimized implementations, such as XGBoost, LightGBM, and CatBoost, which we will look at later in this chapter, have emerged in recent years and firmly established gradient boosting as the go-to solution for structured data. In the following sections, we'll briefly introduce AdaBoost and then focus on the gradient boosting model, as well as the three state-of-the-art implementations of this very powerful and flexible algorithm we just mentioned.

The AdaBoost algorithm

When it emerged in the 1990s, AdaBoost was the first ensemble algorithm to iteratively adapt to the cumulative learning progress when fitting an additional ensemble member. In particular, AdaBoost changed the weights on the training data to reflect the cumulative errors of the current ensemble on the training set, before fitting a new, weak learner. AdaBoost was the most accurate classification algorithm at the time, and Leo Breiman referred to it as the best off-the-shelf classifier in the world at the 1996 NIPS conference (Hastie, Tibshirani, and Friedman 2009).

Over the subsequent decades, the algorithm had a large impact on machine learning because it provided theoretical performance guarantees. These guarantees only require sufficient data and a weak learner that reliably predicts just better than a random guess. As a result of this adaptive method that learns in stages, the development of an accurate ML model no longer required accurate performance over the entire feature space. Instead, the design of a model could focus on finding weak learners that just outperformed a coin flip using a small subset of the features.

In contrast to bagging, which builds ensembles of very large trees to reduce bias, AdaBoost grows shallow trees as weak learners, often producing superior accuracy with stumps—that is, trees formed by a single split. The algorithm starts with an equally weighted training set and then successively alters the sample distribution. After each iteration, AdaBoost increases the weights of incorrectly classified observations and reduces the weights of correctly predicted samples so that subsequent weak learners focus more on particularly difficult cases. Once trained, the new decision tree is incorporated into the ensemble with a weight that reflects its contribution to reducing the training error.

The AdaBoost algorithm for an ensemble of base learners, $h_m(x)$, $m=1$, ..., M, that predicts discrete classes, $y \in [-1, 1]$, and N training observations can be summarized as follows:

1. Initialize sample weights $w_i=1/N$ for observations $i=1$, ..., N.
2. For each base classifier, h_m, $m=1$, ..., M, do the following:
    1. Fit $h_m(x)$ to the training data, weighted by w_i.
    2. Compute the base learner's weighted error rate ε_m on the training set.
    3. Compute the base learner's ensemble weight α_m as a function of its error rate, as shown in the following formula:

$$\alpha_m = \log\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right)$$

4. Update the weights for misclassified samples according to $w_i * exp(\alpha_m)$

3. Predict the positive class when the weighted sum of the ensemble members is positive, and negative otherwise, as shown in the following formula:

$$H(x) = \text{sign}\left(\sum_{m=1}^{M} \underbrace{\alpha_m h_m(x)}_{\text{weighted weak learner}}\right)$$

AdaBoost has many practical **advantages**, including ease of implementation and fast computation, and can be combined with any method for identifying weak learners. Apart from the size of the ensemble, there are no hyperparameters that require tuning. AdaBoost is also useful for identifying outliers because the samples that receive the highest weights are those that are consistently misclassified and inherently ambiguous, which is also typical for outliers.

There are also **disadvantages**: the performance of AdaBoost on a given dataset depends on the ability of the weak learner to adequately capture the relationship between features and outcome. As the theory suggests, boosting will not perform well when there is insufficient data, or when the complexity of the ensemble members is not a good match for the complexity of the data. It can also be susceptible to noise in the data.

See Schapire and Freund (2012) for a thorough introduction and review of boosting algorithms.

Using AdaBoost to predict monthly price moves

As part of its ensemble module, scikit-learn provides an `AdaBoostClassifier` implementation that supports two or more classes. The code examples for this section are in the notebook `boosting_baseline`, which compares the performance of various algorithms with a dummy classifier that always predicts the most frequent class.

We need to first define a `base_estimator` as a template for all ensemble members and then configure the ensemble itself. We'll use the default `DecisionTreeClassifier` with `max_depth=1` — that is, a stump with a single split. Alternatives include any other model from linear or logistic regression to a neural network that conforms to the scikit-learn interface (see the documentation). However, decision trees are by far the most common in practice.

The complexity of `base_estimator` is a key tuning parameter because it depends on the nature of the data. As demonstrated in the previous chap-

ter, changes to `max_depth` should be combined with appropriate regularization constraints using adjustments to, for example, `min_samples_split`, as shown in the following code:

```
base_estimator = DecisionTreeClassifier(criterion='gini',
                                        splitter='best',
                                        max_depth=1,
                                        min_samples_split=2,
                                        min_samples_leaf=20,
                                        min_weight_fraction_leaf=0.0,
                                        max_features=None,
                                        random_state=None,
                                        max_leaf_nodes=None,
                                        min_impurity_decrease=0.0,
                                        min_impurity_split=None)
```

In the second step, we'll design the ensemble. The `n_estimators` parameter controls the number of weak learners, and `learning_rate` determines the contribution of each weak learner, as shown in the following code. By default, weak learners are decision tree stumps:

```
ada_clf = AdaBoostClassifier(base_estimator=base_estimator,
                             n_estimators=100,
                             learning_rate=1.0,
                             algorithm='SAMME.R',
                             random_state=42)
```

The main tuning parameters that are responsible for good results are `n_estimators` and the `base_estimator` complexity. This is because the depth of the tree controls the extent of the interaction among the features.

We will cross-validate the AdaBoost ensemble using the custom `OneStepTimeSeriesSplit`, a simplified version of the more flexible `MultipleTimeSeriesCV` (see *Chapter 6*, *The Machine Learning Process*). It implements a 12-fold rolling time-series split to predict 1 month ahead for the last 12 months in the sample, using all available prior data for training, as shown in the following code:

```
cv = OneStepTimeSeriesSplit(n_splits=12, test_period_length=1, shuffle=True)
def run_cv(clf, X=X_dummies, y=y, metrics=metrics, cv=cv, fit_params=None):
    return cross_validate(estimator=clf,
                          X=X,
                          y=y,
                          scoring=list(metrics.keys()),
                          cv=cv,
                          return_train_score=True,
                          n_jobs=-1,                # use all cores
                          verbose=1,
                          fit_params=fit_params)
```

The validation results show a weighted accuracy of 0.5068, an AUC score of 0.5348, and precision and recall values of 0.547 and 0.576, respectively,

implying an F1 score of 0.467. This is marginally below a random forest with default settings that achieves a validation AUC of 0.5358. *Figure 12.1* shows the distribution of the various metrics for the 12 train and test folds as a boxplot (note that the random forest perfectly fits the training set):

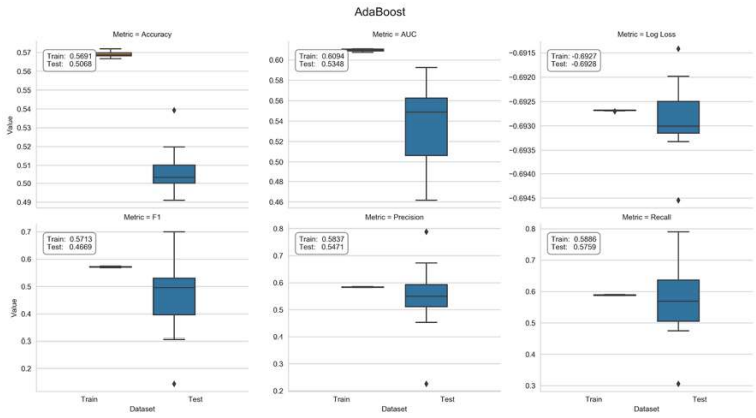

Figure 12.1: AdaBoost cross-validation performance

See the companion notebook for additional details on the code to cross-validate and process the results.

Gradient boosting – ensembles for most tasks

AdaBoost can also be interpreted as a stagewise forward approach to minimizing an exponential loss function for a binary outcome, $y \in [-1, 1]$, that identifies a new base learner, h_m, at each iteration, m, with the corresponding weight,α_m, and adds it to the ensemble, as shown in the following formula:

$$\underset{\alpha,h}{\operatorname{argmin}} \sum_{i=1}^{N} \exp\left(\underbrace{-y_i\left(f_{m-1}(x_i)\right)}_{\text{current ensemble}} + \underbrace{\alpha_m h_m(x_i)}_{\text{new member}}\right)$$

This interpretation of AdaBoost as a gradient descent algorithm that minimizes a particular loss function, namely exponential loss, was only discovered several years after its original publication.

Gradient boosting leverages this insight and **applies the boosting method to a much wider range of loss functions**. The method enables the design of machine learning algorithms to solve any regression, classification, or ranking problem, as long as it can be formulated using a loss function that is differentiable and thus has a gradient. Common example loss functions for different tasks include:

- **Regression**: The mean-squared and absolute loss
- **Classification**: Cross-entropy

- **Learning to rank**: Lambda rank loss

We covered regression and classification loss functions in *Chapter 6*, *The Machine Learning Process*; learning to rank is outside the scope of this book, but see Nakamoto (2011) for an introduction and Chen et al. (2009) for details on ranking loss.

The flexibility to customize this general method to many specific prediction tasks is essential to boosting's popularity. Gradient boosting is also not limited to weak learners and often achieves the best performance with decision trees several levels deep.

The main idea behind the resulting **gradient boosting machines** (**GBMs**) algorithm is training the base learners to learn the negative gradient of the current loss function of the ensemble. As a result, each addition to the ensemble directly contributes to reducing the overall training error, given the errors made by prior ensemble members. Since each new member represents a new function of the data, gradient boosting is also said to optimize over the functions h_m in an additive fashion.

In short, the algorithm successively fits weak learners h_m, such as decision trees, to the negative gradient of the loss function that is evaluated for the current ensemble, as shown in the following formula:

$$H_m(x) = \underbrace{H_{m-1}(x)}_{\text{current ensemble}} + \underbrace{\gamma_m h_m(x)}_{\text{new member}} = H_{m-1}(x) + \underset{\gamma, h}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{L(y_i, H_{m-1}(x_i) + h(x))}_{\text{loss function}}$$

In other words, at a given iteration m, the algorithm computes the gradient of the current loss for each observation and then fits a regression tree to these pseudo-residuals. In a second step, it identifies an optimal prediction for each leaf node that minimizes the incremental loss due to adding this new learner to the ensemble.

This differs from standalone decision trees and random forests, where the prediction depends on the outcomes for the training samples assigned to a terminal node, namely their average, in the case of regression, or the frequency of the positive class for binary classification. The focus on the gradient of the loss function also implies that gradient boosting uses regression trees to learn both regression and classification rules because the gradient is always a continuous function.

The final ensemble model makes predictions based on the weighted sum of the predictions of the individual decision trees, each of which has been trained to minimize the ensemble loss, given the prior prediction for a given set of feature values, as shown in the following diagram:

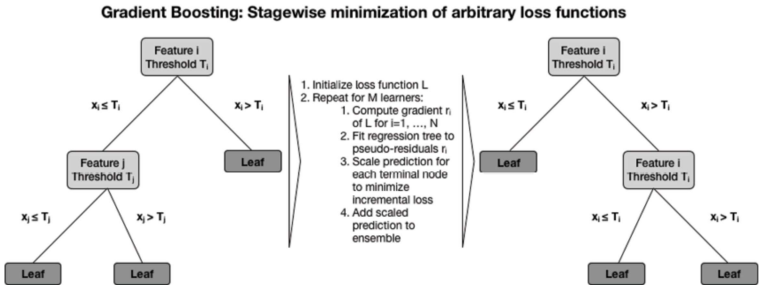

Figure 12.2: The gradient boosting algorithm

Gradient boosting trees have produced **state-of-the-art performance on many classification**, **regression**, and **ranking benchmarks**. They are probably the most popular ensemble learning algorithms as standalone predictors in a diverse set of ML competitions, as well as in real-world production pipelines, for example, to predict click-through rates for online ads.

The success of gradient boosting is based on its ability to learn complex functional relationships in an incremental fashion. However, the flexibility of this algorithm requires the careful management of the **risk of overfitting** by tuning **hyperparameters** that constrain the model's tendency to learn noise, as opposed to the signal, in the training data.

We will introduce the key mechanisms to control the complexity of a gradient boosting tree model, and then illustrate model tuning using the sklearn implementation.

How to train and tune GBM models

Boosting has often demonstrated **remarkable resilience to overfitting**, despite significant growth of the ensemble and, thus, the complexity of the model. The combination of very low and decreasing training error with non-increasing validation error is often associated with improved confidence in the predictions: as boosting continues to grow the ensemble with the goal of improving predictions for the most challenging cases, it adjusts the decision boundary to maximize the distance, or margin, of the data points.

However, overfitting certainly happens, and the **two key drivers of gradient boosting performance** are the size of the ensemble and the complexity of its constituent decision trees.

The control of the **complexity of decision trees** aims to avoid learning highly specific rules that typically imply a very small number of samples in leaf nodes. We covered the most effective constraints used to limit the ability of a decision tree to overfit to the training data in the previous chapter. They include minimum thresholds for:

- The number of samples to either split a node or accept it as a terminal node.
- The improvement in node quality, as measured by the purity or entropy for classification, or mean-squared error for regression, to fur-

ther grow the tree.

In addition to directly controlling the size of the ensemble, there are various regularization techniques, such as **shrinkage**, that we encountered in the context of the ridge and lasso linear regression models in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*. Furthermore, the randomization techniques used in the context of random forests are also commonly applied to gradient boosting machines.

Ensemble size and early stopping

Each boosting iteration aims to reduce the training loss, increasing the risk of overfitting for a large ensemble. Cross-validation is the best approach to find the optimal ensemble size that minimizes the generalization error.

Since the ensemble size needs to be specified before training, it is useful to monitor the performance on the validation set and abort the training process when, for a given number of iterations, the validation error no longer decreases. This technique is called **early stopping** and is frequently used for models that require a large number of iterations and are prone to overfitting, including deep neural networks.

Keep in mind that using early stopping with the same validation set for a large number of trials will also lead to overfitting, but just for the particular validation set rather than the training set. It is best to avoid running a large number of experiments when developing a trading strategy as the risk of **false discoveries** increases significantly. In any case, keep a **hold-out test set** to obtain an unbiased estimate of the generalization error.

Shrinkage and learning rate

Shrinkage techniques apply a penalty for increased model complexity to the model's loss function. For boosting ensembles, shrinkage can be applied by **scaling the contribution of each new ensemble member down** by a factor between 0 and 1. This factor is called the **learning rate** of the boosting ensemble. Reducing the learning rate increases shrinkage because it lowers the contribution of each new decision tree to the ensemble.

The learning rate has the opposite effect of the ensemble size, which tends to increase for lower learning rates. Lower learning rates coupled with larger ensembles have been found to reduce the test error, in particular for regression and probability estimation. Large numbers of iterations are computationally more expensive but often feasible with fast, state-of-the-art implementations as long as the individual trees remain shallow.

Depending on the implementation, you can also use **adaptive learning rates** that adjust to the number of iterations, typically lowering the impact of trees added later in the process. We will see some examples later in this chapter.

Subsampling and stochastic gradient boosting

As discussed in detail in the previous chapter, bootstrap averaging (bagging) improves the performance of an otherwise noisy classifier.

Stochastic gradient boosting samples the training data without replacement at each iteration to grow the next tree (whereas bagging uses sampling with replacement). The benefit is lower computational effort due to the smaller sample and often better accuracy, but subsampling should be combined with shrinkage.

As you can see, the number of hyperparameters keeps increasing, which drives up the number of potential combinations. As a result, the risk of false positives increases when choosing the best model from a large number of trials based on a limited amount of training data. The best approach is to proceed sequentially and select parameter values individually or use combinations of subsets of low cardinality.

How to use gradient boosting with sklearn

The ensemble module of sklearn contains an implementation of gradient boosting trees for regression and classification, both binary and multiclass. The following `GradientBoostingClassifier` initialization code illustrates the key tuning parameters. The notebook `sklearn_gbm_tuning` contains the code examples for this section. More recently (version 0.21), scikit-learn introduced a much faster, yet still experimental, `HistGradientBoostingClassifier` inspired by the implementations in the following section.

The available loss functions include the exponential loss that leads to the AdaBoost algorithm and the deviance that corresponds to the logistic regression for probabilistic outputs. The `friedman_mse` node quality measure is a variation on the mean-squared error, which includes an improvement score (see the scikit-learn documentation linked on GitHub), as shown in the following code:

```
# deviance = logistic reg; exponential: AdaBoost
gb_clf = GradientBoostingClassifier(loss='deviance',
# shrinks the contribution of each tree
                                    learning_rate=0.1,
# number of boosting stages
                                    n_estimators=100,
# fraction of samples used t fit base learners
                                    subsample=1.0,
# measures the quality of a split
                                    criterion='friedman_mse',
                                    min_samples_split=2,
                                    min_samples_leaf=1,
# min. fraction of sum of weights
                                    min_weight_fraction_leaf=0.0,
# opt value depends on interaction
                                    max_depth=3,
                                    min_impurity_decrease=0.0,
                                    min_impurity_split=None,
                                    max_features=None,
                                    max_leaf_nodes=None,
                                    warm_start=False,
```

```
                                    presort='auto',
                                    validation_fraction=0.1,
                                    tol=0.0001)
```

Similar to `AdaBoostClassifier`, this model cannot handle missing values. We'll again use 12-fold cross-validation to obtain errors for classifying the directional return for rolling 1-month holding periods, as shown in the following code:

```
gb_cv_result = run_cv(gb_clf, y=y_clean, X=X_dummies_clean)
gb_result = stack_results(gb_cv_result)
```

We parse and plot the result to find a slight improvement—using default parameter values—over both `AdaBoostClassifier` and the random forest as the test AUC increases to 0.537. *Figure 12.3* shows boxplots for the various loss metrics we are tracking:

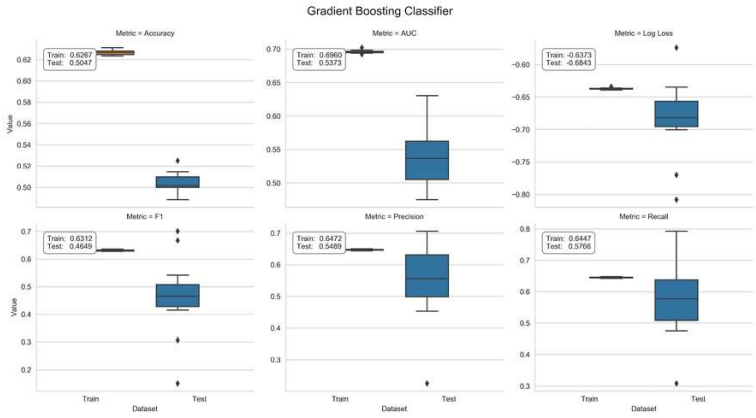

Figure 12.3: Cross-validation performance of the scikit-learn gradient boosting classifier

How to tune parameters with GridSearchCV

The `GridSearchCV` class in the `model_selection` module facilitates the systematic evaluation of all combinations of the hyperparameter values that we would like to test. In the following code, we will illustrate this functionality for seven tuning parameters, which, when defined, will result in a total of $24 \times 32 \times 4 = 576$ different model configurations:

```
cv = OneStepTimeSeriesSplit(n_splits=12)
param_grid = dict(
        n_estimators=[100, 300],
        learning_rate=[.01, .1, .2],
        max_depth=list(range(3, 13, 3)),
        subsample=[.8, 1],
        min_samples_split=[10, 50],
        min_impurity_decrease=[0, .01],
        max_features=['sqrt', .8, 1])
```

The `.fit()` method executes the cross-validation using the custom `OneStepTimeSeriesSplit` and the `roc_auc` score to evaluate the 12 folds. Sklearn lets us persist the result, as it would for any other model, using the `joblib` pickle implementation, as shown in the following code:

```
gs = GridSearchCV(gb_clf,
                  param_grid,
                  cv=cv,
                  scoring='roc_auc',
                  verbose=3,
                  n_jobs=-1,
                  return_train_score=True)
gs.fit(X=X, y=y)
# persist result using joblib for more efficient storage of large numpy arrays
joblib.dump(gs, 'gbm_gridsearch.joblib')
```

The `GridSearchCV` object has several additional attributes, after completion, that we can access after loading the pickled result. We can use them to learn which hyperparameter combination performed best and its average cross-validation AUC score, which results in a modest improvement over the default values. This is shown in the following code:

```
pd.Series(gridsearch_result.best_params_)
learning_rate              0.01
max_depth                  9.00
max_features               1.00
min_impurity_decrease      0.01
min_samples_split         50.00
n_estimators             300.00
subsample                  1.00
gridsearch_result.best_score_
0.5569
```

Parameter impact on test scores

The `GridSearchCV` result stores the average cross-validation scores so that we can analyze how different hyperparameter settings affect the outcome.

The six seaborn swarm plots in the right panel of *Figure 12.4* show the distribution of AUC test scores for all hyperparameter values. In this case, the highest AUC test scores required a low `learning_rate` and a large value for `max_features`. Some parameter settings, such as a low `learning_rate`, produce a wide range of outcomes that depend on the complementary settings of other parameters:

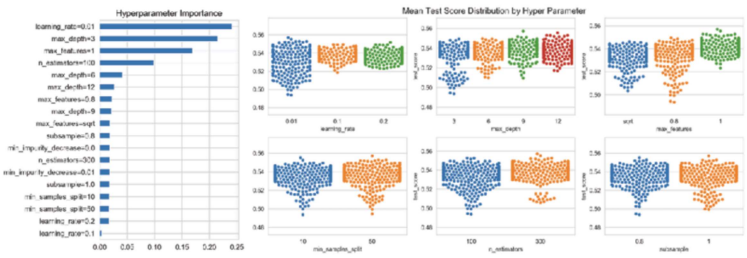

Figure 12.4: Hyperparameter impact for the scikit-learn gradient boosting model

We will now explore how hyperparameter settings jointly affect the cross-validation performance. To gain insight into how parameter settings interact, we can train a `DecisionTreeRegressor` with the mean CV AUC as the outcome and the parameter settings, encoded in one-hot or dummy format (see the notebook for details). The tree structure highlights that using all features (`max_features=1`), a low `learning_rate` , and a `max_depth` above three led to the best results, as shown in the following diagram:

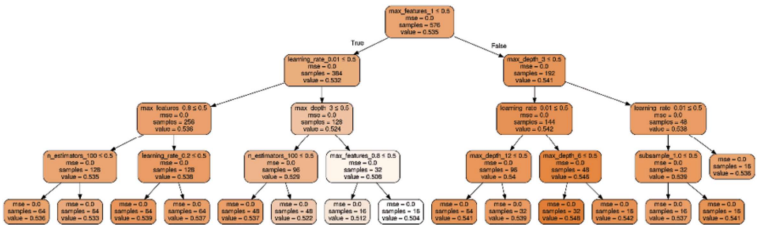

Figure 12.5: Impact of the gradient boosting model hyperparameter settings on test performance

The bar chart in the left panel of *Figure 12.4* displays the influence of the hyperparameter settings in producing different outcomes, measured by their feature importance for a decision tree that has grown to its maximum depth. Naturally, the features that appear near the top of the tree also accumulate the highest importance scores.

How to test on the holdout set

Finally, we would like to evaluate the best model's performance on the holdout set that we excluded from the `GridSearchCV` exercise. It contains the last 7 months of the sample period (through February 2018; see the notebook for details).

We obtain a generalization performance estimate based on the AUC score of 0.5381 for the first month of the hold-out period using the following code example:

```
idx = pd.IndexSlice
auc = {}
for i, test_date in enumerate(test_dates):
    test_data = test_feature_data.loc[idx[:, test_date], :]
    preds = best_model.predict(test_data)
    auc[i] = roc_auc_score(y_true=test_target.loc[test_data.index], y_score=preds)
auc = pd.Series(auc)
```

The downside of the sklearn gradient boosting implementation is the **limited training speed**, which makes it difficult to try out different hyperparameter settings quickly. In the next section, we will see that several optimized implementations have emerged over the last few years that significantly reduce the time required to train even large-scale models, and

have greatly contributed to a broader scope for applications of this highly effective algorithm.

Using XGBoost, LightGBM, and CatBoost

Over the last few years, several new gradient boosting implementations have used various innovations that accelerate training, improve resource efficiency, and allow the algorithm to scale to very large datasets. The new implementations and their sources are as follows:

- **XGBoost**: Started in 2014 by T. Chen during his Ph.D. (T. Chen and Guestrin 2016)
- **LightGBM**: Released in January 2017 by Microsoft (Ke et al. 2017)
- **CatBoost**: Released in April 2017 by Yandex (Prokhorenkova et al. 2019)

These innovations address specific challenges of training a gradient boosting model (see this chapter's `README` file on GitHub for links to the documentation). The XGBoost implementation was the first new implementation to gain popularity: among the 29 winning solutions published by Kaggle in 2015, 17 solutions used XGBoost. Eight of these solely relied on XGBoost, while the others combined XGBoost with neural networks.

We will first introduce the key innovations that have emerged over time and subsequently converged (so that most features are available for all implementations), before illustrating their implementation.

How algorithmic innovations boost performance

Random forests can be trained in parallel by growing individual trees on independent bootstrap samples. The **sequential approach of gradient boosting**, in contrast, slows down training, which, in turn, complicates experimentation with the large number of hyperparameters that need to be adapted to the nature of the task and the dataset.

To add a tree to the ensemble, the algorithm minimizes the prediction error with respect to the negative gradient of the loss function, similar to a conventional gradient descent optimizer. The **computational cost during training is thus proportional to the time it takes to evaluate potential split points** for each feature.

Second-order loss function approximation

The most important algorithmic innovations lower the cost of evaluating the loss function by using an approximation that relies on second-order derivatives, resembling Newton's method to find stationary points. As a result, scoring potential splits becomes much faster.

As discussed, a gradient boosting ensemble H_M is trained incrementally to minimize the sum of the prediction error and the regularization penalty. Denoting the prediction of the outcome y_i by the ensemble after step m as

$\hat{y}_i(m)$, as a differentiable convex loss function that measures the difference between the outcome and the prediction, and Ω as a penalty that increases with the complexity of the ensemble H_M. The incremental hypothesis $h_\mathbf{m}$ aims to minimize the following objective L:

$$\mathcal{L}^{(m)} = \sum_{i=1}^{n} \underbrace{l\left(y_i, \hat{y}_i^{(m)}\right)}_{\text{Loss at step m}} + \sum_{i=1}^{t} \underbrace{\Omega(H_m)}_{\text{Regularization}} = \sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(m-1)} + \underbrace{h_m(x_i)}_{\text{additional tree}}\right) + \Omega(H_m)$$

The regularization penalty helps to avoid overfitting by favoring a model that uses simple yet predictive regression trees. In the case of XGBoost, for example, the penalty for a regression tree h depends on the number of leaves per tree T, the regression tree scores for each terminal node w, and the hyperparameters γ and λ. This is summarized in the following formula:

$$\Omega(h) = \gamma T + \frac{1}{2}\lambda\left|\left|w\right|\right|^2$$

Therefore, at each step, the algorithm greedily adds the hypothesis h_m that most improves the regularized objective. The second-order approximation of a loss function, based on a Taylor expansion, speeds up the evaluation of the objective, as summarized in the following formula:

$$\mathcal{L}^{(m)} \simeq \sum_{i=1}^{n}\left[g_i f_m(x_i) + \frac{1}{2}h_i f_m^2(x_i)\right] + \Omega\left(h_m\right)$$

Here, g_i is the first-order gradient of the loss function before adding the new learner for a given feature value, and h_i is the corresponding second-order gradient (or Hessian) value, as shown in the following formulas:

$$g_i = \partial_{\hat{y}_i^{(m-1)}} l\left(y_i, \hat{y}_i^{(m-1)}\right)$$

$$h_i = \partial^2_{\hat{y}_i^{(m-1)}} l\left(y_i, \hat{y}_i^{(m-1)}\right)$$

The XGBoost algorithm was the first open source algorithm to leverage this approximation of the loss function to compute the optimal leaf scores for a given tree structure and the corresponding value of the loss function. The score consists of the ratio of the sums of the gradient and Hessian for the samples in a terminal node. It uses this value to score the information gain that would result from a split, similar to the node impu-

rity measures we saw in the previous chapter, but applicable to arbitrary loss functions. See Chen and Guestrin (2016) for the detailed derivation.

Simplified split-finding algorithms

The original gradient boosting implementation by sklearn finds the optimal split that enumerates all options for continuous features. This **exact greedy algorithm** is computationally very demanding due to the potentially very large number of split options for each feature. This approach faces additional challenges when the data does not fit in memory or when training in a distributed setting on multiple machines.

An **approximate split-finding** algorithm reduces the number of split points by assigning feature values to a user-determined set of bins, which can also greatly reduce the memory requirements during training. This is because only a single value needs to be stored for each bin. XGBoost introduced a **quantile sketch** algorithm that divides weighted training samples into percentile bins to achieve a uniform distribution. XGBoost also introduced the ability to handle sparse data caused by missing values, frequent zero-gradient statistics, and one-hot encoding, and can learn an optimal default direction for a given split. As a result, the algorithm only needs to evaluate non-missing values.

In contrast, LightGBM uses **gradient-based one-side sampling** (**GOSS**) to exclude a significant proportion of samples with small gradients, and only uses the remainder to estimate the information gain and select a split value accordingly. Samples with larger gradients require more training and tend to contribute more to the information gain.

LightGBM also uses exclusive feature bundling to combine features that are mutually exclusive, in that they rarely take nonzero values simultaneously, to reduce the number of features. As a result, LightGBM was the fastest implementation when released and still often performs best.

Depth-wise versus leaf-wise growth

LightGBM differs from XGBoost and CatBoost in how it prioritizes which nodes to split. LightGBM decides on splits leaf-wise, that is, it splits the leaf node that maximizes the information gain, even when this leads to unbalanced trees. In contrast, XGBoost and CatBoost expand all nodes depth-wise and first split all nodes at a given level of depth, before adding more levels. The two approaches expand nodes in a different order and will produce different results except for complete trees. The following diagram illustrates these two approaches:

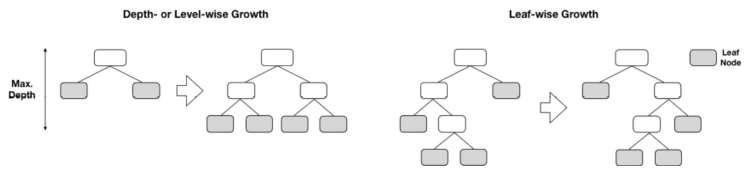

Figure 12.6: Depth-wise vs leaf-wise growth

LightGBM's leaf-wise splits tend to increase model complexity and may speed up convergence, but also increase the risk of overfitting. A tree grown depth-wise with n levels has up to 2^n terminal nodes, whereas a leaf-wise tree with 2^n leaves can have significantly more levels and contain correspondingly fewer samples in some leaves. Hence, tuning LightGBM's `num_leaves` setting requires extra caution, and the library allows us to control `max_depth` at the same time to avoid undue node imbalance. More recent versions of LightGBM also offer depth-wise tree growth.

GPU-based training

All new implementations support training and prediction on one or more GPUs to achieve significant speedups. They are compatible with current CUDA-enabled GPUs. Installation requirements vary and are evolving quickly. The XGBoost and CatBoost implementations work for several current versions, but LightGBM may require local compilation (see GitHub for links to the documentation).

The speedups depend on the library and the type of the data, and they range from low, single-digit multiples to factors of several dozen. Activation of the GPU only requires the change of a task parameter and no other hyperparameter modifications.

DART – dropout for additive regression trees

Rashmi and Gilad-Bachrach (2015) proposed a new model to train gradient boosting trees to address a problem they labeled **over-specialization**: trees added during later iterations tend only to affect the prediction of a few instances, while making a minor contribution to the remaining instances. However, the model's out-of-sample performance can suffer, and it may become over-sensitive to the contributions of a small number of trees.

The new algorithms employ dropouts that have been successfully used for learning more accurate deep neural networks, where they mute a random fraction of the neural connections during training. As a result, nodes in higher layers cannot rely on a few connections to pass the information needed for the prediction. This method has made a significant contribution to the success of deep neural networks for many tasks and has also been used with other learning techniques, such as logistic regression.

DART, or **dropout for additive regression trees**, operates at the level of trees and mutes complete trees as opposed to individual features. The goal is for trees in the ensemble generated using DART to contribute more evenly toward the final prediction. In some cases, this has been shown to produce more accurate predictions for ranking, regression, and classification tasks. The approach was first implemented in LightGBM and is also available for XGBoost.

Treatment of categorical features

The CatBoost and LightGBM implementations handle categorical variables directly without the need for dummy encoding.

The CatBoost implementation (which is named for its treatment of categorical features) includes several options to handle such features, in addition to automatic one-hot encoding. It assigns either the categories of individual features or combinations of categories for several features to numerical values. In other words, CatBoost can create new categorical features from combinations of existing features. The numerical values associated with the category levels of individual features or combinations of features depend on their relationship with the outcome value. In the classification case, this is related to the probability of observing the positive class, computed cumulatively over the sample, based on a prior, and with a smoothing factor. See the CatBoost documentation for more detailed numerical examples.

The LightGBM implementation groups the levels of the categorical features to maximize homogeneity (or minimize variance) within groups with respect to the outcome values. The XGBoost implementation does not handle categorical features directly and requires one-hot (or dummy) encoding.

Additional features and optimizations

XGBoost optimizes computation in several respects to enable multithreading. Most importantly, it keeps data in memory in compressed column blocks, where each column is sorted by the corresponding feature value. It computes this input data layout once before training and reuses it throughout to amortize the up-front cost. As a result, the search for split statistics over columns becomes a linear scan of quantiles that can be done in parallel and supports column subsampling.

The subsequently released LightGBM and CatBoost libraries built on these innovations, and LightGBM further accelerated training through optimized threading and reduced memory usage. Because of their open source nature, libraries have tended to converge over time.

XGBoost also supports **monotonicity constraints**. These constraints ensure that the values for a given feature are only positively or negatively related to the outcome over its entire range. They are useful to incorporate external assumptions about the model that are known to be true.

A long-short trading strategy with boosting

In this section, we'll design, implement, and evaluate a trading strategy for US equities driven by daily return forecasts produced by gradient boosting models. We'll use the Quandl Wiki data to engineer a few simple features (see the notebook `preparing_the_model_data` for details), select a model while using 2015/16 as validation period, and run an out-of-sample test for 2017.

As in the previous examples, we'll lay out a framework and build a specific example that you can adapt to run your own experiments. There are numerous aspects that you can vary, from the asset class and investment universe to more granular aspects like the features, holding period, or trading rules. See, for example, the Alpha Factor Library in the *Appendix* for numerous additional features.

We'll keep the trading strategy simple and only use a single ML signal; a real-life application will likely use multiple signals from different sources, such as complementary ML models trained on different datasets or with different lookahead or lookback periods. It would also use sophisticated risk management, from simple stop-loss to value-at-risk analysis.

Generating signals with LightGBM and CatBoost

XGBoost, LightGBM, and CatBoost offer interfaces for multiple languages, including Python, and have both a scikit-learn interface that is compatible with other scikit-learn features, such as `GridSearchCV` and their own methods to train and predict gradient boosting models. The notebook `boosting_baseline.ipynb` that we used in the first two sections of this chapter illustrates the scikit-learn interface for each library. The notebook compares the predictive performance and running times of various libraries. It does so by training boosting models to predict monthly US equity returns for the 2001-2018 range with the features we created in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*.

The left panel of the following image displays the predictive accuracy of the forecasts of 1-month stock price movements using default settings for all implementations, measured in terms of the mean AUC resulting from 12-fold cross-validation:

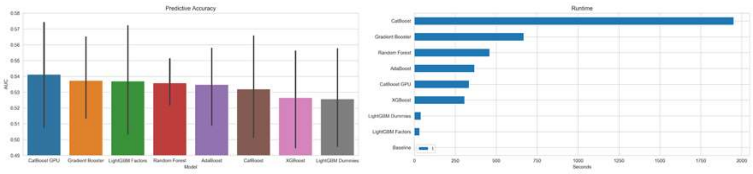

Figure 12.7: Predictive performance and runtimes of the various gradient boosting models

The **predictive performance** varies from 0.525 to 0.541. This may look like a small range but with the random benchmark AUC at 0.5, the worst-performing model improves on the benchmark by 5 percent while the best does so by 8 percent, which, in turn, is a relative rise of 60 percent. CatBoost with GPUs and LightGBM (using integer-encoded categorical variables) perform best, underlining the benefits of converting categorical into numerical variables outlined previously.

The **running time** for the experiment varies much more significantly than the predictive performance. LightGBM is 10x faster on this dataset than either XGBoost or CatBoost (using GPU) while delivering very similar predictive performance. Due to this large speed advantage and because

GPU is not available to everyone, we'll focus on LightGBM but also illustrate how to use CatBoost; XGBoost works very similarly to both.

Working with LightGBM and CatBoost models entails:

1. Creating library-specific binary data formats
2. Configuring and tuning various hyperparameters
3. Evaluating the results

We will describe these steps in the following sections. The notebook `trading_signals_with_lightgbm_and_catboost` contains the code examples for this subsection, unless otherwise noted.

From Python to C++ – creating binary data formats

LightGBM and CatBoost are written in C++ and translate Python objects, like a pandas DataFrame, into binary data formats before precomputing feature statistics to accelerate the search for split points, as described in the previous section. The result can be persisted to accelerate the start of subsequent training.

We'll subset the dataset mentioned in the preceding section through the end of 2016 to cross-validate several model configurations for various lookback and lookahead windows, as well as different roll-forward periods and hyperparameters. Our approach to model selection will be similar to the one we used in the previous chapter and uses the custom `MultipleTimeSeriesCV` introduced in *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*.

We select the train and validation sets, identify labels and features, and integer-encode categorical variables with values starting at zero, as expected by LightGBM (not necessary as long as the category codes have values less than 2^{32}, but avoids a warning):

```
data = (pd.read_hdf('data.h5', 'model_data')
            .sort_index()
            .loc[idx[:, :'2016'], :])
labels = sorted(data.filter(like='fwd').columns)
features = data.columns.difference(labels).tolist()
categoricals = ['year', 'weekday', 'month']
for feature in categoricals:
    data[feature] = pd.factorize(data[feature], sort=True)[0]
```

The notebook example iterates over many configurations, optionally using random samples to speed up model selection using a diverse subset. The goal is to identify the most impactful parameters without trying every possible combination.

To do so, we create the binary `Dataset` objects. For LightGBM, this looks as follows:

```
import lightgbm as lgb
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)
```

The CatBoost data structure is called `Pool` and works similarly:

```
cat_cols_idx = [outcome_data.columns.get_loc(c) for c in categoricals]
catboost_data = Pool(label=outcome_data[label],
                     data=outcome_data.drop(label, axis=1),
                     cat_features=cat_cols_idx)
```

For both libraries, we identify the categorical variables for conversion into numerical variables based on outcome information, as described in the previous section. The CatBoost implementation needs feature columns to be identified using indices rather than labels.

We can simply slice the binary datasets using the train and validation set indices provided by `MultipleTimeSeriesCV` during cross-validation as follows, combining both examples into one snippet:

```
for i, (train_idx, test_idx) in enumerate(cv.split(X=outcome_data)):
   lgb_train = lgb_data.subset(train_idx.tolist()).construct()
   train_set = catboost_data.slice(train_idx.tolist())
```

How to tune hyperparameters

LightGBM and CatBoost implementations come with numerous hyperparameters that permit fine-grained control. Each library has parameter settings to:

- Specify the task objective and learning algorithm
- Design the base learners
- Apply various regularization techniques
- Handle early stopping during training
- Enable the use of GPU or parallelization on CPU

The documentation for each library details the various parameters. Since they implement variations of the same algorithms, parameters may refer to the same concept but have different names across libraries. The GitHub repository lists resources that clarify which XGBoost and LightGBM parameters have a comparable effect.

Objectives and loss functions

The libraries support several boosting algorithms, including gradient boosting for trees and linear base learners, as well as DART for LightGBM and XGBoost. LightGBM also supports the GOSS algorithm, which we described previously, as well as random forests.

The appeal of gradient boosting consists of the efficient support of arbitrary differentiable loss functions, and each library offers various options for regression, classification, and ranking tasks. In addition to the chosen

loss function, additional evaluation metrics can be used to monitor performance during training and cross-validation.

Learning parameters

Gradient boosting models typically use decision trees to capture feature interaction, and the size of individual trees is the most important tuning parameter. XGBoost and CatBoost set the `max_depth` default to 6. In contrast, LightGBM uses a default `num_leaves` value of 31, which corresponds to five levels for a balanced tree, but imposes no constraints on the number of levels. To avoid overfitting, `num_leaves` should be lower than $2^{\text{max\_depth}}$. For example, for a well-performing `max_depth` value of 7, you would set `num_leaves` to 70–80 rather than 2^7=128, or directly constrain `max_depth`.

The number of trees or boosting iterations defines the overall size of the ensemble. All libraries support `early_stopping` to abort training once the loss functions register no further improvements during a given number of iterations. As a result, it is often most efficient to set a large number of iterations and stop training based on the predictive performance on a validation set. However, keep in mind that the validation error will be biased upward due to the implied lookahead bias.

The libraries also permit the use of custom loss metrics to track train and validation performance and execute `early_stopping`. The notebook illustrates how to code the **information coefficient** (**IC**) for LightGBM and CatBoost. However, we will not rely on `early_stopping` for our experiments to avoid said bias.

Regularization

All libraries implement the regularization strategies for base learners, such as minimum values for the number of samples or the minimum information gain required for splits and leaf nodes.

They also support regularization at the ensemble level using shrinkage, which is implemented via a learning rate that constrains the contribution of new trees. It is also possible to implement an adaptive learning rate via callback functions that lower the learning rate as the training progresses, as has been successfully used in the context of neural networks. Furthermore, the gradient boosting loss function can be constrained using L1 or L2 regularization, similar to the ridge and lasso regression models, for example, by increasing the penalty for adding more trees.

The libraries also allow for the use of bagging or column subsampling to randomize tree growth for random forests and decorrelate prediction errors to reduce overall variance. The quantization of features for approximate split finding adds larger bins as an additional option to protect against overfitting.

Randomized grid search

To explore the hyperparameter space, we specify values for key parameters that we would like to test in combination. The sklearn library sup-

ports `RandomizedSearchCV` to cross-validate a subset of parameter combinations that are sampled randomly from specified distributions. We will implement a custom version that allows us to monitor performance so we can abort the search process once we're satisfied with the result, rather than specifying a set number of iterations beforehand.

To this end, we specify options for the relevant hyperparameters of each library, generate all combinations using the Cartesian product generator provided by the itertools library, and shuffle the result.

In the case of LightGBM, we focus on the learning rate, the maximum size of the trees, the randomization of the feature space during training, and the minimum number of data points required for a split. This results in the following code, where we randomly select half of the configurations:

```
learning_rate_ops = [.01, .1, .3]
max_depths = [2, 3, 5, 7]
num_leaves_opts = [2 ** i for i in max_depths]
feature_fraction_opts = [.3, .6, .95]
min_data_in_leaf_opts = [250, 500, 1000]
cv_params = list(product(learning_rate_ops,
                         num_leaves_opts,
                         feature_fraction_opts,
                         min_data_in_leaf_opts))
n_params = len(cv_params)
# randomly sample 50%
cvp = np.random.choice(list(range(n_params)),
                           size=int(n_params / 2),
                           replace=False)
cv_params_ = [cv_params[i] for i in cvp]
```

Now, we are mostly good to go: during each iteration, we create a `MultipleTimeSeriesCV` instance based on the `lookahead`, `train_period_length`, and `test_period_length` parameters, and cross-validate the selected hyperparameters accordingly over a 2-year period.

Note that we generate validation predictions for a range of ensemble sizes so that we can infer the optimal number of iterations:

```
num_iterations = [10, 25, 50, 75] + list(range(100, 501, 50))
num_boost_round = num_iterations[-1]
for lookahead, train_length, test_length in test_params:
   n_splits = int(2 * YEAR / test_length)
   cv = MultipleTimeSeriesCV(n_splits=n_splits,
                             lookahead=lookahead,
                             test_period_length=test_length,
                             train_period_length=train_length)
   for p, param_vals in enumerate(cv_params_):
       for i, (train_idx, test_idx) in enumerate(cv.split(X=outcome_data)):
           lgb_train = lgb_data.subset(train_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}
```

Please see the notebook `trading_signals_with_lightgbm_and_catboost` for additional details, including how to log results and compute and capture various metrics that we need for the evaluation of the results, to which we'll turn to next.

How to evaluate the results

Now that cross-validation of numerous configurations has produced a large number of results, we need to evaluate the predictive performance to identify the model that generates the most reliable and profitable signals for our prospective trading strategy. The notebook `evaluate_trading_signals` contains the code examples for this section.

We produced a larger number of LightGBM models because it runs an order of magnitude faster than CatBoost and will demonstrate some evaluation strategies accordingly.

Cross-validation results – LightGBM versus CatBoost

First, we compare the predictive performance of the models produced by the two libraries across all configurations in terms of their validation IC, both across the entire validation period and averaged over daily forecasts.

The following image shows that that LightGBM performs (slightly) better than CatBoost, especially for longer horizons. This is not an entirely fair comparison because we ran more configurations for LightGBM, which also, unsurprisingly, shows a wider dispersion of outcomes:

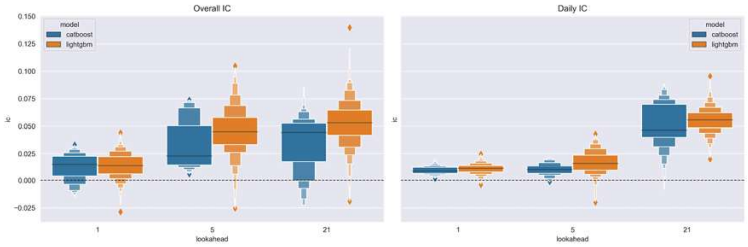

Figure 12.8: Overall and daily IC for the LightGBM and CatBoost models over three prediction horizons

Regardless, we will focus on LightGBM results; see the notebooks `trading_signals_with_lightgbm_and_catboost` and `evaluate_trading_signals` for more details on CatBoost or to run your own experiments.

In view of the substantial dispersion across model results, let's take a closer look at the best-performing parameter settings.

Best-performing parameter settings

The top-performing LightGBM models uses the following parameters for the three different prediction horizons (see the notebook for details):

Lookahead	Learning Rate	# Leaves	Feature Fraction	Min. Data in Leaf	Daily Average		Overall	
					IC	# Rounds	IC	# Rounds
1	0.3	4	95%	1,000	1.70	75	4.41	50
1	0.3	4	95%	250	1.34	250	4.36	25
1	0.3	4	95%	1,000	1.70	75	4.30	75
5	0.1	8	95%	1,000	3.95	300	10.46	300
5	0.3	4	95%	1,000	3.43	150	10.32	50
5	0.3	4	95%	1,000	3.43	150	10.24	150
21	0.1	8	60%	500	5.84	25	13.97	10
21	0.1	32	60%	250	5.89	50	11.59	10
21	0.1	4	60%	250	7.33	75	11.40	10

Note that shallow trees produce the best overall IC across the three prediction horizons. Longer training over 4.5 years also produced better results.

Hyperparameter impact – linear regression

Next, we'd like to understand if there's a systematic, statistical relationship between the hyperparameters and the outcomes across daily predictions. To this end, we will run a linear regression using the various LightGBM hyperparameter settings as dummy variables and the daily validation IC as the outcome.

The chart in *Figure 12.9* shows the coefficient estimates and their confidence intervals for 1- and 21-day forecast horizons. For the shorter horizon, a longer lookback period, a higher learning rate, and deeper trees (more leaf nodes) have a positive impact. For the longer horizon, the picture is a little less clear: shorter trees do better, but the lookback period is not significant. A higher feature sampling rate also helps. In both cases, a larger ensemble does better. Note that these results apply to this specific example only.

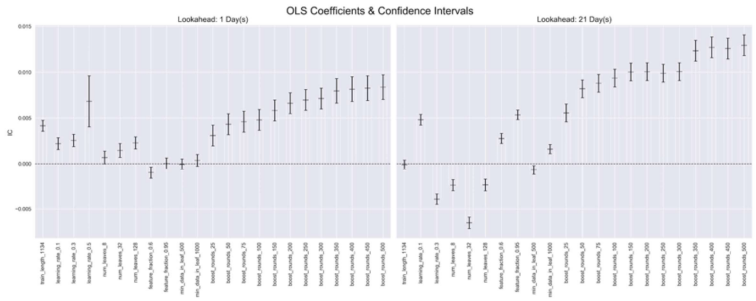

Figure 12.9: Coefficient estimates and their confidence intervals for different forecast horizons

Use IC instead of information coefficient

We average the top five models and provide the corresponding prices to Alphalens, in order to compute the mean period-wise return earned on an equal-weighted portfolio invested in the daily factor quintiles for various holding periods:

Metric	Holding Period			
	1D	5D	10D	21D
Mean Period Wise Spread (bps)	12.1654	6.9514	4.9465	4.4079
Ann. alpha	0.1759	0.0776	0.0446	0.0374
beta	0.0891	0.1516	0.1919	0.1983

We find a 12 bps spread between the top and the bottom quintile, which implies an annual alpha of 0.176 while the beta is low at 0.089 (see *Figure 12.10*):

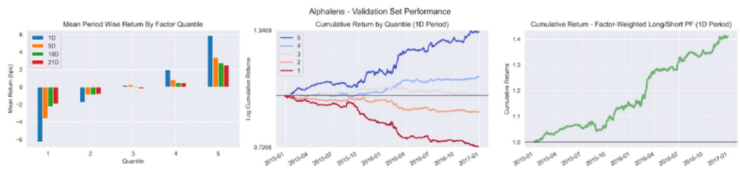

Figure 12.10: Average and cumulative returns by factor quantile

The following charts show the quarterly rolling IC for the 1-day and the 21-day forecasts over the 2-year validation period for the best-performing models:

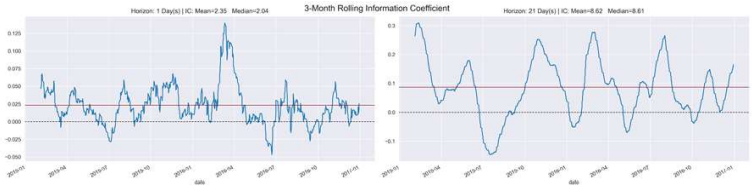

Figure 12.11: Rolling IC for 1-day and 21-day return forecasts

The average IC is 2.35 and 8.52 for the shorter and the longer horizon models, respectively, and remain positive for the large majority of days in the sample.

We'll now take a look at how to gain additional insight into how the model works before we select our models, generate predictions, define a trading strategy, and evaluate their performance.

Inside the black box – interpreting GBM results

Understanding why a model predicts a certain outcome is very important for several reasons, including trust, actionability, accountability, and debugging. Insights into the nonlinear relationship between features and the outcome uncovered by the model, as well as interactions among features, are also of value when the goal is to learn more about the underlying drivers of the phenomenon under study.

A common approach to gaining insights into the predictions made by tree ensemble methods, such as gradient boosting or random forest models, is to attribute feature importance values to each input variable. These feature importance values can be computed on an individual basis for a single prediction or globally for an entire dataset (that is, for all samples) to gain a higher-level perspective of how the model makes predictions.

The code examples for this section are in the notebook `model_interpretation`.

Feature importance

There are three primary ways to compute global feature importance values:

- **Gain**: This classic approach, introduced by Leo Breiman in 1984, uses the total reduction of loss or impurity contributed by all splits for a given feature. The motivation is largely heuristic, but it is a commonly used method to select features.
- **Split count**: This is an alternative approach that counts how often a feature is used to make a split decision, based on the selection of features for this purpose based on the resultant information gain.
- **Permutation**: This approach randomly permutes the feature values in a test set and measures how much the model's error changes, assuming that an important feature should create a large increase in the prediction error. Different permutation choices lead to alternative implementations of this basic approach.

Individualized feature importance values that compute the relevance of features for a single prediction are less common. This is because available model-agnostic explanation methods are much slower than tree-specific methods.

All gradient boosting implementations provide feature-importance scores after training as a model attribute. The LightGBM library provides two versions, as shown in the following list:

- **gain**: Contribution of a feature to reducing the loss
- **split**: The number of times the feature was used

These values are available using the trained model's `.feature_importance()` method with the corresponding `importance_type` parameter. For the best-performing LightGBM model, the results for the 20 most important features are as shown in *Figure 12.12*:

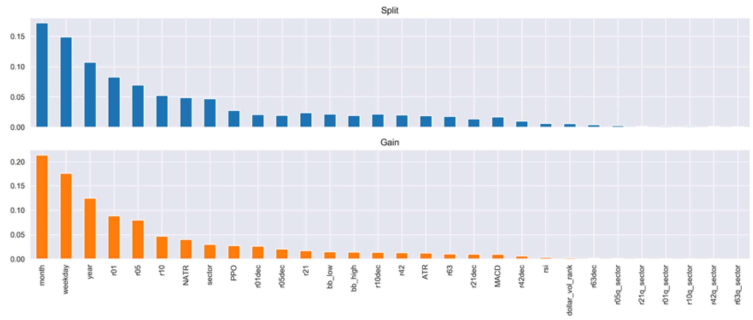

Figure 12.12: LightGBM feature importance

The time period indicators dominate, followed by the latest returns, the normalized ATR, the sector dummy, and the momentum indicator (see the notebook for implementation details).

Partial dependence plots

In addition to the summary contribution of individual features to the model's prediction, partial dependence plots visualize the relationship between the target variable and a set of features. The nonlinear nature of gradient boosting trees causes this relationship to depend on the values of all other features. Hence, we will marginalize these features out. By doing so, we can interpret the partial dependence as the expected target response.

We can visualize partial dependence only for individual features or feature pairs. The latter results in contour plots that show how combinations of feature values produce different predicted probabilities, as shown in the following code:

```
fig, axes = plot_partial_dependence(estimator=best_model,
                                    X=X,
                                    features=['return_12m', 'return_6m',
                                              'CMA', ('return_12m',
```

```
                                        'return_6m')],
                    percentiles=(0.01, 0.99),
                    n_jobs=-1,
                    n_cols=2,
                    grid_resolution=250)
```

After some additional formatting (see the companion notebook), we obtain the results shown in *Figure 12.13*:

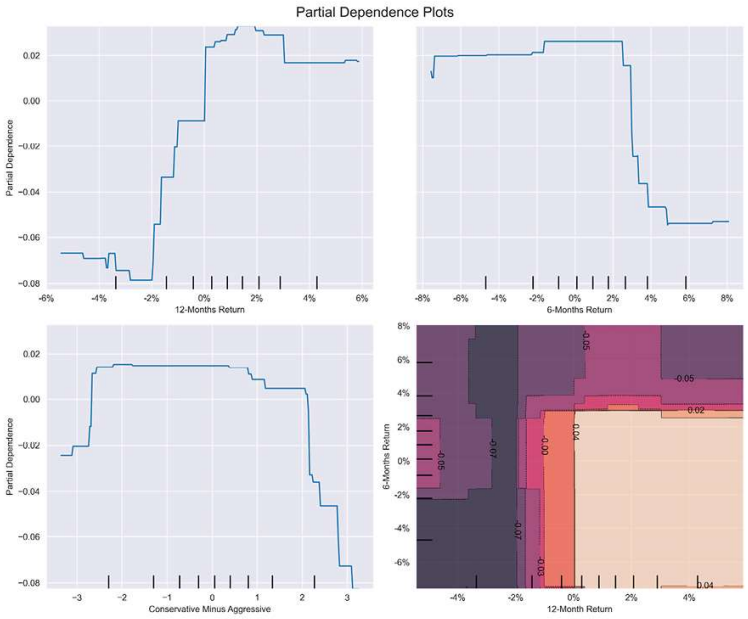

Figure 12.13: Partial dependence plots for scikit-learn GradientBoostingClassifier

The lower-right plot shows the dependence of the probability of a positive return over the next month, given the range of values for lagged 12-month and 6-month returns, after eliminating outliers at the [1%, 99%] percentiles. The `month_9` variable is a dummy variable, hence the step-function-like plot. We can also visualize the dependency in 3D, as shown in the following code:

```
targets = ['return_12m', 'return_6m']
pdp, axes = partial_dependence(estimator=gb_clf,
                               features=targets,
                               X=X_,
                               grid_resolution=100)
XX, YY = np.meshgrid(axes[0], axes[1])
Z = pdp[0].reshape(list(map(np.size, axes))).T
fig = plt.figure(figsize=(14, 8))
ax = Axes3D(fig)
surf = ax.plot_surface(XX, YY, Z,
                       rstride=1,
                       cstride=1,
                       cmap=plt.cm.BuPu,
                       edgecolor='k')
ax.set_xlabel(' '.join(targets[0].split('_')).capitalize())
ax.set_ylabel(' '.join(targets[1].split('_')).capitalize())
```

```
    ax.set_zlabel('Partial Dependence')
    ax.view_init(elev=22, azim=30)
```

This produces the following 3D plot of the partial dependence of the 1-month return direction on lagged 6-month and 12-months returns:

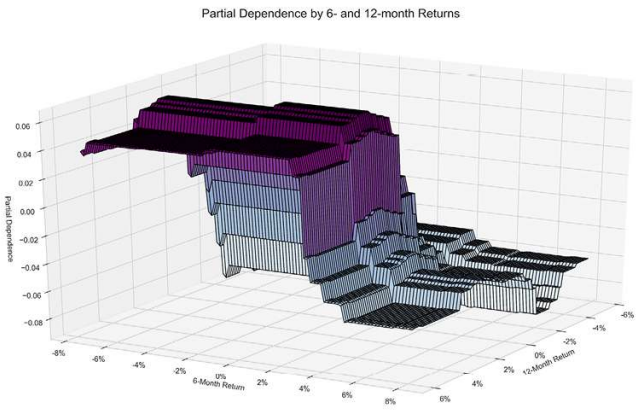

Figure 12.14: Partial dependence as a 3D plot

SHapley Additive exPlanations

At the 2017 NIPS conference, Scott Lundberg and Su-In Lee, from the University of Washington, presented a new and more accurate approach to explaining the contribution of individual features to the output of tree ensemble models called **SHapley Additive exPlanations**, or **SHAP** values.

This new algorithm departs from the observation that feature-attribution methods for tree ensembles, such as the ones we looked at earlier, are inconsistent—that is, a change in a model that increases the impact of a feature on the output can lower the importance values for this feature (see the references on GitHub for detailed illustrations of this).

SHAP values unify ideas from collaborative game theory and local explanations, and have been shown to be theoretically optimal, consistent, and locally accurate based on expectations. Most importantly, Lundberg and Lee have developed an algorithm that manages to reduce the complexity of computing these model-agnostic, additive feature-attribution methods from $O(TLDM)$ to $O(TLD^2)$, where T and M are the number of trees and features, respectively, and D and L are the maximum depth and number of leaves across the trees. This important innovation permits the explanation of predictions from previously intractable models with thousands of trees and features in a fraction of a second. An open source implementation became available in late 2017 and is compatible with XGBoost, LightGBM, CatBoost, and sklearn tree models.

Shapley values originated in game theory as a technique for assigning a value to each player in a collaborative game that reflects their contribution to the team's success. SHAP values are an adaptation of the game theory concept to tree-based models and are calculated for each feature and each sample. They measure how a feature contributes to the model out-

put for a given observation. For this reason, SHAP values provide differentiated insights into how the impact of a feature varies across samples, which is important, given the role of interaction effects in these nonlinear models.

How to summarize SHAP values by feature

To get a high-level overview of the feature importance across a number of samples, there are two ways to plot the SHAP values: a simple average across all samples that resembles the global feature-importance measures computed previously (as shown in the left-hand panel of *Figure 12.15*), or a scatterplot to display the impact of every feature for every sample (as shown in the right-hand panel of the figure). They are very straightforward to produce using a trained model from a compatible library and matching input data, as shown in the following code:

```
# load JS visualization code to notebook
shap.initjs()
# explain the model's predictions using SHAP values
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_test)
shap.summary_plot(shap_values, X_test, show=False)
```

The scatterplot sorts features by their total SHAP values across all samples and then shows how each feature impacts the model output, as measured by the SHAP value, as a function of the feature's value, represented by its color, where red represents high values and blue represents low values relative to the feature's range:

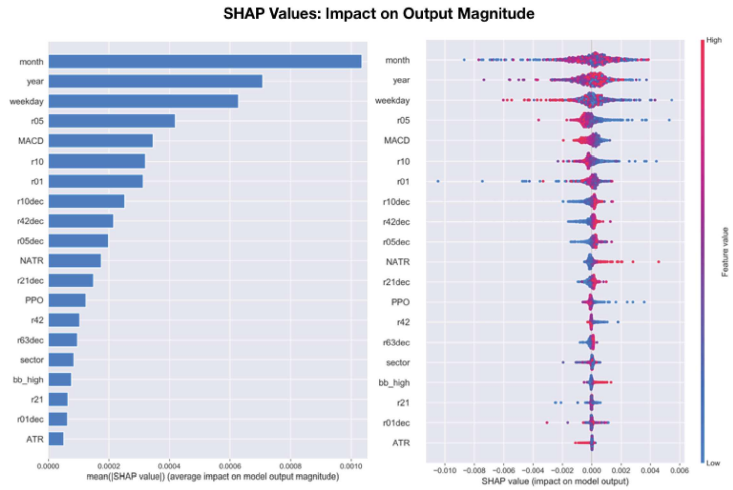

Figure 12.15: SHAP summary plots

There are some interesting differences compared to the conventional feature importance shown in *Figure 12.12*; namely, the MACD indicator turns out more important, as well as the relative return measures.

How to use force plots to explain a prediction

The force plot in the following image shows the **cumulative impact of various features and their values** on the model output, which in this case was 0.6, quite a bit higher than the base value of 0.13 (the average model output over the provided dataset). Features highlighted in red with arrows pointing to the right increase the output. The month being October is the most important feature and increases the output from 0.338 to 0.537, whereas the year being 2017 reduces the output.

Hence, we obtain a detailed breakdown of how the model arrived at a specific prediction, as shown in the following plot:

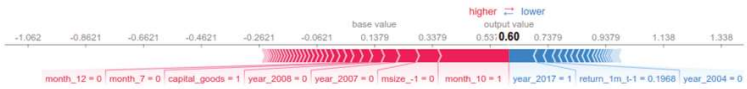

Figure 12.16: SHAP force plot

We can also compute **force plots for multiple data points** or predictions at a time and use a **clustered visualization** to gain insights into how prevalent certain influence patterns are across the dataset. The following plot shows the force plots for the first 1,000 observations rotated by 90 degrees, stacked horizontally, and ordered by the impact of different features on the outcome for the given observation.

The implementation uses hierarchical agglomerative clustering of data points on the feature SHAP values to identify these patterns, and displays the result interactively for exploratory analysis (see the notebook), as shown in the following code:

```
shap.force_plot(explainer.expected_value, shap_values[:1000,:],
                X_test.iloc[:1000])
```

This produces the following output:

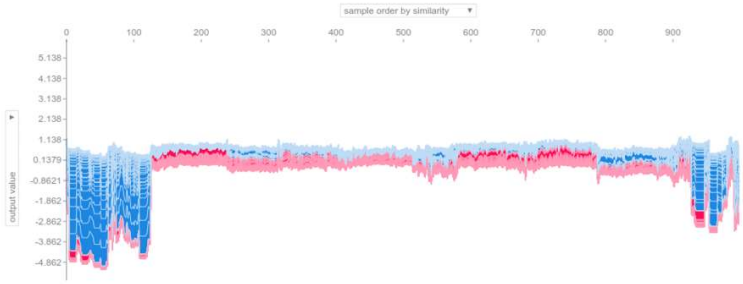

Figure 12.17: SHAP clustered force plot

How to analyze feature interaction

Lastly, SHAP values allow us to gain additional insights into the interaction effects between different features by separating these interactions from the main effects. `shap.dependence_plot` can be defined as follows:

```
shap.dependence_plot(ind='r01',
                     shap_values=shap_values,
                     features=X,
                     interaction_index='r05',
                     title='Interaction between 1- and 5-Day Returns')
```

It displays how different values for 1-month returns (on the x-axis) affect the outcome (SHAP value on the y-axis), differentiated by 3-month returns (see the following plot):

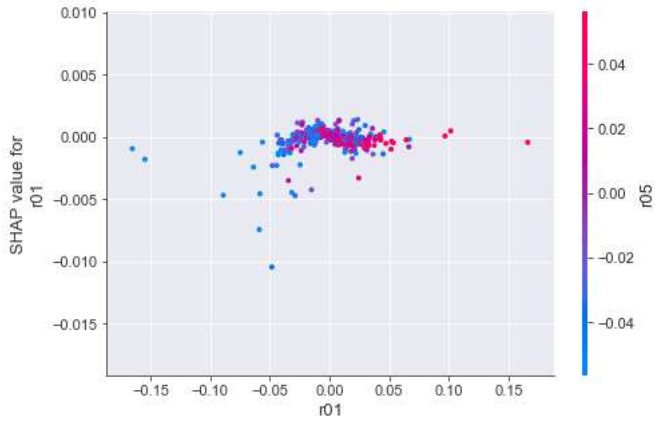

Figure 12.18: SHAP interaction plot

SHAP values provide granular feature attribution at the level of each individual prediction and enable much richer inspection of complex models through (interactive) visualization. The SHAP summary dot plot displayed earlier in this section (*Figure 12.15*) offers much more differentiated insights than a global feature-importance bar chart. Force plots of individual clustered predictions allow more detailed analysis, while SHAP dependence plots capture interaction effects and, as a result, provide more accurate and detailed results than partial dependence plots.

The limitations of SHAP values, as with any current feature-importance measure, concern the attribution of the influence of variables that are highly correlated because their similar impact can be broken down in arbitrary ways.

Backtesting a strategy based on a boosting ensemble

In this section, we'll use Zipline to evaluate the performance of a long-short strategy that enters 25 long and 25 short positions based on a daily return forecast signal. To this end, we'll select the best-performing models, generate forecasts, and design trading rules that act on these predictions.

Based on our evaluation of the cross-validation results, we'll select one or several models to generate signals for a new out-of-sample period. For this example, we'll combine predictions for the best 10 LightGBM models

to reduce variance for the 1-day forecast horizon based on its solid mean quantile spread computed by Alphalens.

We just need to obtain the parameter settings for the best-performing models and then train accordingly. The notebook `making_out_of_sample_predictions` contains the requisite code. Model training uses the hyperparameter settings of the best-performing models and data for the test period, but otherwise follows the logic used during cross-validation very closely, so we'll omit the details here.

In the notebook `backtesting_with_zipline`, we've combined the predictions of the top 10 models for the validation and test periods, as follows:

```
def load_predictions(bundle):
    predictions = (pd.read_hdf('predictions.h5', 'train/01')
                   .append(pd.read_hdf('predictions.h5', 'test/01')
                   .drop('y_test', axis=1)))
    predictions = (predictions.loc[~predictions.index.duplicated()]
                   .iloc[:, :10]
                   .mean(1)
                   .sort_index()
                   .dropna()
                  .to_frame('prediction'))
```

We'll use the custom ML factor that we introduced in *Chapter 8, The ML4T Workflow – From Model to Strategy Backtesting*, to import the predictions and make it accessible in a pipeline.

We'll execute `Pipeline` from the beginning of the validation period to the end of the test period. *Figure 12.19* shows (unsurprisingly) solid in-sample performance with annual returns of 27.3 percent, compared to 8.0 percent out-of-sample. The right panel of the image shows the cumulative returns relative to the S&P 500:

Metric	All	In-sample	Out-of-sample
Annual return	20.60%	27.30%	8.00%
Cumulative returns	75.00%	62.20%	7.90%
Annual volatility	19.40%	21.40%	14.40%
Sharpe ratio	1.06	1.24	0.61
Max drawdown	-17.60%	-17.60%	-9.80%
Sortino ratio	1.69	2.01	0.87
Skew	0.86	0.95	-0.16
Kurtosis	8.61	7.94	3.07

Daily value at risk	-2.40%	-2.60%	-1.80%
Daily turnover	115.10%	108.60%	127.30%
Alpha	0.18	0.25	0.05
Beta	0.24	0.24	0.22

The Sharpe ratio is 1.24 in-sample and 0.61 out-of-sample; the right panel shows the quarterly rolling value. Alpha is 0.25 in-sample versus 0.05 out-of-sample, with beta values of 0.24 and 0.22, respectively. The worst drawdown leads to losses of 17.59 percent in the second half of 2015:

Figure 12.19: Strategy performance—cumulative returns and rolling Sharpe ratio

Long trades are slightly more profitable than short trades, which lose on average:

Summary stats	All trades	Short trades	Long trades
Total number of round_trips	22,352	11,631	10,721
Percent profitable	50.0%	48.0%	51.0%
Winning round_trips	11,131	5,616	5,515
Losing round_trips	11,023	5,935	5,088
Even round_trips	198	80	118

Lessons learned and next steps

Overall, we can see that despite using only market data in a highly liquid environment, the gradient boosting models manage to deliver predictions that are significantly better than random guesses. Clearly, profits are anything but guaranteed, not least since we made very generous assumptions regarding transaction costs (note the high turnover).

However, there are several ways to improve on this basic framework, that is, by varying parameters from more general and strategic to more

specific and tactical aspects, such as:

1. Try a different investment universe (for example, fewer liquid stocks or other assets).
2. Be creative about adding complementary data sources.
3. Engineer more sophisticated features.
4. Vary the experiment setup using, for example, longer or shorter holding and lookback periods.
5. Come up with more interesting trading rules and use several rather than a single ML signal.

Hopefully, these suggestions inspire you to build on the template we laid out and come up with an effective ML-driven trading strategy!

Boosting for an intraday strategy

We introduced **high-frequency trading** (**HFT**) in *Chapter 1*, *Machine Learning for Trading – From Idea to Execution*, as a key trend that accelerated the adoption of algorithmic strategies. There is no objective definition of HFT that pins down the properties of the activities it encompasses, including holding periods, order types (for example, passive versus aggressive), and strategies (momentum or reversion, directional or liquidity provision, and so on). However, most of the more technical treatments of HFT seem to agree that the data driving HFT activity tends to be the most granular available. Typically, this would be microstructure data directly from the exchanges such as the NASDAQ ITCH data that we introduced in *Chapter 2*, *Market and Fundamental Data – Sources and Techniques*, to demonstrate how it details every order placed, every execution, and every cancelation, and thus permits the reconstruction of the full limit order book, at least for equities and except for certain hidden orders.

The application of ML to HFT includes the optimization of trade execution both on official exchanges and in dark pools. ML can also be used to generate trading signals, as we will show in this section; see also Kearns and Nevmyvaka (2013) for additional details and examples of how ML can add value in the HFT context.

This section uses the **AlgoSeek NASDAQ 100 dataset** from the Consolidated Feed produced by the Securities Information Processor. The data includes information on the National Best Bid and Offer quotes and trade prices at **minute bar frequency**. It also contains some features on the price dynamic, such as the number of trades at the bid or ask price, or those following positive and negative price moves at the tick level (see *Chapter 2*, *Market and Fundamental Data – Sources and Techniques*, for additional background and the download and preprocessing instructions in the data directory in the GitHub repository).

We'll first describe how we can engineer features for this dataset, then train a gradient boosting model to predict the volume-weighted average price for the next minute, and then evaluate the quality of the resulting trading signals.

Engineering features for high-frequency data

The dataset that AlgoSeek generously made available for this book contains over 50 variables on 100 tickers for any given day at minute frequency for the period 2013-2017. The data also covers pre-market and after-hours trading, but we'll limit this example to official market hours to the 390 minutes from 9:30 a.m. to 4:00 p.m. to somewhat restrict the size of the data, as well as to avoid having to deal with periods of irregular trading activity. See the notebook `intraday_features` for the code examples in this section.

We'll select 12 variables with over 51 million observations as raw material to create features for an ML model. This will aim predict the 1-min forward return for the volume-weighted average price:

```
MultiIndex: 51242505 entries, ('AAL', Timestamp('2014-12-22 09:30:00')) to ('YHOO', Timestamp('201
Data columns (total 12 columns):
 #   Column  Non-Null Count     Dtype
---  ------  --------------     -----
 0   first   51242500 non-null  float64
 1   high    51242500 non-null  float64
 2   low     51242500 non-null  float64
 3   last    51242500 non-null  float64
 4   price   49242369 non-null  float64
 5   volume  51242505 non-null  int64
 6   up      51242505 non-null  int64
 7   down    51242505 non-null  int64
 8   rup     51242505 non-null  int64
 9   rdown   51242505 non-null  int64
 10  atask   51242505 non-null  int64
 11  atbid   51242505 non-null  int64
dtypes: float64(5), int64(7)
memory usage: 6.1+ GB
```

Due to the large memory footprint of the data, we only create 20 simple features, namely:

- The lagged returns for each of the last 10 minutes.
- The number of shares traded with upticks and downticks during a bar, divided by the total number of shares.
- The number of shares traded where the trade price is the same (repeated) following and upticks or downticks during a bar, divided by the total number of shares.
- The difference between the number of shares traded at the ask versus the bid price, divided by total volume during the bar.
- Several technical indicators, including the Balance of Power, the Commodity Channel Index, and the Stochastic RSI (see the *Appendix*, *Alpha Factor Library*, for details).

We'll make sure that we shift the data to avoid lookahead bias, as exemplified by the computation of the Money Flow Index, which uses the TA-Lib implementation:

```
data['MFI'] = (by_ticker
               .apply(lambda x: talib.MFI(x.high,
                                          x.low,
                                          x['last'],
                                          x.volume,
                                          timeperiod=14)
                      .shift()))
```

The following graph shows a standalone evaluation of the individual features' predictive content using their rank correlation with the 1-minute forward returns. It reveals that the recent lagged returns are presumably the most informative variables:

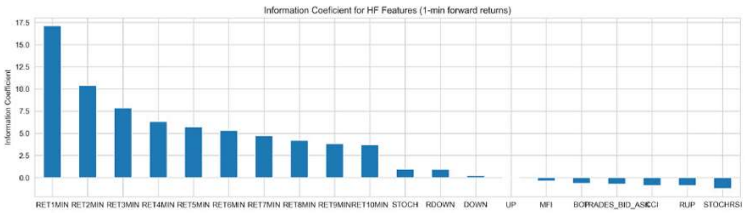

Figure 12.20: Information coefficient for high-frequency features

We can now proceed to train a gradient boosting model using these features.

Minute-frequency signals with LightGBM

To generate predictive signals for our HFT strategy, we'll train a LightGBM boosting model to predict the 1-min forward returns. The model receives 12 months of minute data during training the model and generates out-of-sample forecasts for the subsequent 21 trading days. We'll repeat this for 24 train-test splits to cover the last 2 years of our 5-year sample.

The training process follows the preceding LightGBM example closely; see the notebook `intraday_model` for the implementation details.

One key difference is the adaptation of the custom `MultipleTimeSeriesCV` to minute frequency; we'll be referencing the `date_time` level of `MultiIndex` (see notebook for implementation). We compute the lengths of the train and test periods based on 390 observations per ticker and day as follows:

```
DAY = 390   # minutes; 6.5 hrs (9:30 - 15:59)
MONTH = 21  # trading days
n_splits = 24
cv = MultipleTimeSeriesCV(n_splits=n_splits,
                          lookahead=1,
                          test_period_length=MONTH * DAY,
                          train_period_length=12 * MONTH * DAY,
                          date_idx='date_time')
```

The large data size significantly drives up training time, so we use default settings but set the number to trees per ensemble to 250. We track the IC on the test set using the following `ic_lgbm()` custom metric definition that we pass to the model's `.train()` method.

The custom metric receives the model prediction and the binary training dataset, which we can use to compute any metric of interest; note that we set `is_higher_better` to `True` since the model minimizes loss functions by default (see the LightGBM documentation for additional information):

```
def ic_lgbm(preds, train_data):
    """Custom IC eval metric for lightgbm"""
    is_higher_better = True
    return 'ic', spearmanr(preds, train_data.get_label())[0], is_higher_better
model = lgb.train(params=params,
                  train_set=lgb_train,
                  valid_sets=[lgb_train, lgb_test],
                  feval=ic_lgbm,
                  num_boost_round=num_boost_round,
                  early_stopping_rounds=50,
                  verbose_eval=50)
```

At 250 iterations, the validation IC is still improving for most folds, so our results are not optimal, but training already takes several hours this way. Let's now take a look at the predictive content of the signals generated by our model.

Evaluating the trading signal quality

Now, we would like to know how accurate the model's out-of-sample predictions are, and whether they could be the basis for a profitable trading strategy.

First, we compute the IC, both for all predictions and on a daily basis, as follows:

```
ic = spearmanr(cv_preds.y_test, cv_preds.y_pred)[0]
by_day = cv_preds.groupby(cv_preds.index.get_level_values('date_time').date)
ic_by_day = by_day.apply(lambda x: spearmanr(x.y_test, x.y_pred)[0])
daily_ic_mean = ic_by_day.mean()
daily_ic_median = ic_by_day.median()
```

For the 2 years of rolling out-of-sample tests, we obtain a statistically significant, positive 1.90. On a daily basis, the mean IC is 1.98 and the median IC equals 1.91.

These results clearly suggest that the predictions contain meaningful information about the direction and size of short-term price movements that we could use for a trading strategy.

Next, we calculate the average and cumulative forward returns for each decile of the predictions:

```
dates = cv_preds.index.get_level_values('date_time').date
cv_preds['decile'] = (cv_preds.groupby(dates, group_keys=False)
min_ret_by_decile = cv_preds.groupby(['date_time', 'decile']).y_test.mean()
                      .apply(lambda x: pd.qcut(x.y_pred, q=10))))
cumulative_ret_by_decile = (min_ret_by_decile
                            .unstack('decile')
                            .add(1)
                            .cumprod()
                            .sub(1))
```

Figure 12.21 displays the results. The left panel shows the average 1-min return per decile and indicates an average spread of 0.5 basis points per minute. The right panel shows the cumulative return of an equal-weighted portfolio invested in each decile, suggesting that—before transaction costs—a long-short strategy appears attractive:

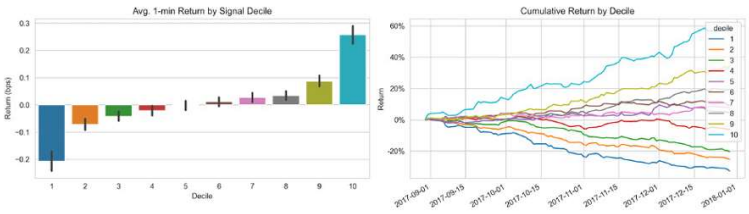

Figure 12.21: Average 1-min returns and cumulative returns by decile

The backtest with minute data is quite time-consuming, so we've omitted this step; however, feel free to experiment with Zipline or backtrader to evaluate this strategy under more realistic assumptions regarding transaction costs or using proper risk controls.

Summary

In this chapter, we explored the gradient boosting algorithm, which is used to build ensembles in a sequential manner, adding a shallow decision tree that only uses a very small number of features to improve on the predictions that have been made. We saw how gradient boosting trees can be very flexibly applied to a broad range of loss functions, as well as offer many opportunities to tune the model to a given dataset and learning task.

Recent implementations have greatly facilitated the use of gradient boosting. They've done this by accelerating the training process and offering more consistent and detailed insights into the importance of features and the drivers of individual predictions.

Finally, we developed a simple trading strategy driven by an ensemble of gradient boosting models that was actually profitable, at least before significant trading costs. We also saw how to use gradient boosting with high-frequency data.

In the next chapter, we will turn to Bayesian approaches to machine learning.