On the other hand, random forests also have a few disadvantages:

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

Summary

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

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

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

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

11
Gradient Boosting Machines

In the previous chapter, we learned about how random forests improve the predictions made by individual decision trees by combining them into an ensemble that reduces the high variance of individual trees. Random forests use bagging, which is short for bootstrap aggregation, to introduce random elements into the process of growing individual trees.

More specifically, bagging draws samples from the data with replacement so that each tree is trained on a different but equal-sized random subset of the data (with some observations repeating). Random forests also randomly select a subset of the features so that both the rows and the columns of the data that are used to train each tree are random versions of the original data. The ensemble then generates predictions by averaging over the outputs of the individual trees.

Individual 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 or diversify the errors made by the individual trees so that the ensemble is much less susceptible to overfitting, has lower variance, and generalizes better to new data.

In this chapter, we will explore boosting, an alternative **machine learning** (**ML**) algorithm for ensembles of decision trees that often produces even better results. The key difference is that boosting modifies the data that is used to train each tree based on the cumulative errors made by the model before adding the new tree. In contrast to random forests which train many trees independently from each other using different versions 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.

In this chapter, we will see how boosting has evolved into one of the most successful ML algorithms over the last three decades. At the time of writing, it has come to dominate machine learning competitions for structured data (as opposed to high-dimensional images or speech, for example, where the relationship between the input and output is more complex, and deep learning excels at). More specifically, in this chapter we will cover the following topics:

- How boosting works, and how it compares to bagging
- How boosting has evolved from adaptive to gradient boosting
- How to use and tune AdaBoost and gradient boosting models with sklearn
- How state-of-the-art GBM implementations dramatically speed up computation
- How to prevent overfitting of gradient boosting models
- How to build, tune, and evaluate gradient boosting models on large datasets using `xgboost`, `lightgbm`, and `catboost`
- How to interpret and gain insights from gradient boosting models

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 (as described in *Elements of Statistical Learning* by Trevor Hastie, et al.; see GitHub for links to references). Like bagging, it is a general method or metamethod that can be applied to many statistical learning models.

The motivation for the development of boosting was to find a method to combine the outputs of many *weak* models (a predictor is called weak when it performs just slightly better than random guessing) into a more powerful, that is, boosted joint prediction. In general, boosting learns an additive hypothesis, H_M, of a form similar to linear regression. However, now each of the $m = 1,\ldots, M$ elements of the summation is a weak base learner, called h_t that itself requires training. The following formula summarizes the approach:

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

As discussed in the last chapter, bagging trains base learners on different random samples of the training data. Boosting, in contrast, proceeds sequentially by training the base learners on data that is repeatedly modified to reflect the cumulative learning results. 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. About another five years later, this algorithm was extended to arbitrary objective functions when Leo Breiman (who invented random forests) connected the approach to gradient descent, and Jerome Friedman came up with gradient boosting in 1999. Numerous optimized implementations, such as XGBoost, LightGBM, and CatBoost, have emerged in recent years and firmly established gradient boosting as the go-to solution for structured data.

In the following sections, we will briefly introduce AdaBoost and then focus on the gradient boosting model, as well as several state-of-the-art implementations of this very powerful and flexible algorithm.

The AdaBoost algorithm

AdaBoost was the first boosting 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.

The algorithm had a very significant impact on ML 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.

AdaBoost is a significant departure from bagging, which builds ensembles on very deep trees to reduce bias. AdaBoost, in contrast, 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 equal-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 predict 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:

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

    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 it 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.

On the other hand, 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.

AdaBoost with sklearn

As part of its ensemble module, sklearn provides an `AdaBoostClassifier` implementation that supports two or more classes. The code examples for this section are in the notebook `gbm_baseline` that 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. The complexity of the `base_estimator` is a key tuning parameter because it depends on the nature of the data. As demonstrated in the previous chapter, 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 the `learning_rate` determines the contribution of each weak learner, as shown in the following code. By default, weak learners are decision tree stumps:

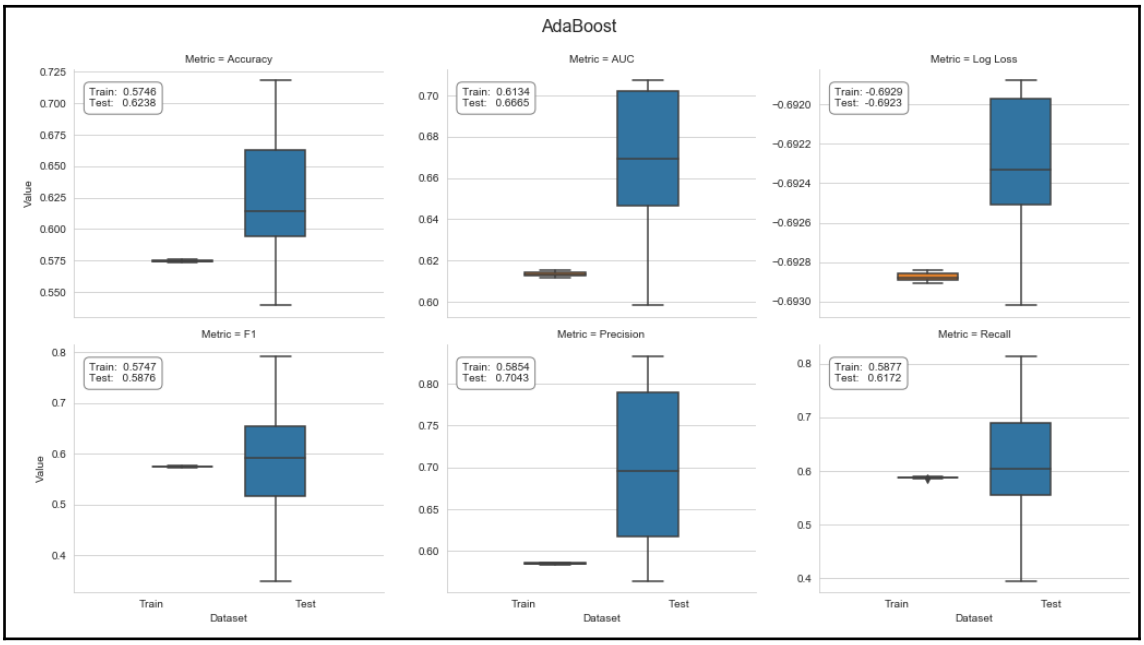
```
ada_clf = AdaBoostClassifier(base_estimator=base_estimator,
                             n_estimators=200,
                             learning_rate=1.0,
                             algorithm='SAMME.R',
                             random_state=42)
```

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

We will cross-validate the AdaBoost ensemble using a custom 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 result shows a weighted test accuracy of 0.62, a test **AUC** of **0.6665**, and a negative log loss of **-0.6923**, as well as a test **F1** score of **0.5876**, as shown in the following screenshot:



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

Gradient boosting machines

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

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

This interpretation of the AdaBoost algorithm was only discovered several years after its publication. It views AdaBoost as a coordinate-based gradient descent algorithm that minimizes a particular loss function, namely exponential loss.

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. The flexibility to customize this general method to many specific prediction tasks is essential to boosting's popularity.

The main idea behind the resulting **Gradient Boosting Machines** (**GBM**) algorithm is the training of 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.

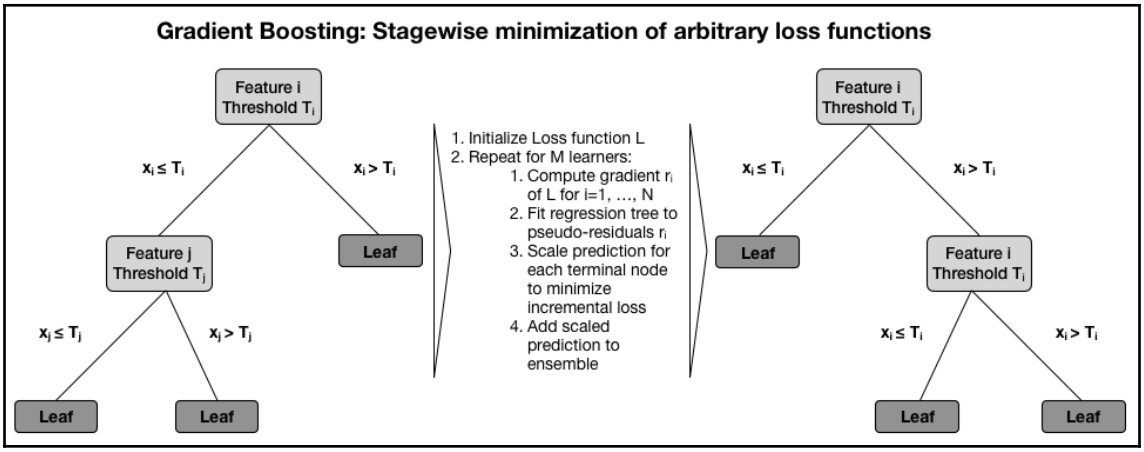
In short, the algorithm successively fits weak learners h_m, such as decision trees, to the negative gradient of the loss function that is evaluated for the current ensemble, as shown in the following formula:

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

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 constant prediction for each terminal node that minimizes the incremental loss that results from adding this new learner to the ensemble.

This differs from standalone decision trees and random forests, where the prediction depends on the outcome values of the training samples present in the relevant terminal or leaf node: 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 since 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:



Gradient boosting trees have demonstrated state-of-the-art performance on many classification, regression, and ranking benchmarks. They are probably the most popular ensemble learning algorithm both as a standalone predictor in a diverse set of machine learning 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. The flexibility of this algorithm requires the careful management of the risk of overfitting by tuning hyperparameters that constrain the model's inherent 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

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 complexity for 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 requiring:

- A minimum number of samples to either split a node or accept it as a terminal node, or
- A minimum improvement in node quality as measured by the purity or entropy or mean square error, in the case of regression.

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*. 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 so that for a large ensemble, the training error can potentially become very small, increasing the risk of overfitting and poor performance on unseen data. Cross-validation is the best approach to find the optimal ensemble size that minimizes the generalization error because it depends on the application and the available data.

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 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, just to 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 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 uses sampling without replacement at each iteration to grow the next tree on a subset of the training samples. The benefit is both lower computational effort and often better accuracy, but subsampling should be combined with shrinkage.

As you can see, the number of hyperparameters keeps increasing, driving up the number of potential combinations, which in turn increases the risk of false positives when choosing the best model from a large number of parameter trials on a limited amount of training data. The best approach is to proceed sequentially and select parameter values individually or using 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 that we previously introduced, in addition to those that we are familiar with from looking at standalone decision tree models. The notebook `gbm_tuning_with_sklearn` contains the code examples for this section.

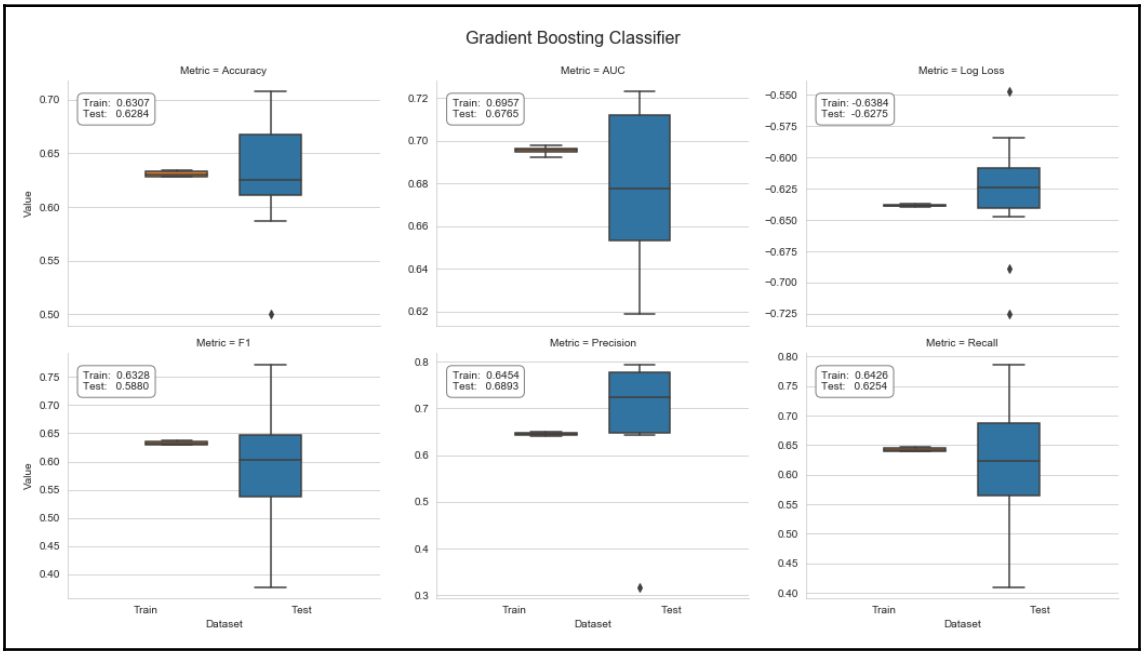
The available loss functions include the exponential loss that leads to the AdaBoost algorithm and the deviance that corresponds to the logistic regression for probabilistic outputs. The `friedman_mse` node quality measure is a variation on the mean squared error that includes an improvement score (see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

```
gb_clf = GradientBoostingClassifier(loss='deviance',                #
deviance = logistic reg; exponential: AdaBoost
                                    learning_rate=0.1,              #
shrinks the contribution of each tree
                                    n_estimators=100,               #
number of boosting stages
                                    subsample=1.0,                  #
fraction of samples used t fit base learners
                                    criterion='friedman_mse',       #
measures the quality of a split
                                    min_samples_split=2,
                                    min_samples_leaf=1,
                                    min_weight_fraction_leaf=0.0,   # min.
fraction of sum of weights
                                    max_depth=3,                    # opt
value depends on interaction
                                    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 will parse and plot the result to find a slight improvement—using default parameter values—over the `AdaBoostClassifier`, as shown in the following screenshot:



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 that when defined will result in a total of 2^4 x 3^2 x 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 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:

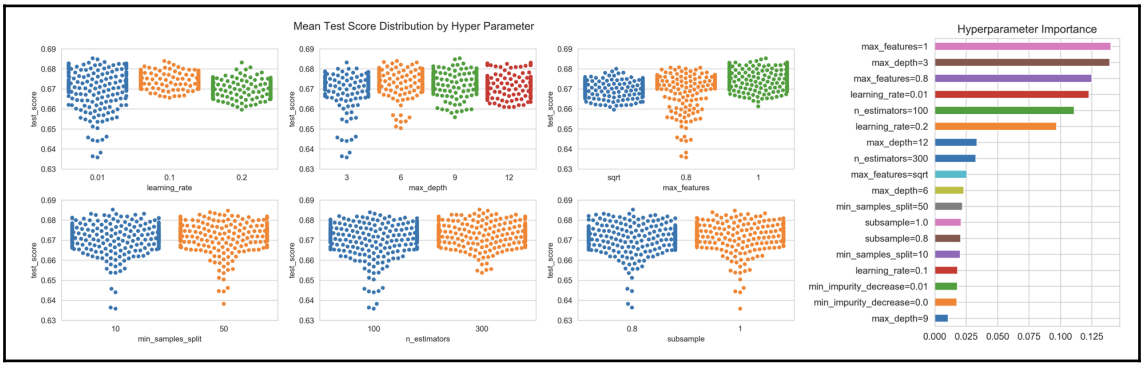
```
pd.Series(gridsearch_result.best_params_)
learning_rate              0.01
max_depth                  9.00
max_features               1.00
min_impurity_decrease      0.01
min_samples_split         10.00
n_estimators             300.00
subsample                  0.80

gridsearch_result.best_score_
0.6853
```

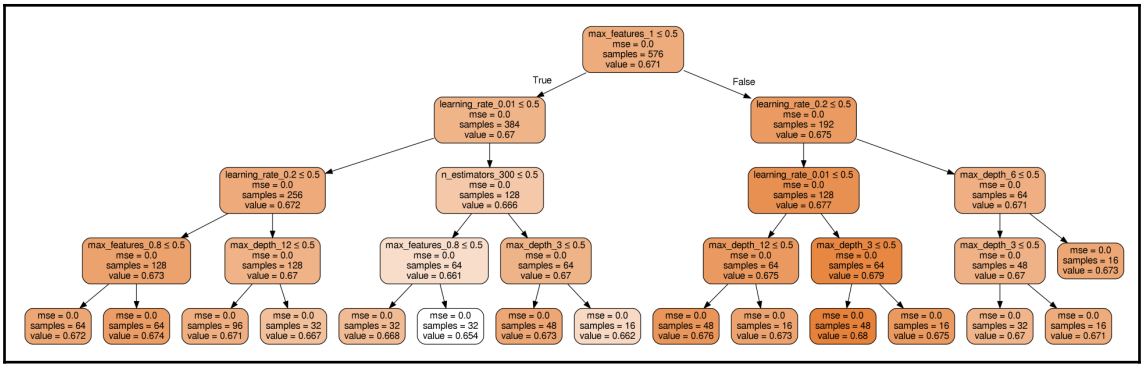
Parameter impact on test scores

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 left-hand panel of the below chart show the distribution of AUC test scores for all parameter 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. Other parameters are compatible with high scores for all settings use in the experiment:



We will now explore how hyperparameter settings jointly affect the mean cross-validation score. To gain insight into how parameter settings interact, we can train a `DecisionTreeRegressor` with the mean test score as the outcome and the parameter settings, encoded as categorical variables 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` over three led to the best results, as shown in the following diagram:



The bar chart in the right-hand panel of the first chart in this section displays the influence of the hyperparameter settings in producing different outcomes, measured by their feature importance for a decision tree that is 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 six 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 `0.6622` using the following code:

```
best_model = gridsearch_result.best_estimator_
preds= best_model.predict(test_feature_data)
roc_auc_score(y_true=test_target, y_score=preds)
0.6622
```

The downside of the sklearn gradient boosting implementation is the limited speed of computation 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.

Fast scalable GBM implementations

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 (extreme gradient boosting), started in 2014 by Tianqi Chen at the University of Washington
- LightGBM, first released in January 2017, by Microsoft
- CatBoost, first released in April 2017 by Yandex

These innovations address specific challenges of training a gradient boosting model (see this chapter's `README` on GitHub for detailed references). 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 drive performance

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

To expand the ensemble by a tree, the training algorithm incrementally minimizes the prediction error with respect to the negative gradient of the ensemble's loss function, similar to a conventional gradient descent optimizer. Hence, the computational cost during training is proportional to the time it takes to evaluate the impact of potential split points for each feature on the decision tree's fit to the current gradient.

Second-order loss function approximation

The most important algorithmic innovations lower the cost of evaluating the loss function by using approximations that rely on second-order derivatives, resembling Newton's method to find stationary points. As a result, scoring potential splits during greedy tree expansion is faster relative to using the full loss function.

As mentioned previously, a gradient boosting model is trained in an incremental manner with the goal of minimizing the combination of the prediction error and the regularization penalty for the ensemble H_M. Denoting the prediction of the outcome y_i by the ensemble after step m as $\hat{y}_i^{(m)}$, l 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_m aims to minimize the following objective:

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

The regularization penalty helps to avoid overfitting by favoring the selection of a model that uses simple and 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:

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

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:

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

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:

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

The XGBoost algorithm was the first open-source algorithm to leverage this approximation of the loss function to compute the optimal leave 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 impurity measures we saw in the previous chapter, but applicable to arbitrary loss functions (see the references on GitHub for the detailed derivation).

Simplified split-finding algorithms

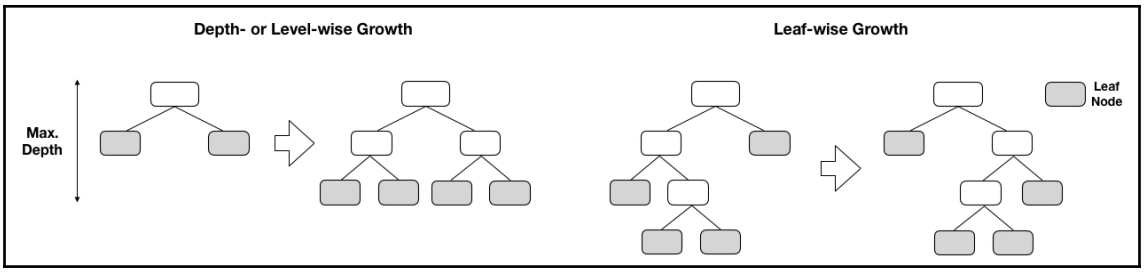
The gradient boosting implementation by sklearn finds the optimal split that enumerates all options for continuous features. This precise greedy algorithm is computationally very demanding because it must first sort the data by feature values before scoring the potentially very large number of split options and making a decision. This approach faces 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 because only a single split needs to be stored for each bin. XGBoost introduced a quantile sketch algorithm that was also able to divide 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 also 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.

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, i.e., 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 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 the two approaches:



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 relevant documentation).

The speedups depend on the library and the type of the data, and 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 trees

In 2015, Rashmi and Gilad-Bachrach proposed a new model to train gradient boosting trees that aimed 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 regarding 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 added earlier in the process.

The new algorithms employ dropouts which have been successfully used for learning more accurate deep neural networks where dropouts mute a random fraction of the neural connections during the learning process. 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, to mute a random share of the features. Random forests and stochastic gradient boosting also drop out a random subset of features.

DART 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 towards 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, and 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 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 optimized computation in several respects to enable multithreading by keeping data in memory in compressed column blocks, where each column is sorted by the corresponding feature value. XGBoost computes this input data layout once before training and reuses it throughout to amortize the additional up-front cost. The search for split statistics over columns becomes a linear scan when using quantiles that can be done in parallel with easy support for 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.

How to use XGBoost, LightGBM, and CatBoost

XGBoost, LightGBM, and CatBoost offer interfaces for multiple languages, including Python, and have both a sklearn interface that is compatible with other sklearn features, such as `GridSearchCV` and their own methods to train and predict gradient boosting models. The `gbm_baseline.ipynb` notebook illustrates the use of the sklearn interface for each implementation. The library methods are often better documented and are also easy to use, so we'll use them to illustrate the use of these models.

The process entails the creation of library-specific data formats, the tuning of various hyperparameters, and the evaluation of results that we will describe in the following sections. The accompanying notebook contains the `gbm_tuning.py`, `gbm_utils.py` and, `gbm_params.py` files that jointly provide the following functionalities and have produced the corresponding results.

How to create binary data formats

All libraries have their own data format to precompute feature statistics to accelerate the search for split points, as described previously. These can also be persisted to accelerate the start of subsequent training.

The following code constructs binary train and validation datasets for each model to be used with the `OneStepTimeSeriesSplit`:

```
cat_cols = ['year', 'month', 'age', 'msize', 'sector']
data = {}
for fold, (train_idx, test_idx) in enumerate(kfold.split(features)):
    print(fold, end=' ', flush=True)
    if model == 'xgboost':
        data[fold] = {'train': xgb.DMatrix(label=target.iloc[train_idx],
                                           data=features.iloc[train_idx],
                                           nthread=-1),                    #
use avail. threads
                      'valid': xgb.DMatrix(label=target.iloc[test_idx],
                                           data=features.iloc[test_idx],
                                           nthread=-1)}
    elif model == 'lightgbm':
        train = lgb.Dataset(label=target.iloc[train_idx],
                            data=features.iloc[train_idx],
                            categorical_feature=cat_cols,
                            free_raw_data=False)

        # align validation set histograms with training set
        valid = train.create_valid(label=target.iloc[test_idx],
                                   data=features.iloc[test_idx])

        data[fold] = {'train': train.construct(),
                      'valid': valid.construct()}

    elif model == 'catboost':
        # get categorical feature indices
        cat_cols_idx = [features.columns.get_loc(c) for c in cat_cols]
        data[fold] = {'train': Pool(label=target.iloc[train_idx],
                                    data=features.iloc[train_idx],
                                    cat_features=cat_cols_idx),

                      'valid': Pool(label=target.iloc[test_idx],
                                    data=features.iloc[test_idx],
                                    cat_features=cat_cols_idx)}
```

The available options vary slightly:

- `xgboost` allows the use of all available threads
- `lightgbm` explicitly aligns the quantiles that are created for the validation set with the training set
- The `catboost` implementation needs feature columns identified using indices rather than labels

How to tune hyperparameters

The numerous hyperparameters are listed in `gbm_params.py`. Each library has parameter settings to:

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

The documentation for each library details the various parameters that may refer to the same concept, but which have different names across libraries. The GitHub repository contains links to a site that highlights the corresponding parameters for `xgboost` and `lightgbm`.

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^{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 usually best to set a large number of iterations and stop training based on the predictive performance on a validation set.

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 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 regularized using *L1* or *L2*, regularization similar to the Ridge and Lasso linear regression models by modifying $\Omega(h_m)$ or by increasing the penalty γ for adding more trees, as described previously.

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 supports `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 leverage early stopping while monitoring the current best-performing combinations so we can abort the search process once satisfied with the result rather than specifying a set number of iterations beforehand.

To this end, we specify a parameter grid according to each library's parameters as before, generate all combinations using the built-in Cartesian `product` generator provided by the `itertools` library, and randomly `shuffle` the result. In the case of LightGBM, we automatically set `max_depth` as a function of the current `num_leaves` value, as shown in the following code:

```
param_grid = dict(
        # common options
        learning_rate=[.01, .1, .3],
        colsample_bytree=[.8, 1],  # except catboost

        # lightgbm
        num_leaves=[2 ** i for i in range(9, 14)],
        boosting=['gbdt', 'dart'],
        min_gain_to_split=[0, 1, 5],  # not supported on GPU

all_params = list(product(*param_grid.values()))
n_models = len(all_params) # max number of models to cross-validate
shuffle(all_params)
```

We then execute cross-validation as follows:

```
GBM = 'lightgbm'
for test_param in all_params:
    cv_params = get_params(GBM)
    cv_params.update(dict(zip(param_grid.keys(), test_param)))
    if GBM == 'lightgbm':
        cv_params['max_depth'] =
int(ceil(np.log2(cv_params['num_leaves'])))
    results[n] = run_cv(test_params=cv_params,
                        data=datasets,
                        n_splits=n_splits,
                        gb_machine=GBM)
```

The `run_cv` function implements cross-validation for all three libraries. For the `light_gbm` example, the process looks as follows:

```
def run_cv(test_params, data, n_splits=10):
    """Train-Validate with early stopping"""
    result = []
    cols = ['rounds', 'train', 'valid']
    for fold in range(n_splits):
        train = data[fold]['train']
        valid = data[fold]['valid']

        scores = {}
        model = lgb.train(params=test_params,
```

```
                            train_set=train,
                            valid_sets=[train, valid],
                            valid_names=['train', 'valid'],
                            num_boost_round=250,
                            early_stopping_rounds=25,
                            verbose_eval=50,
                            evals_result=scores)

        result.append([model.current_iteration(),
                       scores['train']['auc'][-1],
                       scores['valid']['auc'][-1]])

    return pd.DataFrame(result, columns=cols)
```

The `train()` method also produces validation scores that are stored in the `scores` dictionary. When early stopping takes effect, the last iteration is also the best score. See the full implementation on GitHub for additional details.

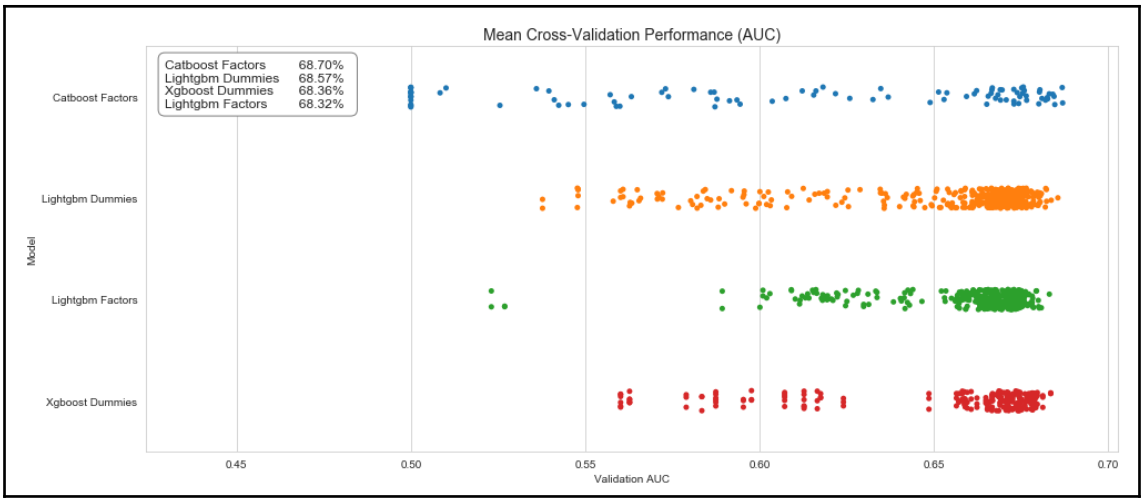
How to evaluate the results

Using a GPU, we can train a model in a few minutes and evaluate several hundred parameter combinations in a matter of hours, which would take many days using the sklearn implementation. For the LightGBM model, we explore both a factor version that uses the libraries' ability to handle categorical variables and a dummy version that uses one-hot encoding.

The results are available in the `model_tuning.h5` HDF5 store. The model evaluation code samples are in the `eval_results.ipynb` notebook.

Cross-validation results across models

When comparing average cross-validation AUC across the four test runs with the three libraries, we find that CatBoost produces a slightly higher AUC score for the top-performing model, while also producing the widest dispersion of outcomes, as shown in the following graph:



The top-performing CatBoost model uses the following parameters (see notebook for detail):

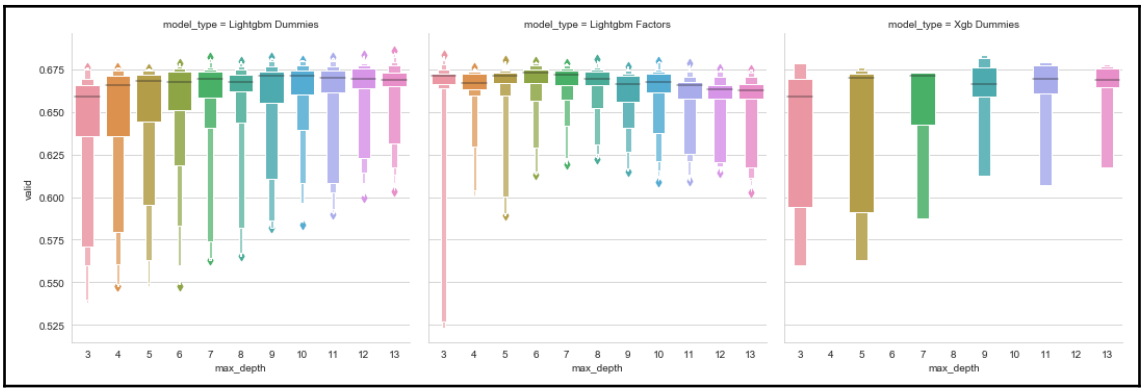
- `max_depth` of 12 and `max_bin` of 128
- `max_ctr_complexity` of 2, which limits the number of combinations of categorical features
- `one_hot_max_size` of 2, which excludes binary features from the assignment of numerical variables
- `random_strength` different from 0 to randomize the evaluation of splits

Training is a bit slower compared to LightGBM and XGBoost (all use the GPU) at an average of 230 seconds per model.

A more detailed look at the top-performing models for the LightGBM and XGBoost models shows that the LightGBM Factors model achieves nearly as good a performance as the other two models with much lower model complexity. It only consists on average of 41 trees up to three levels deep with no more than eight leaves each, while also using regularization in the form of `min_gain_to_split`. It overfits significantly less on the training set, with a train AUC only slightly above the validation AUC. It also trains much faster, taking only 18 seconds per model because of its lower complexity. In practice, this model would be preferable since it is more likely to produce good out-of-sample performance. The details are shown in the following table:

	LightGBM dummies	**XGBoost dummies**	**LightGBM factors**
Validation AUC	68.57%	68.36%	68.32%
`Train AUC`	82.35%	79.81%	72.12%
`learning_rate`	0.1	0.1	0.3
`max_depth`	13	9	3
`num_leaves`	8192		8
`colsample_bytree`	0.8	1	1
`min_gain_to_split`	0	1	0
Rounds	44.42	59.17	41.00
Time	86.55	85.37	18.78

The following plot shows the effect of different `max_depth` settings on the validation score for the LightGBM and XGBoost models: shallower trees produce a wider range of outcomes and need to be combined with appropriate learning rates and regularization settings to produce the strong result shown in the preceding table:



Instead of a `DecisionTreeRegressor` as shown previously, we can also use linear regression to evaluate the statistical significance of different features concerning the validation AUC score. For the LightGBM Dummy model, where the regression explains 68% of the variation in outcomes, we find that only the `min_gain_to_split` regularization parameter was not significant, as shown in the following screenshot:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   valid   R-squared:                       0.687
Model:                             OLS   Adj. R-squared:                  0.673
Method:                  Least Squares   F-statistic:                     26.94
Date:                 Wed, 24 Oct 2018   Prob (F-statistic):           7.92e-55
Time:                         14:03:45   Log-Likelihood:                1018.7
No. Observations:                  396   AIC:                            -2001.
Df Residuals:                      378   BIC:                            -1930.
Df Model:                           17
Covariance Type:                  HC3
===============================================================================================
                                  coef    std err          z      P>|z|      [0.025      0.975]
-----------------------------------------------------------------------------------------------
const                          0.6145      0.005    127.970      0.000       0.605       0.624
boosting_gbtree                0.0056      0.002      2.866      0.004       0.002       0.009
learning_rate_0.1              0.0501      0.003     18.977      0.000       0.045       0.055
learning_rate_0.3              0.0516      0.003     19.150      0.000       0.046       0.057
max_depth_4                    0.0060      0.005      1.094      0.274      -0.005       0.017
max_depth_5                    0.0096      0.005      1.823      0.068      -0.001       0.020
max_depth_6                    0.0153      0.005      3.024      0.002       0.005       0.025
max_depth_7                    0.0194      0.005      3.753      0.000       0.009       0.030
max_depth_8                    0.0196      0.005      3.733      0.000       0.009       0.030
max_depth_9                    0.0266      0.005      5.176      0.000       0.017       0.037
max_depth_10                   0.0307      0.005      5.954      0.000       0.021       0.041
max_depth_11                   0.0285      0.005      5.484      0.000       0.018       0.039
max_depth_12                   0.0312      0.005      6.178      0.000       0.021       0.041
max_depth_13                   0.0320      0.005      6.218      0.000       0.022       0.042
colsample_bytree_0.8          -0.0112      0.003     -4.143      0.000      -0.017      -0.006
colsample_bytree_1.0          -0.0278      0.003     -8.388      0.000      -0.034      -0.021
min_gain_to_split_1           -0.0009      0.003     -0.307      0.759      -0.006       0.005
min_gain_to_split_5           -0.0016      0.002     -0.726      0.468      -0.006       0.003
==============================================================================
Omnibus:                       11.763   Durbin-Watson:                   0.856
Prob(Omnibus):                  0.003   Jarque-Bera (JB):               11.104
Skew:                          -0.361   Prob(JB):                      0.00388
Kurtosis:                       2.609   Cond. No.                         17.1
==============================================================================

Warnings:
[1] Standard Errors are heteroscedasticity robust (HC3)
```

In practice, gaining deeper insights into how the models arrive at predictions is extremely important, in particular for investment strategies where decision makers often require plausible explanations.

How to interpret 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 on how the model makes predictions.

Feature importance

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

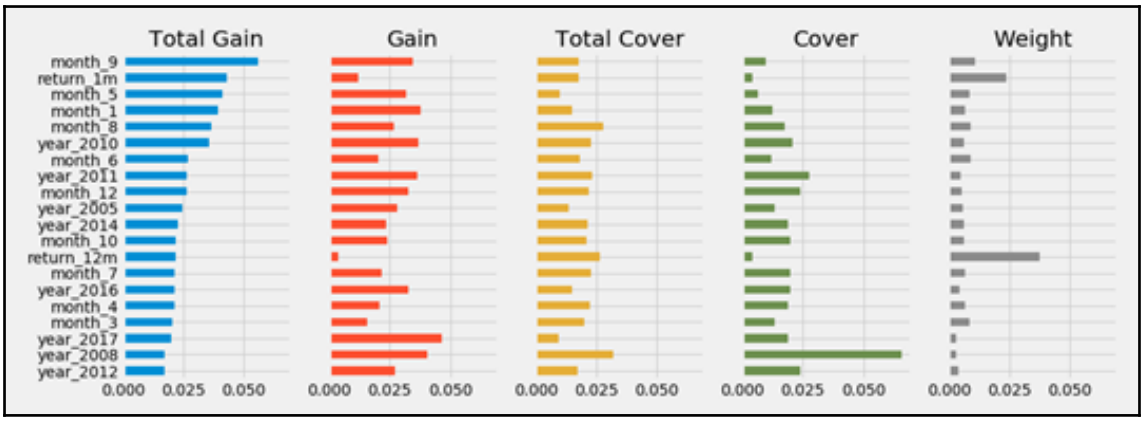
- **Gain**: This classic approach introduced by Leo Breiman in 1984 uses the total reduction of loss or impurity contributed by all splits for a given feature. The motivation is largely heuristic, but it is a commonly used method to select features.
- **Split count**: This is an alternative approach 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 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 XGBoost library provides five versions, as shown in the following list:

- `total_gain` and `gain` as its average per split
- `total_cover` as the number of samples per split when a feature was used
- `weight` as the split count from preceding values

These values are available using the trained model's `.get_score()` method with the corresponding `importance_type` parameter. For the best performing XGBoost model, the results are as follows (the *total* measures have a correlation of 0.8, as do `cover` and `total_cover`):



While the indicators for different months and years dominate, the most recent 1 month return is the second-most important feature from a `total_gain` perspective, and is used frequently according to the `weight` measure, but produces low average gains as it is applied to relatively few instances on average (see the notebook for implementation details).

Partial dependence plots

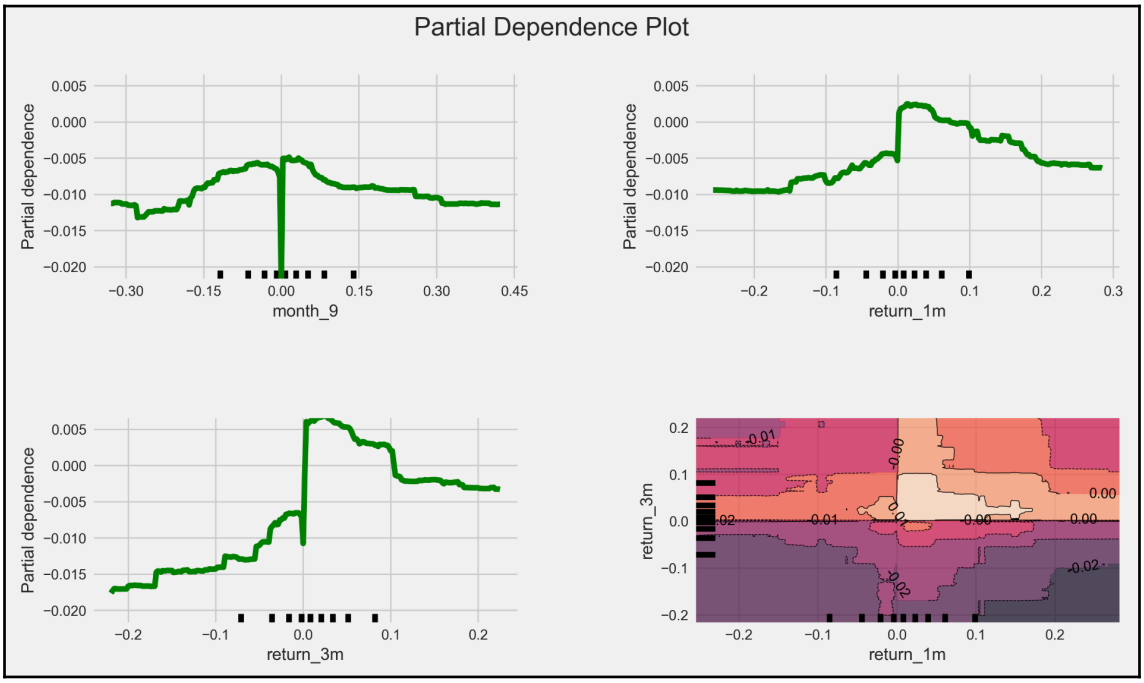
In addition to the summary contribution of individual features to the model's prediction, partial dependence plots visualize the relationship between the target variable and a set of features. The nonlinear nature of gradient boosting trees causes this relationship to depends on the values of all other 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(gbrt=gb_clf,
                                    X=X_dummies_clean,
                                    features=['month_9', 'return_1m',
'return_3m', ('return_1m', 'return_3m')],
                                    feature_names=['month_9','return_1m',
```

```
    'return_3m'],
                                                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 following plot:



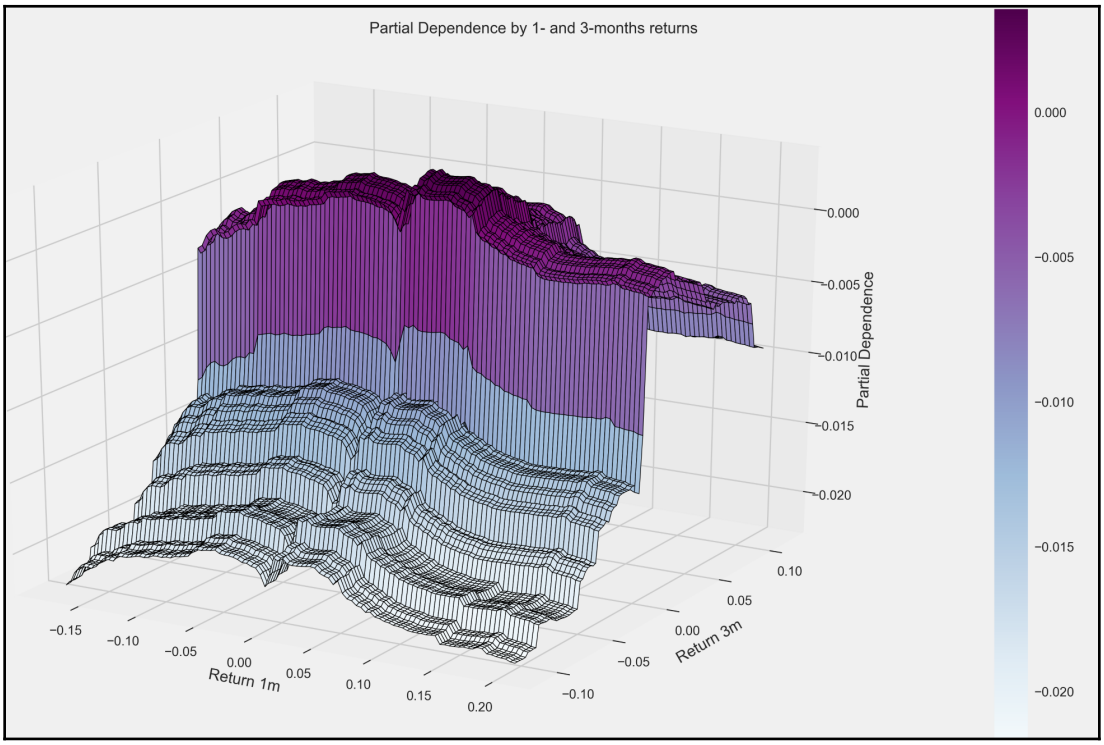
The lower-right plot shows the dependence of the probability of a positive return over the next month given the range of values for lagged 1-month and 3-month returns after eliminating outliers at the [1%, 99%] percentiles. The `month_9` variable is a dummy variable, hence the step-function-like plot.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

```
targets = ['return_1m', 'return_3m']
target_feature = [X_dummies_clean.columns.get_loc(t) for t in targets]
pdp, axes = partial_dependence(gb_clf,
                               target_feature,
                               X=X_dummies_clean,
                               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 1-month and 3-months returns:



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 $O(TLD^M)$ 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 output 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 the following screenshot), or a scatter graph to display the impact of every feature for every sample (as shown in the right-hand panel of the following screenshot). They are very straightforward to produce using a trained model of a compatible library and matching input data, as shown in the following code:

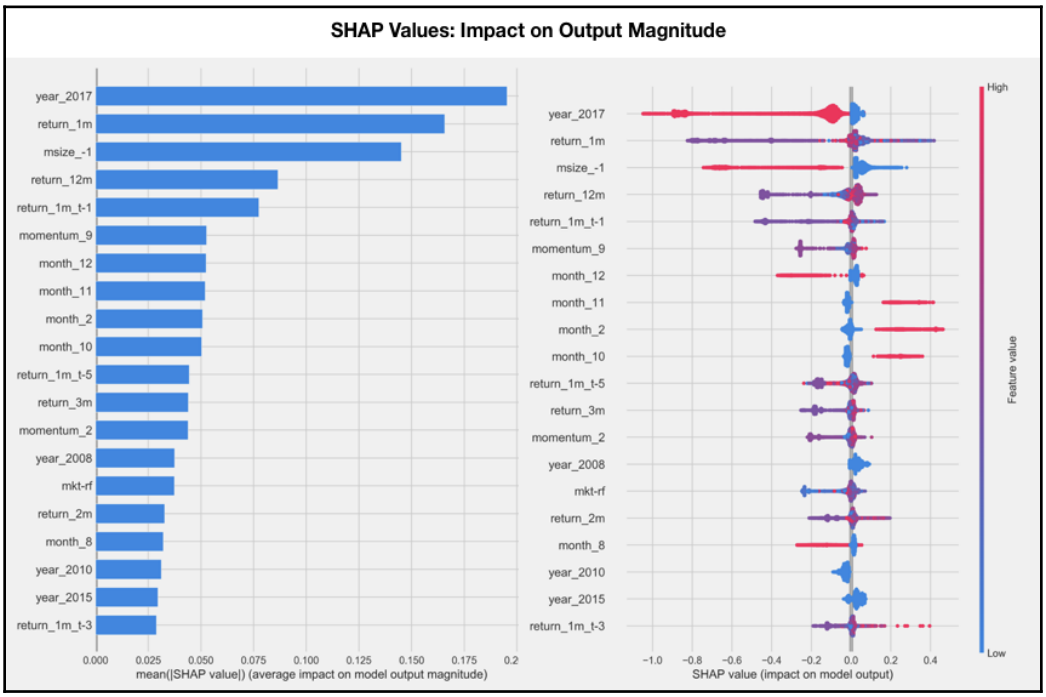
```
# load JS visualization code to notebook
```

```
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)
```

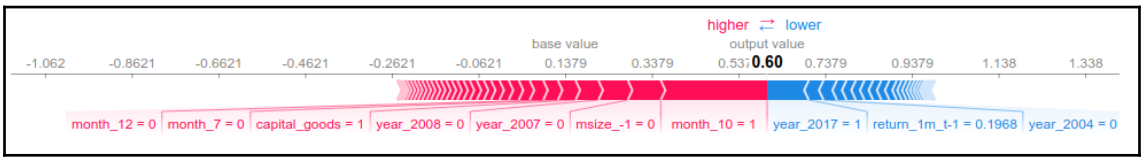
The scatter plot on the right of the following screenshot sorts features by their total SHAP values across all samples, and then shows how each feature impacts the model output as measured by the SHAP value as a function of the feature's value, represented by its color, where red represents high and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



How to use force plots to explain a prediction

The following force plot 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 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.

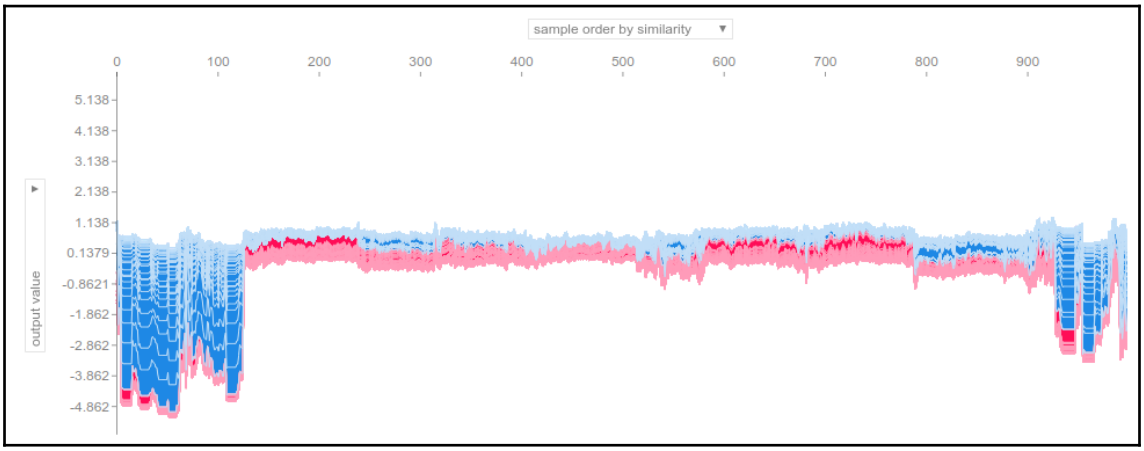
Hence, we obtain a detailed breakdown of how the model arrived at a specific prediction, as shown in the following image:



We can also compute force plots for numerous 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:

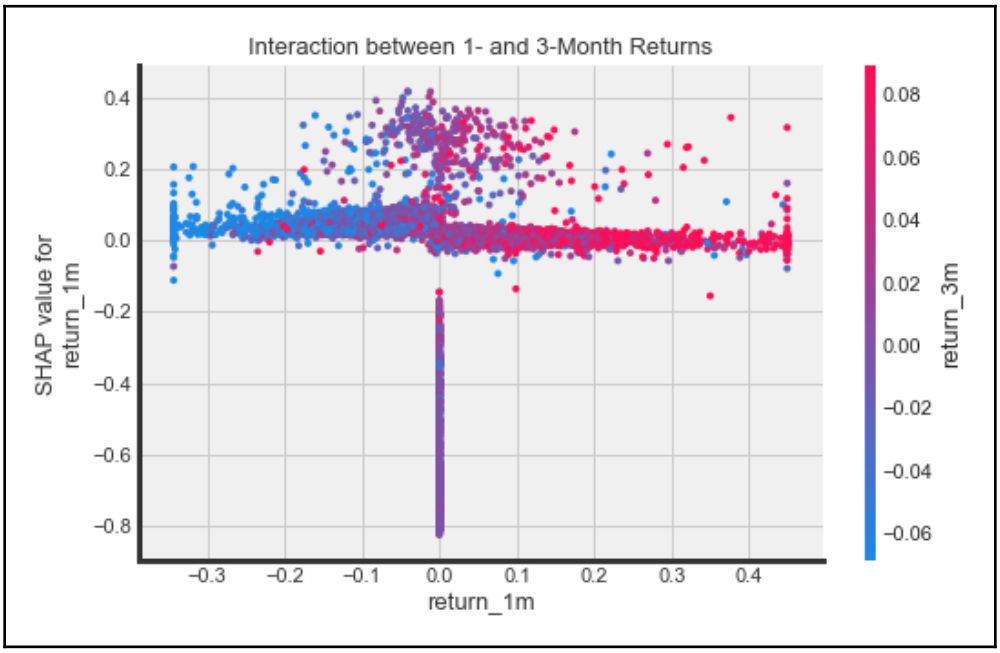


How to analyze feature interaction

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

```
shap.dependence_plot("return_1m", shap_values, X_test, interaction_index=2,
title='Interaction between 1- and 3-Month 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:



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 scatterplot displayed at the beginning of this section offers much more differentiated insights than a global feature-importance bar chart. Force plots of individual clustered predictions allow for 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 could be broken down in arbitrary ways.

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 and offer many opportunities to tune the model to a given dataset and learning task.

Recent implementations have greatly facilitated the use of gradient boosting by accelerating the training process and offering more consistent and detailed insights into the importance of features and the drivers of individual predictions. In the next chapter, we will turn to Bayesian approaches to ML.