Overview

After reading this chapter, you will understand the concept of gradient boosting, a fundamental idea underlying the XGBoost package. You will then get experience training XGBoost models on synthetic data, learning about early stopping as well as several XGBoost hyperparameters along the way. In addition to using a similar method for growing trees as we’ve done previously (by setting max\_depth), you’ll also discover a new way of growing trees that is offered by XGBoost: loss-guided tree growing. After learning about XGBoost, you’ll then be introduced to a new and powerful way of explaining model predictions, called SHAP (SHapley Additive exPlanations). You will see how SHAP values can be used to provide individualized explanations for model predictions from any dataset, not just the training data, and also understand the additive property of SHAP values.

Introduction

As we saw in the previous chapter, decision trees and ensemble models based on them provide powerful methods for creating machine learning models. While random forests have been around for decades, recent work on a different kind of tree ensemble, gradient boosted trees, has resulted in state-of-the-art models that have come to dominate the landscape of predictive modeling with tabular data, or data that is organized into a structured table, similar to the case study data. The two main packages used by machine learning data scientists today to create the most accurate predictive models with tabular data are XGBoost and LightGBM. In this chapter we’ll get familiar with XGBoost using a synthetic dataset, then apply it to the case study data in the activity.

Perhaps some of the best motivation to use XGBoost comes from the paper describing this machine learning system, in the context of Kaggle, a popular online forum for machine learning competitions:

Among the 29 challenge winning solutions published at Kaggle’s blog during 2015, 17 solutions used XGBoost. Among these solutions, eight solely used XGBoost to train the model, while most others combined XGBoost with neural nets in ensembles. For comparison, the second most popular method, deep neural nets, was used in 11 solutions.

* Chen and Guestrin, 2016 (<https://dl.acm.org/doi/abs/10.1145/2939672.2939785>)

As we’ll see, XGBoost ties together a few of the different ideas we’ve discussed so far, including decision trees and ensemble modeling as well as gradient descent.

In addition to more performant models, recent machine learning research has yielded more detailed ways to explain the predictions of models. Rather than relying on interpretations that only represent the model training set in aggregate, such as logistic regression coefficients or feature importances of a random forest, a new package called SHAP allows us to interpret model predictions individually, and for any dataset we want, such as validation or testing data. This can be very helpful in helping us as data scientists, as well as our business partners, understand the workings of a model at a granular level and for new data.

Gradient Boosting and XGBoost

What is Boosting?

Boosting is a procedure for creating ensembles of many machine learning models, or estimators, similar to the bagging concept which underlies the random forest model. Like bagging, while boosting can be used with any kind of machine learning model, it is commonly used to build ensembles of decision trees. A key difference from bagging is that in boosting, each new estimator added to the ensemble depends on all the estimators added before it. Because the boosting procedure proceeds in stages, it is also called stagewise additive modeling. The difference between bagging and boosting can be visualized as in Figure 6.1.

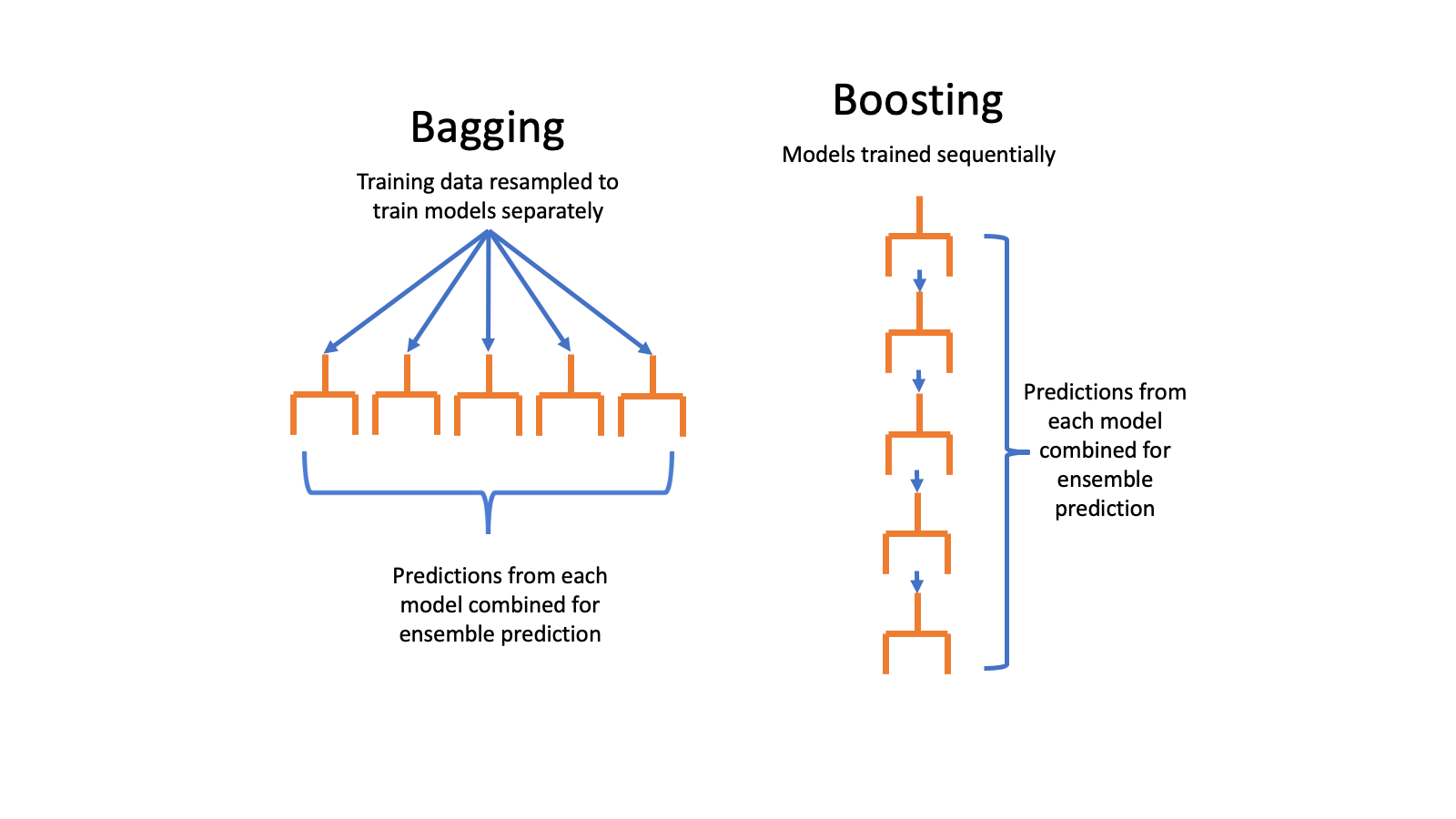


Figure 6.1: Bagging versus boosting

While bagging trains many estimators using different random samples of the training data, boosting trains new estimators using information about which samples were incorrectly classified by the previous estimators in the ensemble. By focusing new estimators on these samples, the goal is that the overall ensemble will have better performance across the whole training data set. AdaBoost, a precursor to XGBoost, accomplished this goal by giving more weight to incorrectly classified samples as new estimators in the ensemble are trained.

Gradient Boosting and XGBoost

XGBoost is a modeling procedure and python package that is one of the most popular machine learning methods in use today, due to its superior performance in many domains, from business to the natural sciences. XGBoost has also proven to be one of the most successful tools in machine learning competitions. We will not discuss all the details of how XGBoost is implemented, but rather get a high-level idea of how it works and look at some of the most important hyperparameters. For further details, the interested reader should refer to the publication “XGBoost: A Scalable Tree Boosting System” by Tianqi Chen and Carlos Guestrin (<https://dl.acm.org/doi/abs/10.1145/2939672.2939785>).

The XGBoost implementation of the gradient boosting procedure is a stagewise additive model similar to AdaBoost. However, instead of directly giving more weight to misclassified samples during model training, XGBoost uses a procedure similar in nature to gradient descent. Recall from Chapter 4 that optimization with gradient descent uses information about the derivative of a loss function (another name for the cost function), for example to update the estimated coefficients for a logistic regression. The derivative of the loss function contains information about which direction and how much to adjust the coefficient estimates at each iteration of the procedure, to reduce the error in prediction.

XGBoost applies the gradient descent idea to stagewise additive modeling, using information about the gradient (another word for derivative) of a loss function to train new decision trees to add to the ensemble. In fact, XGBoost takes things a step further than gradient descent as described in Chapter 4, and uses information about both the first and second derivatives. The approach of training decision trees using error gradients is an alternative to the node impurity idea introduced in Chapter 5. Conceptually, XGBoost trains new trees with the goal of moving the ensemble prediction in the direction of decreasing error. How big of a step to take in that direction is controlled by the learning\_rate hyperparameter, analogous to the learning\_rate in Exercise 16.

At this point, we should have enough intuition about how XGBoost works to start getting our hands dirty and using it. In order to illustrate XGBoost, we’ll create a synthetic data set for binary classification, with scikit-learn’s make\_classification function. This data set will have 5000 samples and 40 features. The rest of the options here control how challenging of a classification task this will be, and you should consult the scikit-learn documentation to better understand them. Of particular interest is that we’ll have multiple clusters (n\_clusters\_per\_class) meaning there will be several regions of points in multidimensional feature space that belong to a certain class, similar to the cluster shown in the last chapter in Figure 5.8. A tree-based model should be able to identify these clusters. Also, we are specifying that there are only 3 informative features out of the total 40 (n\_informative), as well as 2 redundant features (n\_redundant) which will contain the same information as the informative ones. So all told, only 5 out of the 40 features should be useful in making predictions, and of those all the information is encoded in 3 of them.

If you want to follow along with the examples in this chapter on your computer, please refer to the jupyter notebook at (Insert URL) for the preliminary step of importing packages.

from sklearn.datasets import make\_classification

X, y = make\_classification(

n\_samples=5000, n\_features=40, n\_informative=3,

n\_redundant=2, n\_repeated=0, n\_classes=2,

n\_clusters\_per\_class=3, weights=None, flip\_y=0.05,

class\_sep=0.1, hypercube=True, shift=0.0,

scale=1.0, shuffle=True, random\_state=2)

Note that the class fraction of the response variable y is about 50%:

y.mean()

should output

0.4986

Instead of using cross-validation, in this chapter we will split this synthetic data set just once into a training and validation set. However, the concepts we introduce here could be extended to the cross-validation scenario. We’ll split these synthetic data into 80% for training and 20% for validation. In a real-world data problem, we would also want to have a test set reserved for later use in evaluating the final model, but we’ll forego this here.

from sklearn.model\_selection import train\_test\_split

X\_train, X\_val, y\_train, y\_val = \

train\_test\_split(X, y, test\_size=0.2, random\_state=24)

Now that we’ve prepared the data for modeling, we need to instantiate an object of the class XGBClassifier. Note that we will now be using the XGBoost package and not scikit-learn, to develop a predictive model. However, XGBoost has an API (application programming interface) that was designed to be similar to that of scikit-learn, so using this class should be intuitive. The XGBClassifier class can be used to create a model object with fit and predict methods and other familiar functionality, and we can specify model hyperparameters when instantiating the class. We’ll specify just a few hyperparameters here, which we’ve already discussed: n\_estimators is the number of boosting rounds to use for the model, in other words the number of stages for the stagewise additive modeling procedure, objective is the loss function that will be used to calculate gradients, and learning\_rate controls how much each new estimator adds to the ensemble, or in essence how far of a step to take to decrease prediction error. The remaining hyperparameters are related to how much output we want to see during model training (verbosity) and the soon-to-be-deprecated option label\_encoder that XGBoost developers recommend setting to False.

xgb\_model\_1 = xgb.XGBClassifier(

n\_estimators=1000,

verbosity=1,

use\_label\_encoder=False,

objective='binary:logistic',

learning\_rate=0.3)

The hyperparameter values we’ve indicated specify that:

* We will have 1000 estimators, or boosting rounds. We’ll discuss more shortly about how many rounds are needed; the default value is 100.
* We are familiar with the objective function (also known as cost function) for binary logistic regression from Chapter 4. XGBoost also offers a wide variety of objective functions for a range of tasks including classification and regression.
* The learning rate is set to 0.3, which is the default. Different values can be explored via hyperparameter search procedures which we’ll demonstrate.

Now that we have a model object and some training data, we are ready to fit the model. This looks similar to how it did in scikit-learn:

%%time

xgb\_model\_1.fit(X\_train, y\_train, eval\_metric="auc", verbose=True)

Here we are tracking how long the fitting procedure takes using the %%time “cell magic” in the Jupyter notebook. We need to supply the features X\_train and the response variable y\_train of the training data. We also supply an eval\_metric and set the verbosity, which we’ll explain shortly. Executing this cell should give output similar to this:

CPU times: user 52.5 s, sys: 986 ms, total: 53.4 s

Wall time: 17.5 s

Out[7]:

XGBClassifier(base\_score=0.5, booster='gbtree', colsample\_bylevel=1,

colsample\_bynode=1, colsample\_bytree=1, gamma=0, gpu\_id=-1,

importance\_type='gain', interaction\_constraints='',

learning\_rate=0.3, max\_delta\_step=0, max\_depth=6,

min\_child\_weight=1, missing=nan, monotone\_constraints='()',

n\_estimators=1000, n\_jobs=4, num\_parallel\_tree=1, random\_state=0,

reg\_alpha=0, reg\_lambda=1, scale\_pos\_weight=1, subsample=1,

tree\_method='exact', use\_label\_encoder=False,

validate\_parameters=1, verbosity=1)

The output tells us that this cell took 17.5 seconds to execute, called the “Wall time” or the elapsed time on a clock that might be on your wall. The CPU times are longer than this because XGBoost efficiently uses multiple processors simultaneously. XGBoost also prints out all the hyperparameters, including the ones we set and the others that were left at their defaults.

Now, to examine the performance of this fitted model, we’ll evaluate the area under the ROC curve on the validation set. First we need obtain the predicted probabilities:

val\_set\_pred\_proba = xgb\_model\_1.predict\_proba(X\_val)[:,1]

from sklearn.metrics import roc\_auc\_score

roc\_auc\_score(y\_val, val\_set\_pred\_proba)

The output of this cell should be:

0.7773798710782294

Indicating the ROC AUC of about 0.78. This will be our baseline of model performance, using nearly default options for XGBoost.

XGBoost hyperparameters

Early Stopping

When training ensembles of decision trees with XGBoost, there are many options available for reducing overfitting and leveraging the bias-variance tradeoff. Early stopping is a simple one of these and can help provide an automated answer to the question “how many boosting rounds are needed?”. It’s important to note that early stopping relies on having a separate validation set of data, aside from the training set. However this validation set will actually be used during the model training process, so it does not qualify as “unseen” data that was held out from model training, similar to how we used validation sets in cross-validation to select model hyperparameters in Chapter 4.

When XGBoost is training successive decision trees to reduce error on the training set, it’s possible that adding more and more trees to the ensemble will provide increasingly better fits to the training data, but start to cause lower performance on held-out data. In order to avoid this, we can use a validation set, also called an evaluation set or eval\_set by XGBoost. The evaluation set will be supplied as a list of tuples of features and their corresponding response variables. Whichever tuple comes last in this list will be the one that is used for early stopping. We want this to be the validation set, since the training data will be used to fit the model and can’t provide an estimate of out-of-sample generalization:

eval\_set = [(X\_train, y\_train), (X\_val, y\_val)]

Now we can fit the model again, but this time we supply the eval\_set keyword argument with the evaluation set we just created. At this point, the eval\_metric of auc becomes important. This means that after each boosting round, before training another decision tree, the area under the ROC curve will be evaluated on all the data sets supplied with eval\_set. Since we’ll indicate verbosity=True, we’ll get output printed below the cell with the ROC AUC for both the training set and the validation set. This provides a nice live look at how model performance changes on the training and validation data as more boosting rounds are trained.

Since, in predictive modeling, we’re primarily interested in how a model performs on new and unseen data, we would like to stop training additional boosting rounds when it becomes clear that they are not improving model performance on out-of-sample data. The early\_stopping\_rounds=30 argument indicates that once 30 boosting rounds have been completed without any additional improvement in the AUC ROC on the validation set, XGBoost should stop model training. After model training is complete, the final fitted model will only have as many ensemble members as needed to get the highest model performance on the validation set. This means that the last 30 members of the ensemble will be discarded, since they didn’t provide any increase in validation set performance. Let’s now fit this model and watch the progress:

%%time

xgb\_model\_1.fit(X\_train, y\_train, eval\_set=eval\_set,

eval\_metric='auc',

verbose=True, early\_stopping\_rounds=30)

The output should look something like this:

[0] validation\_0-auc:0.80412 validation\_1-auc:0.75223

[1] validation\_0-auc:0.84422 validation\_1-auc:0.79207

[2] validation\_0-auc:0.85920 validation\_1-auc:0.79278

[3] validation\_0-auc:0.86616 validation\_1-auc:0.79517

[4] validation\_0-auc:0.88261 validation\_1-auc:0.79659

[5] validation\_0-auc:0.88605 validation\_1-auc:0.80061

[6] validation\_0-auc:0.89226 validation\_1-auc:0.80224

[7] validation\_0-auc:0.89826 validation\_1-auc:0.80305

[8] validation\_0-auc:0.90559 validation\_1-auc:0.80095

[9] validation\_0-auc:0.91954 validation\_1-auc:0.79685

[10] validation\_0-auc:0.92113 validation\_1-auc:0.79608

…

[33] validation\_0-auc:0.99169 validation\_1-auc:0.78323

[34] validation\_0-auc:0.99278 validation\_1-auc:0.78261

[35] validation\_0-auc:0.99329 validation\_1-auc:0.78139

[36] validation\_0-auc:0.99344 validation\_1-auc:0.77994

CPU times: user 2.65 s, sys: 136 ms, total: 2.78 s

Wall time: 2.36 s

…

Notice that this took much less time than the previous fit. This is because, due to early stopping, we only trained 36 rounds of boosting. This means that the boosting procedure only needed 7 rounds to achieve the best validation score, as opposed to the 1000 we tried previously! You can access the number of boosting rounds needed to reach the optimal validation set score, as well as that score, with the booster attribute of the model object. This attribute presents a lower-level interface to the model than the scikit-learn API we have been using:

xgb\_model\_1.get\_booster().attributes()

The output should look like this, confirming the number of iterations and best validation score:

{'best\_iteration': '7', 'best\_score': '0.80305'}

From the training procedure, we can also see the ROC AUC after each round for both the training data, validation\_0-auc, and the validation data, validation\_1-auc, which provide insights into overfitting as the boosting procedure progresses. Here we can see that the validation score increased up to round 7, after which it started to decrease, indicating that further boosting would likely produce an undesirably overfit model. However, the training score continued to increase up to the point the procedure was terminated, showing how powerfully XGBoost is able to fit the training data.

We can further confirm that the fitted model object only represents 7 rounds of boosting, and check validation set performance, by manually calculating the ROC AUC as we did previously:

val\_set\_pred\_proba\_2 = xgb\_model\_1.predict\_proba(X\_val)[:,1]

roc\_auc\_score(y\_val, val\_set\_pred\_proba\_2)

This should output:

0.8030501882609966

This matches the highest validation score achieved after 7 rounds of boosting. So, with one simple tweak to the model training procedure, by using a validation set and early stopping, we were able to improve model performance on the validation set from about 0.78 to 0.80, a substantial increase. This shows the importance of early stopping in boosting.

One natural question to ask here is: “how did we know that 30 rounds for early stopping would be enough?”. One can experiment with this number like with any hyperparameter and different values may be appropriate for different data sets. You can look to see how the validation score changes with each boosting round to get an idea for this. Sometimes the validation score can increase and decrease in a jumpy way from round to round, so it’s a good idea to have enough rounds to make sure you’ve found the maximum, and boosted through any temporary decreases.

Tuning the Learning Rate

The learning rate is also referred to as eta in the XGBoost documentation, as well as step size shrinkage. This hyperparameter controls how large of a contribution each new estimator will make to the ensemble prediction. If you increase the learning rate, you may reach the optimal model, defined as having the highest performance on the validation set, faster. However, there is the danger that setting it too high will result in boosting steps that are too large. In this case the gradient boosting procedure may not converge on the optimal model, due to similar issues to those discussed in Exercise 16 regarding large learning rates in gradient descent. Let’s explore how the learning rate affects model performance on our synthetic data.

The learning rate is a number between zero and 1 (inclusive of endpoints, although a learning rate of zero is not useful). We make an array of 25 evenly spaced numbers between 0.01 and 1 for the learning rates we’ll test:

learning\_rates = np.linspace(start=0.01, stop=1, num=25)

Now we set up a for loop to train a model for each learning rate and save the validation scores in an array. We’ll also track the number of boosting rounds that it takes to reach the best iteration. The next several code blocks should be run together as one cell in a jupyter notebook. We start by measuring how long this will take, creating empty lists to store results, and opening the for loop:

%%time

val\_aucs = []

best\_iters = []

for learning\_rate in learning\_rates:

At each loop iteration, the variable learning\_rate will hold successive elements of the learning\_rates array. Once inside the loop, the first step is to update the hyperparameters of the model object with the new learning rate. This is accomplished using the set\_params method, which we supply with a double asterisk \*\* and a dictionary mapping hyperparameter names to values. The \*\* function call syntax allows us to supply an arbitrary number of keyword arguments, also called kwargs, as a dictionary. In this case we are only changing one keyword argument, so the dictionary only has one item:

xgb\_model\_1.set\_params(\*\*{'learning\_rate':learning\_rate})

Now that we’ve set the new learning rate on the model object, we train the model using early stopping as before:

xgb\_model\_1.fit(X\_train, y\_train, eval\_set=eval\_set,

eval\_metric='auc',

verbose=False, early\_stopping\_rounds=30)

After fitting, we obtain the predicted probabilities for the validation set, then use them to calculate the validation ROC AUC. This is added to our list of results using the append method:

val\_set\_pred\_proba\_2 = xgb\_model\_1.predict\_proba(X\_val)[:,1]

val\_aucs.append(roc\_auc\_score(y\_val, val\_set\_pred\_proba\_2))

Finally, we also capture the number of rounds needed for each learning rate:

best\_iters.append(

int(xgb\_model\_1.get\_booster().attributes()['best\_iteration']))

The previous five code snippets should all be run together in one cell. The output should be similar to this:

CPU times: user 1min 23s, sys: 526 ms, total: 1min 24s

Wall time: 22.2 s

Now that we have our results from this hyperparameter search, we can visualize validation set performance and number of iterations. Since these two metrics are on different scales, we’ll want to create a dual y-axis plot. Pandas makes this easier, so first we’ll put all the data into a dataframe:

learning\_rate\_df = \

pd.DataFrame({'Learning rate':learning\_rates,

'Validation AUC':val\_aucs,

'Best iteration':best\_iters})

Now we can visualize performance and number of iterations for different learning rates like this, noting that

* We set the index (set\_index) so that learning rate is plotted on the x-axis, and the other columns on the y-axes,
* The secondary\_y keyword argument indicates which column to plot on the right-hand y-axis, and
* The style argument allows us to specify different line styles for each column plotted. -o is a solid line with dots, while --o is a dashed line with dots.

mpl.rcParams['figure.dpi'] = 400

learning\_rate\_df.set\_index('Learning rate')\

.plot(secondary\_y='Best iteration', style=['-o', '--o'])

The resulting plot should look like this:

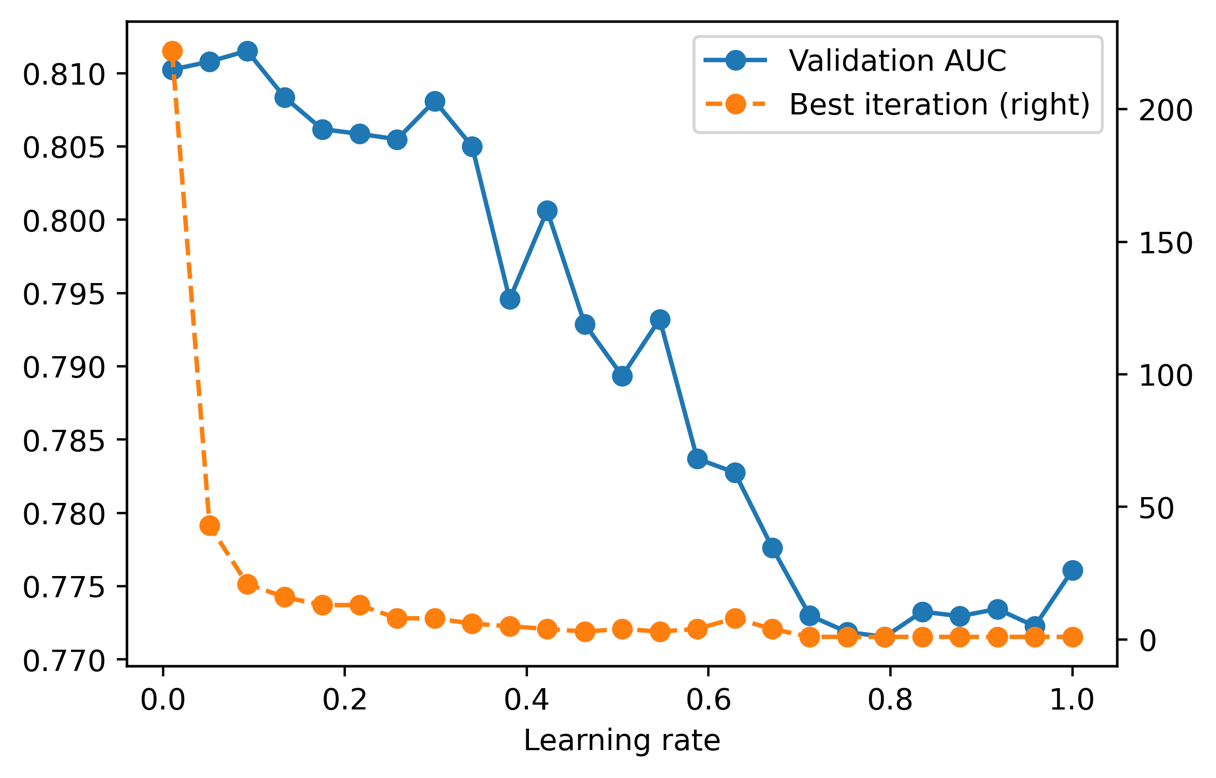


Figure 6.2: XGBoost model performance on a validation set, with number of boosting rounds until best iteration, for different learning rates

Overall, it appears that smaller learning rates result in better model performance for these synthetic data. By using a learning rate smaller than the default of 0.3, the best performance we can obtain can be seen as follows:

max(val\_aucs)

with output:

0.8115309360232714

By adjusting the learning rate, we were able to increase validation AUC from about 0.80 to 0.81, indicating the benefits of using an appropriate learning rate.

In general, smaller learning rates will usually result in better model performance, although they will require a larger number of boosting rounds, since the contribution of each round is smaller. This will translate into more time needed for model training. We can see this in the plot of number of rounds needed to reach the best iteration in Figure 6.2. In this case, it looks like good performance can be attained with less than 50 rounds, and the model training time is not that long for these data in any case. For larger data sets, training time may be longer. Depending on how much computational time you have, decreasing the learning rate and training more rounds can be an effective way to increase model performance.

When explore smaller learning rates, be sure to set the n\_estimators hyperparameter large enough to allow the training process to find the optimal model, ideally in conjunction with early stopping.

Other Important Hyperparameters in XGBoost

We’ve seen that overfitting in XGBoost can be compensated for by using different learning rates, as well as early stopping. What are some of the other hyperparameters that may be relevant? XGBoost has many hyperparameters and we won’t list them all here. You’re encouraged to consult the documentation (<https://xgboost.readthedocs.io/en/latest/parameter.html>) for a full list.

In the following exercise, we’ll do a grid search over ranges of 6 hyperparameters, including the learning rate. We will also include max\_depth, which should be familiar from Chapter 5 and controls the depth to which trees in the ensemble are grown. Aside from these we’ll also consider:

* gamma limits the complexity of trees in the ensemble, by only allowing a node to be split if the reduction in the loss function value is greater than a certain amount.
* min\_child\_weight also controls the complexity of trees, by only splitting nodes if they have at least a certain amount of “sample weight”. If all samples have equal weight (as they do for our exercise), this equates to the minimum number of training samples in a node. This is similar to min\_weight\_fraction\_leaf and min\_samples\_leaf for decision trees in scikit-learn.
* colsample\_bytree is a randomly selected fraction of features that will be used to grow each tree in the ensemble. This is similar to the max\_features parameter in scikit-learn (which does the selection at a node level as opposed to the tree level here). XGBoost also makes colsample\_bylevel and colsample\_bynode available to do the feature sampling at each level of each tree, and each node, respectively.
* subsample controls what fraction of samples from the training data is randomly selected prior to growing a new tree for the ensemble. This is similar to the bootstrap option for random forests in scikit-learn. Both this and the colsample parameters limit the information available during model training, increasing the bias of the individual ensemble members but hopefully also reducing the variance of the overall ensemble and improving out-of-sample model performance.

As you can see, gradient boosted trees in XGBoost implement several concepts that are familiar from decision trees and random forests. Now let’s explore how these hyperparameters affect model performance.

Exercise 22: Randomized grid search for tuning XGBoost hyperparameters

In this exercise we’ll use a randomized grid search to explore the space of six hyperparameters. Randomized grid search is a good option when you have many values of many hyperparameters you’d like to search over. We’ll look at 6 hyperparameters here. If, for example, there were 5 values for each of these that we’d like to test, we’d need 56 = 15,625 searches. Even if each model fit took only a second, we’d still need several hours to exhaustively search all possible combinations. A randomized grid search can achieve satisfactory results by only searching a random sample of all these combinations. Here we’ll show how to do this using scikit-learn and XGBoost.

The first step in a randomized grid search is to specify the range of values you’d like to sample from, for each hyperparameter. This can be done by either supplying a list of values, or a distribution object to sample from. In the case of discrete hyperparameters like max\_depth, where there are only a few possible values, it makes sense to specify them as a list. On the other hand, for continuous hyperparameters, such as subsample that can vary anywhere on the interval (0, 1], we don’t need to specify a list of values. Rather we can ask that the grid search randomly sample values in a uniform way over this interval. We will use a uniform distribution to sample several of the hyperparameters we consider.

1. Import the uniform distribution class from scipy and specify ranges for all hyperparameters to be searched, using a dictionary. uniform can take two arguments, loc and scale, specifying the lower bound of the interval to sample from, and the width of the interval, respectively:

from scipy.stats import uniform

param\_grid = {'max\_depth':[2,3,4,5,6,7],

'gamma':uniform(loc=0.0, scale=3),

'min\_child\_weight':list(range(1,151)),

'colsample\_bytree':uniform(loc=0.1, scale=0.9),

'subsample':uniform(loc=0.5, scale=0.5),

'learning\_rate':uniform(loc=0.01, scale=0.5)}

Here we’ve selected parameter ranges based on experimentation and experience. For example with subsample, the XGBoost documentation recommends choosing values of at least 0.5, so we’ve indicated uniform(loc=0.5, scale=0.5), which means sampling from the interval [0.5, 1].

1. Now that we’ve indicated which distributions to sample from, we need to do the sampling. scikit-learn offers the ParameterSampler class that will randomly sample the supplied param\_grid and return as many samples as requested (n\_iter). We also set the RandomState for repeatable results across different runs of the notebook:

from sklearn.model\_selection import ParameterSampler

rng = np.random.RandomState(0)

n\_iter=1000

param\_list = list(ParameterSampler(param\_grid, n\_iter=n\_iter,

random\_state=rng))

We have returned the results in a list of dictionaries of specific parameter values, corresponding to locations in the 6-dimensional hyperparameter space.

Note that in this exercise we are iterating through 1000 hyperparameter combinations, which will likely take over 5 minutes. You may wish to decrease this number for faster results.

1. Examine the first item of param\_list:

param\_list[0]

This should return a combination of 6 parameter values, from the indicated distributions:

{'colsample\_bytree': 0.5939321535345923,

'gamma': 2.1455680991172583,

'learning\_rate': 0.31138168803582195,

'max\_depth': 5,

'min\_child\_weight': 104,

'subsample': 0.7118273996694524}

1. Observe how you can set multiple XGBoost hyperparameters at once with a dictionary, using the \*\* syntax:

xgb\_model\_1.set\_params(\*\*param\_list[0])

The output should show the indicated hyperparameters being set:

XGBClassifier(base\_score=0.5, booster='gbtree', colsample\_bylevel=1,

colsample\_bynode=1, colsample\_bytree=0.5939321535345923,

gamma=2.1455680991172583, gpu\_id=-1, importance\_type='gain',

interaction\_constraints='', learning\_rate=0.31138168803582195,

max\_delta\_step=0, max\_depth=5, min\_child\_weight=104, missing=nan,

monotone\_constraints='()', n\_estimators=1000, n\_jobs=4,

num\_parallel\_tree=1, random\_state=0, reg\_alpha=0, reg\_lambda=1,

scale\_pos\_weight=1, subsample=0.7118273996694524,

tree\_method='exact', use\_label\_encoder=False,

validate\_parameters=1, verbosity=1)

We will use this procedure in a loop to look at all hyperparameter values.

1. The next several steps will be contained in one cell with a for loop. First, measure the time it will take to do this, create an empty list to save validation AUCs, and start a counter:

%%time

val\_aucs = []

counter = 1

1. Open the for loop, set hyperparameters, and fit the XGBoost model, similar to the example of tuning the learning rate above:

for params in param\_list:

#Set hyperparameters and fit model

xgb\_model\_1.set\_params(\*\*params)

xgb\_model\_1.fit(X\_train, y\_train, eval\_set=eval\_set,

eval\_metric='auc',

verbose=False, early\_stopping\_rounds=30)

1. Within the for loop, get the predicted probability and validation set AUC:

#Get predicted probabilities and save validation ROC AUC

val\_set\_pred\_proba = xgb\_model\_1.predict\_proba(X\_val)[:,1]

val\_aucs.append(roc\_auc\_score(y\_val, val\_set\_pred\_proba))

1. Since this procedure will take a few minutes, it’s nice to print the progress to the jupyter notebook output. We use the python remainder syntax % to print a message every 50 iterations, in other words when the remainder of counter divided by 50 equals zero. Finally we increment the counter.

#Print progress

if counter % 50 == 0:

print('Done with {counter} of {n\_iter}'.format(

counter=counter, n\_iter=n\_iter))

counter += 1

Assembling steps 5-8 in one cell and running the for loop should give output like this:

Done with 50 of 1000

Done with 100 of 1000

…

Done with 950 of 1000

Done with 1000 of 1000

CPU times: user 24min 20s, sys: 18.9 s, total: 24min 39s

Wall time: 6min 27s

1. Now that we have all the results from our hyperparameter exploration, we need to examine them. We can easily put all the hyperparameter combinations in a dataframe, since they are organized as a list of dictionaries. Do this and look at the first few rows:

xgb\_param\_search\_df = pd.DataFrame(param\_list)

xgb\_param\_search\_df.head()

The output should look like this:

Table

Description automatically generated

Figure 6.3: Hyperparameter combinations from a randomized grid search

1. We can also add the validation set ROC AUCs to the dataframe and see what the maximum is:

xgb\_param\_search\_df['Validation ROC AUC'] = val\_aucs

xgb\_param\_search\_df['Validation ROC AUC'].max()

The output should be:

0.8151220995602575

The result of searching over the hyperparameter space is that the validation set AUC is about 0.815. This is larger than the 0.812 we obtained with early stopping and searching over learning rates above, although not much. This means that, for these data, the default hyperparameters (aside from the learning rate) were sufficient to achieve pretty good performance. While we didn’t improve performance much with the hyperparameter search, it is instructive to see how the changing values of the hyperparameters affect model performance. We’ll examine the marginal distributions of AUCs with respect to each parameter individually in the following steps. This means we’ll look at how the AUCs change as one hyperparameter at a time changes, keeping in mind the fact that the other hyperparameters are also changing in our grid search results.

1. Set up a grid of 6 subplots for plotting performance against each hyperparameter using the following code, which also adjusts the figure resolution and starts a counter we’ll use to loop through the subplots:

mpl.rcParams['figure.dpi'] = 400

fig, axs = plt.subplots(3,2,figsize=(8,6))

counter = 0

1. Open a for loop to iterate through the hyperparameter names, which are the columns of the dataframe, not including the last column. Access the axes objects by flattening the 3 x 2 array returned by subplot above and indexing it with the counter. For each hyperparameter, use the plot.scatter method of the dataframe to make a scatter plot on each axes successively. The x-axis will show the hyperparameter, the y-axis the validation AUC, and the other options help us get black circular markers with white face colors (interiors).

for col in xgb\_param\_search\_df.columns[:-1]:

this\_ax = axs.flatten()[counter]

xgb\_param\_search\_df.plot.scatter(

x=col, y='Validation ROC AUC',

ax=this\_ax, marker='o',

color='w',

edgecolor='k',

linewidth=0.5)

1. The dataframe plot method will automatically create x- and y-axis labels. However, since the y-axis label will be the same for all of these plots, we only need to include it on the first one. So we set all the others to an empty string, '', and increment the counter.

if counter > 0:

this\_ax.set\_ylabel('')

counter += 1

Since we will be plotting marginal distributions, as we look at how validation AUC changes with a given hyperparameter, all the other hyperparameters are also changing. This means the relationship may be noisy. To get an idea of the overall trend, we are also going to create line plots with the average value of the validation AUC in each decile of the hyperparameter. Deciles organize data into bins based on whether the values fall into the bottom 10%, the next 10%, and so on up to the top 10%. Pandas offers a function called qcut which cuts a Series into quantiles, returning another series of the quantiles, as well as the endpoints of the quantile bins, which you can think of like histogram edges.

1. Use pandas qcut to generate a series of deciles (10 quantiles) for each hyperparameter (except max\_depth), returning the bin edges (there will be 11 of these for 10 quantiles) and dropping bin edges as needed if there are not enough unique values to divide into 10 quantiles (duplicates='drop'). Create a list of points halfway between each pair of bin edges for plotting.

if col != 'max\_depth':

out, bins = pd.qcut(xgb\_param\_search\_df[col], q=10,

retbins=True, duplicates='drop')

half\_points = [(bins[ix] + bins[ix+1])/2

for ix in range(len(bins)-1)]

1. For max\_depth, since there are only 6 unique values, we can directly use these values in a similar way to the deciles.

else:

out = xgb\_param\_search\_df[col]

half\_points = np.sort(xgb\_param\_search\_df[col].unique())

1. Create a temporary dataframe by copying the hyperparameter search dataframe, create a new column with the Series of deciles, and use this to find the average value of the validation AUC within each hyperparameter decile:

tmp\_df = xgb\_param\_search\_df.copy()

tmp\_df['param\_decile'] = out

mean\_df = tmp\_df.groupby('param\_decile').agg(

{'Validation ROC AUC':'mean'})

1. We can visualize results with a dashed line plot of the decile averages of validation AUC within each grouping, on the same axis as each scatter plot. Close the for loop and clean up the subplot formatting with plt.tight\_layout():

this\_ax.plot(half\_points, mean\_df.values, color='k', linestyle='--')

plt.tight\_layout()

The resulting image should look like this:

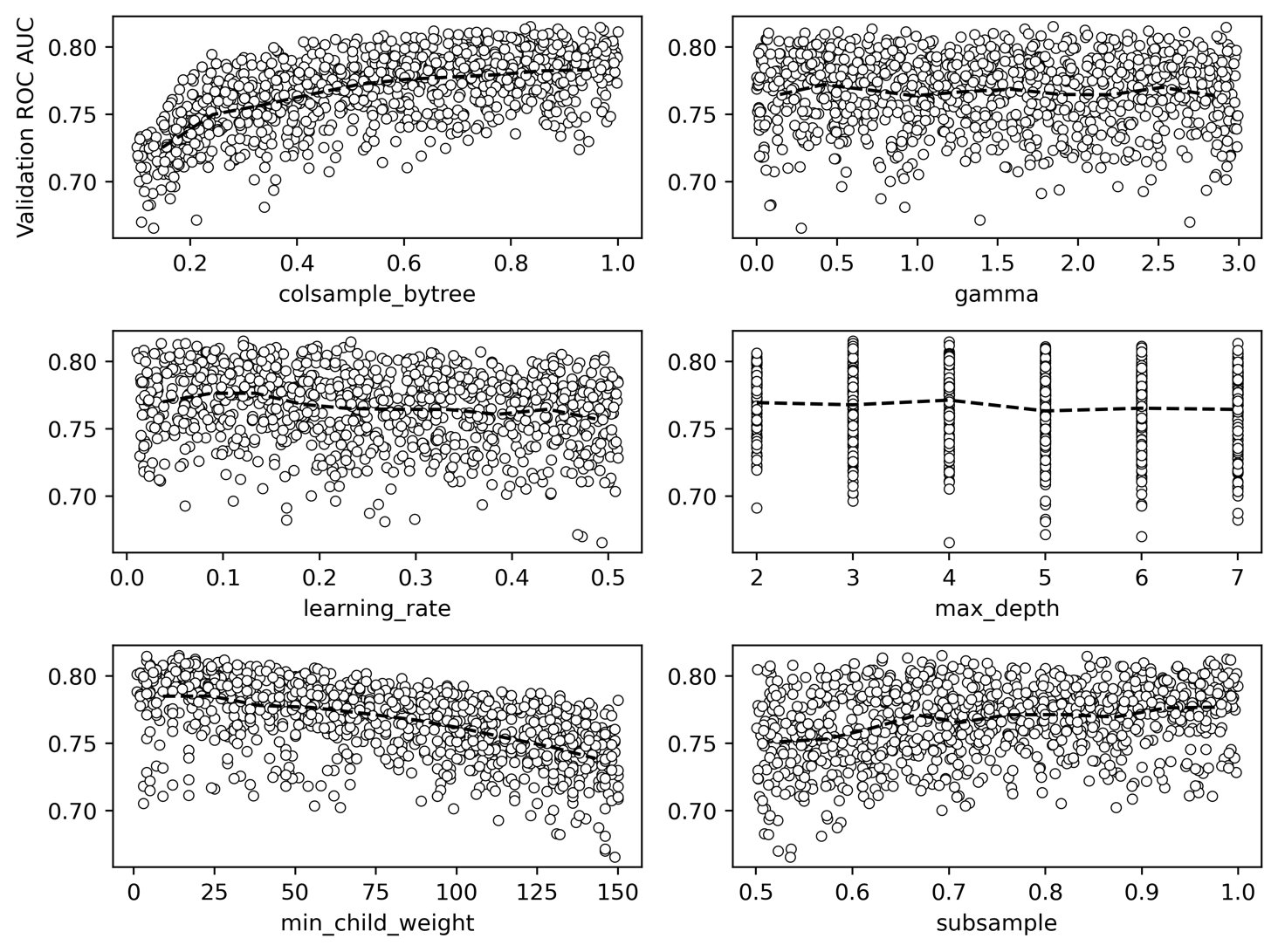


Figure 6.4: Validation AUCs plotted against each hyperparameter, along with average values within hyperparameter deciles

While we noted that the hyperparameter search in this exercise did not result in a substantial increase in validation AUC over previous efforts in this chapter, the plots in Figure 6.4 can still show us how XGBoost hyperparameters affect model performance for this particular dataset. One way that XGBoost combats overfitting is by limiting the data available when growing trees, either by randomly selecting only a fraction of the features available to each tree (colsample\_bytree), or a fraction of the training samples (subsample). However for these synthetic data, it appears the model performs best when using 100% of the features and samples for each tree; less than this and model performance steadily degrades. Another way to control overfitting is to limit the complexity of trees in the ensemble, by controlling their max\_depth, the minimum number of training samples in the leaves (min\_child\_weight), or the minimum loss function reduction required to split a node (gamma). Neither max\_depth nor gamma appear to have much effect on model performance in our example here, while limiting the number of samples in the leaves appears to be detrimental.

It appears that in this case, the gradient boosting procedure is robust enough on its own to achieve good model performance, without any additional tricks needed to reduce overfitting. Similar to what we observed above, however, having a smaller learning\_rate is beneficial.

1. We can show the optimal hyperparameter combination and the corresponding validation set AUC as follows:

max\_ix = xgb\_param\_search\_df['Validation ROC AUC'] == max\_auc

xgb\_param\_search\_df[max\_ix]

which should return a row of the dataframe similar to this:

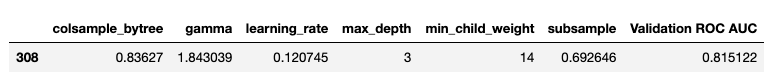


Figure 6.5: Optimal hyperparameter combination and validation set AUC

The validation set AUC is similar to what we achieved above by tuning only the learning rate.

Another way of growing trees: XGBoost’s grow\_policy

In addition to limiting the maximum depth of trees using a max\_depth hyperparameter, there is another paradigm for controlling tree growth: finding the node where a split would result in the greatest reduction in the loss function, and splitting this node, regardless of how deep it will make the tree. This may result in a tree with one or two very deep branches, while the other branches may not be grown very far. XGBoost offers a hyperparameter called grow\_policy and setting this to lossguide results in this kind of tree growth, while the depthwise option is the default and grows trees to an indicated max\_depth, as we’ve done in Chapter 5 and so far in this chapter. The lossguide grow\_policy is a newer option in XGBoost and mimics the behavior of LightGBM, another popular gradient boosting package.

In order to use the lossguide policy, it is necessary to set another hyperparameter we haven’t discussed yet: tree\_method, which must be set to hist or gpu-hist. Without going into too much detail, the hist policy will use a faster way of searching for splits. Instead of looking between every feature value for the training samples in a node, the hist method builds a histogram, and only considers splits on the edges of the histogram. So for example if there are 100 samples in a node, their feature values may be binned into 10 groups, meaning there are only 9 possible splits to consider instead of 99.

We can instantiate an XGBoost model for the lossguide grow\_policy as follows, using a learning rate of 0.1 based on intuition from our hyperparameter exploration in the previous exercise:

xgb\_model\_3 = xgb.XGBClassifier(

n\_estimators=1000,

max\_depth=0,

max\_leaves=50,

learning\_rate=0.1,

verbosity=1,

objective='binary:logistic',

use\_label\_encoder=False,

n\_jobs=-1,

tree\_method='hist',

grow\_policy='lossguide')

Notice here that we’ve set max\_depth=0, since this hyperparameter is not relevant for the lossguide policy. Instead, we are going to set a hyperparameter called max\_leaves, which simply controls the maximum number of leaves in the trees that will be grown. We’ll do a hyperparameter search of values ranging from 5 to 100 leaves:

max\_leaves\_values = list(range(5,105,5))

print(max\_leaves\_values[:5])

print(max\_leaves\_values[-5:])

which should output

[5, 10, 15, 20, 25]

[80, 85, 90, 95, 100]

Now we are ready to repeatedly fit and validate the model across this range of hyperparameter values, similar to how we’ve done above:

%%time

val\_aucs = []

for max\_leaves in max\_leaves\_values:

#Set parameter and fit model

xgb\_model\_3.set\_params(\*\*{'max\_leaves':max\_leaves})

xgb\_model\_3.fit(X\_train, y\_train, eval\_set=eval\_set,

eval\_metric='auc',

verbose=False, early\_stopping\_rounds=30)

#Get validation score

val\_set\_pred\_proba = xgb\_model\_3.predict\_proba(X\_val)[:,1]

val\_aucs.append(roc\_auc\_score(y\_val, val\_set\_pred\_proba))

The output will include the wall time for all of these fits, which was about 24 seconds in testing. Now let’s put the results in a dataframe:

max\_leaves\_df = \

pd.DataFrame({'Max leaves':max\_leaves\_values,

'Validation AUC':val\_aucs})

We can visualize how the validation AUC changes with the maximum number of leaves, similar to our visualization of the learning rate above:

mpl.rcParams['figure.dpi'] = 400

max\_leaves\_df.set\_index('Max leaves').plot()

Which will result in a plot like this:

Chart, line chart

Description automatically generated

Figure 6.6: Validation AUC against the max\_leaves hyperparameter

Smaller values of max\_leaves will limit the complexity of the trees grown for the ensemble, which will ideally increase bias but also decrease variance for improved out-of-sample performance. We can see this in higher validation set AUC when the threes are limited to 15 or 20 leaves. What is the maximum validation set AUC?

max\_auc = max\_leaves\_df['Validation AUC'].max()

max\_auc

which should output

0.8151200989120475

Let’s confirm this maximum validation AUC occurs at max\_leaves=20, as it appears in Figure 6.6:

max\_ix = max\_leaves\_df['Validation AUC'] == max\_auc

max\_leaves\_df[max\_ix]

This should return a row of the dataframe:

Table

Description automatically generated

Figure 6.7: Optimal max\_leaves

By using the lossguide grow\_policy, we can achieve performance at least as good as anything else we’ve tried so far. One key advantage of the lossguide policy is that, for larger datasets, it can result in training times that are faster than the depthwise policy, especially for smaller values of max\_leaves. While the dataset here is small enough that this is not of practical importance, this speed may be desirable in other applications.

Explaining model predictions with SHAP values

Along with cutting-edge modeling techniques like XGBoost, the practice of explaining model predictions has undergone substantial development in recent years. So far we’ve learned that logistic regression coefficients, or feature importances from random forests, can provide insight into the reasons for model predictions. A more powerful technique for explaining model predictions was described in a 2017 paper “A Unified Approach to Interpreting Model Predictions” by Scott Lundberg and Su-In Lee (<https://arxiv.org/abs/1705.07874>). This technique is known as SHAP (SHapley Additive exPlanations) as it is based on earlier work by mathematician Lloyd Shapley. Shapely developed an area of game theory to understand how coalitions of players can contribute to the overall outcome of a game. Recent machine learning research into model explanation leveraged this concept to consider how groups or coalitions of features in a predictive model contribute to the output model prediction. By considering the contribution of different groups of features, the SHAP method is able to isolate the effect of individual features.

Some notable aspects of using SHAP values to explain model predictions include:

* SHAP values can be used to make individualized explanations of model predictions; in other words the prediction of a single sample, in terms of the contribution of each feature, can be understood using SHAP. This is in contrast to the feature importance method of explaining random forests that we’ve already seen, which only considers the average importance of a feature across the model training set.
* SHAP values are calculated relative to a background dataset. By default this is the training data, although other data sets can be supplied.
* SHAP values are additive, meaning that for the prediction of an individual sample, the SHAP values can be added up to recover the value of the prediction, e.g. a predicted probability.

There are different implementations of the SHAP method for various types of models and here we will focus on SHAP for trees (Lundberg et al. 2019, <https://arxiv.org/abs/1802.03888>) to get insights into XGBoost model predictions on our validation set of synthetic data. First let’s refit the xgb\_model\_3 from the previous section with the optimal number of max\_leaves, 20:

%%time

xgb\_model\_3.set\_params(\*\*{'max\_leaves':20})

xgb\_model\_3.fit(X\_train, y\_train, eval\_set=eval\_set,

eval\_metric='auc',

verbose=False, early\_stopping\_rounds=30)

Now we’re ready to start calculating SHAP values for the validation dataset. There are 40 features and 1,000 samples here:

X\_val.shape

Should output

(1000, 40)

To automatically label the plots we can make with the shap package, we’ll put the validation set features in a dataframe with column names. We’ll use a list comprehension to make generic feature names e.g. “Feature 0, Feature 1, …” and create the dataframe as follows:

feature\_names = ['Feature {number}'.format(number=number)

for number in range(X\_val.shape[1])]

X\_val\_df = pd.DataFrame(data=X\_val, columns=feature\_names)

X\_val\_df.head()

The dataframe head should look like this:

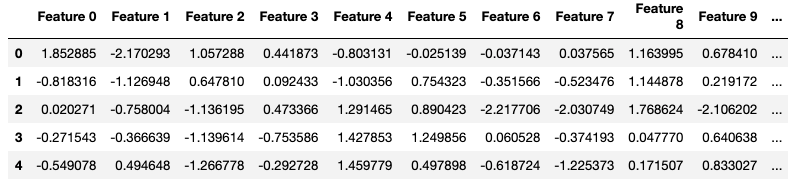


Figure 6.8: Dataframe of the validation features

With the trained model xgb\_model\_3 and the dataframe of validation features, we’re ready to create an explainer. The SHAP package has various kinds of explainers and we’ll use the one specifically for tree models:

explainer = shap.explainers.Tree(xgb\_model\_3, data=X\_val\_df)

This has created an explainer using the model validation data as the background dataset. Now we are ready to use the explainer to obtain SHAP values. The SHAP package makes this very simple. All we need to do is pass in the dataset we want explanations for:

shap\_values = explainer.shap\_values(X\_val\_df)

That’s all there is to it! What is this variable shap\_values that has been created? If you examine the contents of the shap\_values variable directly, you will see it contains three attributes. The first is values which contains the SHAP values. Let’s examine the shape:

shap\_values.values.shape

should return

(1000, 40)

Because SHAPs provide individualized explanations, there is a row for each of the 1,000 samples in the validation set. There are 40 columns because we have 40 features and SHAP values tell us the contribution of each feature to the prediction for each sample. shap\_values also contains a base\_values attribute, which is the naïve prediction before any feature contributions are considered, also defined as the average prediction across the entire dataset. There is one of these for each sample (1,000). Finally, there is also a data attribute which contains the feature values. All of this information can be combined in various ways to explain model predictions.

Thankfully, not only does the shap package provide fast and convenient methods for calculating SHAP values, it also provides a rich suite of visualization techniques. One of the most popular is a SHAP summary plot, which visualizes the contribution of each feature to each sample. Let’s create this plot and then understand what is being shown. Please note that most interesting SHAP visualizations rely heavily on color, so if you’re reading in black and white please refer to the github repository for color figures.

mpl.rcParams['figure.dpi'] = 75

shap.summary\_plot(shap\_values.values, X\_val\_df)

Should produce the following:

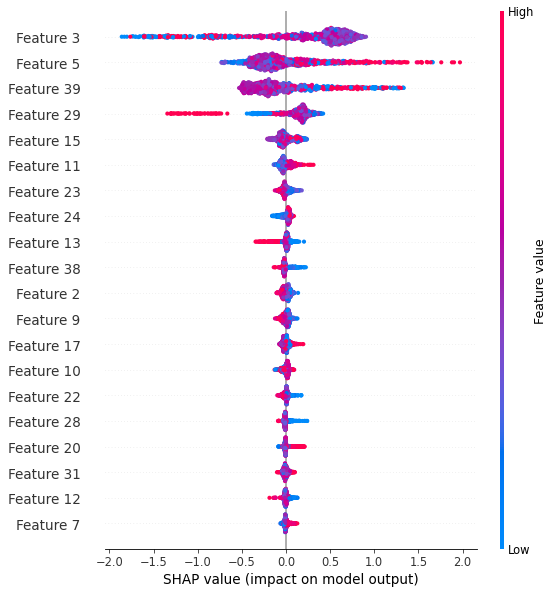


Figure 6.9: SHAP summary plot for the synthetic data validation set.

Figure 6.9 contains a lot of information to help us explain the model. The summary plot may contain up to 40,000 plotted points, one for each of the 40 features and each of the 1,000 validation samples (although only the first 20 features are shown by default). Let’s start by understanding the x-axis. The SHAP value indicates the additive contribution of each feature value to the prediction for a sample. SHAP values are shown here relative to the expected values, which are the base\_values described above. So if a given feature has a small impact on the prediction for a given sample, it will not tend to move the prediction very far from the expected value, and the SHAP value will be close to zero. However if a feature value has a large effect, which in the case of our binary classification problem means that the predicted probability will be pushed closer to 0 or 1, the SHAP value will be further from 0. Negative SHAP values indicate a feature moving the prediction closer to 0, positive SHAP values closer to 1.

Note that the SHAP values shown in Figure 6.9 cannot be directly interpreted as predicted probabilities. By default, SHAP values for the XGBoost binary classification model with the binary:logistic objective function are calculated and plotted using the log-odds representation of logistic regression, which was introduced in Chapter 3 in the section “Why is Logistic Regression Considered a Linear Model?”. This means that they can be added and subtracted, or in other words we can do linear transformations on them.

What about the color of the dots in Figure 6.9? These represent the values of the features for each sample, with red meaning a higher value and blue lower. So for example we can see in the second row of the plot, that high feature values (red dots) for Feature 5 in several cases push the predicted probability toward 1, since there appear to be many red dots to the right of the zero line. Feature values in the middle of the range are colored purple.

The vertical arrangement of the dots, in other words the width of the band of dots for each feature, indicates how many dots there are at that location on the x-axis. If there are many samples, the band of dots will be wider.

The vertical arrangement of features in the diagram is based on feature importance. The most important features, in other words those with the largest average effect on model predictions, are placed at the top of the list.

While the summary plot in Figure 6.9 is a great way to look at all of the most important features and their SHAP values at once, it may not reveal some interesting relationships. For example, the most important feature, Feature 3, appears to have a large clump of purple dots (middle of the range of feature values) that have positive SHAP values, while the negative SHAP values for this feature may result from high or low feature values. What is going on here? Often, when the effects of features seem unclear from a SHAP summary plot, the tree-based model we are using is capturing interaction effects between features. To gain additional insight into individual features and their interactions with others, we can use a SHAP scatter plot. Firstly, let’s make a simple scatter plot of the SHAP values of Feature 3. Note we can index the shap\_values object similar to a dataframe:

shap.plots.scatter(shap\_values[:,'Feature 3'])

This should produce the following plot:

Chart, scatter chart

Description automatically generated

Figure 6.10: Scatter plot of SHAP values for Feature 3

From Figure 6.10 we can tell pretty much the same information that we could from the summary plot of Figure 6.9: feature values in the middle of the range have high SHAP values, while those at the extremes are lower. However the scatter method also allows us to color the points of the scatter plot by another feature value, so we can see if there are interactions between the features. We’ll color points by the second most-important feature, Feature 5:

shap.plots.scatter(shap\_values[:,'Feature 3'],

color=shap\_values[:,'Feature 5'])

The resulting plot should look like this:

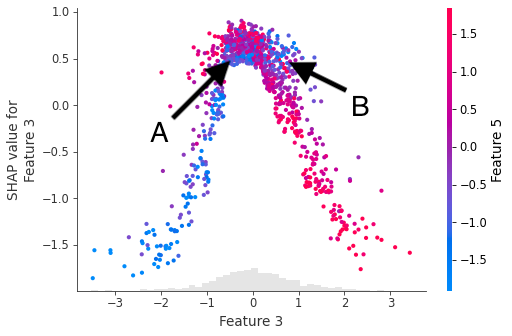


Figure 6.11: Scatter plot of SHAP values for Feature 3, colored by feature values of Feature 5. Arrows A and B indicated interesting interaction effects between these features.

Figure 6.11 shows an interesting interaction between Feature 3 and Feature 5. When samples are in the middle of the range of feature values for Feature 3, in other words at the top of the hump shape in Figure 6.11, the color of dots appears to get more red going from the bottom to the top of the cluster of dots here (arrow A). This means that for feature values in the middle of the Feature 3 range, as the value of Feature 5 increases, so does the SHAP value for Feature 3. We can also see that as feature values of Feature 3 increase along the x-axis from the middle toward the top of the range, this relationship reverses to where higher feature values for Feature 5 begin to correspond to lower SHAP values for Feature 3 (arrow B). So the interaction with Feature 5 appears to have a substantial impact on the SHAP values for Feature 3.

The complex relationships depicted in Figure 6.11 show how increasing a feature value may lead to either increasing or decreasing SHAP values when interaction effects are present. The specific reasons for the patterns in Figure 6.11 relate back to our creation of the synthetic dataset we are modeling, where we specified multiple clusters in the feature space. As discussed in Chapter 5, in the section “Using Decision Trees: Advantages and Predicted Probabilities”, tree-based models like XGBoost are able to effectively model clusters of points in multi-dimensional feature space, that belong to a certain class. SHAP explanations can help us understand how the model is making these representations.

Here we’ve used synthetic data, and the features have no real-world interpretation, so we can’t assign any meaning to interactions we observe. However with real-world data, detailed exploration with SHAP values and interactions can provide insight into how a model is representing complex relationships between attributes of customers or users, for example. SHAP values are also useful since they can provide explanations relative to any background data set. While logistic regression coefficients and feature importances of random forests are determined entirely by the model training data, SHAP values can be calculated for any background dataset; so far in this chapter we’ve been using the validation data. This provides an opportunity, when predicted models are deployed in a production environment, to understand how new predictions are being made. If the SHAP values for new predictions are very different from those of model training and testing data, it may indicate the nature of incoming data has changed, and it may be time to consider developing a new model. We’ll consider these practical aspects of using models in the real world in the final chapter.

Exercise 23: Plotting SHAP interactions, feature importance, and reconstructing predicted probabilities from SHAP values

In this exercise you’ll get more familiar with using SHAP values to provide visibility into the workings of a model. First we’ll take an alternate look at the interaction between Features 3 and 5, then use SHAP values to calculate feature importances similar to what we did with a random forest model in Chapter 5. Finally we’ll see how model outputs can be obtained from SHAP values, taking advantage of their additive property.

1. Given the preliminary steps accomplished in this section already, we can take another look at the interaction between Features 3 and 5, the two most important features of the synthetic data set. Use the following code to make an alternate version of Figure 6.11, except this time look at the SHAP values of Feature 5:

shap.plots.scatter(shap\_values[:,'Feature 5'],

color=shap\_values[:,'Feature 3'])

The resulting plot should look like this:

Chart, scatter chart

Description automatically generated

Figure 6.12: Figure 6.11: Scatter plot of SHAP values for Feature 5, colored by feature values of Feature 3

As opposed to Figure 6.11, here we are seeing the SHAP values of Feature 5. In general, from the scatter plot, we can see that SHAP values tend to increase as feature values increase for Feature 5. However there are certainly counterexamples to that general trend, as well as an interesting interaction with Feature 3: for a given value of Feature 5, which can be thought of as a vertical slice from the image, the color of the dots can either become more red going from the bottom to the top, for negative feature values, or less red for positive feature values. This means that for a given value of Feature 5, its SHAP value depends on the value of Feature 3. This is further illustration of the interesting interaction between Features 3 and 5. In a real project, which plot you would choose to show depends on what kind of story you want to tell with the data, relating to what real-world quantifies Features 3 and 5 might represent.

1. Create a feature importance bar plot using the following code:

mpl.rcParams['figure.dpi'] = 75

shap.summary\_plot(shap\_values.values, X\_val, plot\_type='bar')

A picture containing graphical user interface

Description automatically generated

Figure 6.13: Feature importance bar plot using SHAP values

The feature importance bar plot gives a visual presentation of information similar to that obtained in Exercise 21 of Chapter 5 with a random forest: this is a single number for each feature, representing how important it is overall for a data set.

Do these results make sense? Recall we created these synthetic data with 3 informative features and 2 redundant ones. In Figure 6.13 it appears that there are 4 features that are substantially more important than all the others, so perhaps one of the redundant features was created in such a way that XGBoost selected it for splitting nodes fairly often, but the other redundant feature was not used as much.

Compared to the feature importances we found previously in Chapter 5, the ones here are a bit different. The feature importances we can obtain from scikit-learn for a random forest model are calculated using the decrease in node impurity due to the feature as well as the fraction of training samples split by the feature. By contrast, feature importances using SHAP values are calculated as follows: first the absolute value of all the SHAP values (shap\_values.values) is taken, then an average over all samples is taken for each feature, as implied by the x-axis label. The interested reader can confirm this by directly calculating these metrics from the shap\_values.

Now that we’ve familiarized ourselves with a range of uses of SHAP values, let’s see how their additive property allows reconstruction of predicted probabilities.

1. SHAP values are calculated relative to the expected value, or base value, of a model. This can be interpreted as the average prediction over all samples in the background dataset, however the prediction will be in units of log-odds as opposed to probability as mentioned above, to support additivity. The expected value of a model can be accessed from the explainer object as follows:

explainer.expected\_value

The output should look like this:

-0.30949621941894295

This information isn’t particularly useful on its own, however it gives us the baseline from which we can reconstruct predicted probabilities.

1. Recall that the shape of the SHAP values matrix is the number of samples by the number of features; in our exercise with the validation data here that would be 1,000 by 40. In order to add up all the SHAP values for each sample, we therefore want to take a sum over the column axis (axis=1). This adds all the feature contributions, effectively providing the offset from the expected value. If we add the expected value to this, we then have the predictions:

shap\_sum = shap\_values.values.sum(axis=1) + explainer.expected\_value

shap\_sum.shape

This should return

(1000,)

Indicating we now have a single number for each sample. However, this prediction is in log-odds space. In order to transform to probability space, we need to apply the logistic function introduced in Chapter 3.

1. Apply the logistic transformation to log-odds predictions like this:

shap\_sum\_prob = 1 / (1 + np.exp(-1 \* shap\_sum))

Now we’d like to compare the predicted probabilities obtained from SHAP values with direct model output for confirmation.

1. Obtain predicted probabilities for the model validation set and check the shape with this code:

y\_pred\_proba = xgb\_model\_3.predict\_proba(X\_val)[:,1]

y\_pred\_proba.shape

The output should be

(1000,)

Which is the same shape as our SHAP-derived predictions, as expected.

1. Put the model output and sums of SHAP values together in a dataframe for side-by-side comparison, and spot check a random selection of 5 rows:

df\_check = pd.DataFrame(

{'SHAP sum':shap\_sum\_prob,

'Predicted probability':y\_pred\_proba})

df\_check.sample(5, random\_state=1)

The output should confirm that the two methods have identical results:

Table

Description automatically generated

Figure 6.14: Comparison of SHAP-derived predicted probabilities and those obtained directly from XGBoost.

The spot-check indicates these 5 samples have identical values. While the values may not be precisely equal due to rounding errors of machine arithmetic, you could use numpy’s allclose function to ensure they’re the same within a user-configurable amount of rounding error.

1. Ensure the SHAP-derived probabilities and model output probabilities are all very close to each other like this:

np.allclose(df\_check['SHAP sum'], df\_check['Predicted probability'])

The output should be

True

Indicating all elements of both columns are equal within rounding error. allclose is useful for when rounding errors are present and exact equality (testable with np.array\_equal) would not hold.

By now you should be getting an impression of the power of SHAP values to help understand machine learning models. The sample-specific, individualized nature of SHAP values opens up the possibility of very detailed analyses which could help answer a wide variety of potential questions from business stakeholders such as “how would the model make predictions for people like this?” or “why did the model make this prediction for this specific person”? Now that we’re familiar with XGBoost and SHAP values, two state-of-the-art machine learning techniques, we return to the case study data to apply them.

Missing data

As a final note on the usage of both XGBoost and SHAP, one valuable trait of both packages is their ability to handle missing values. Recall that in Chapter 1 we found some samples in the case study data had missing values for the feature PAY\_1. So far, our approach has been to simply remove these samples from the dataset when building models. This is because without specifically addressing the missing values in some way, the machine learning models implemented by scikit-learn cannot work with the data. Ignoring them is one approach, although this may not be satisfactory as it involves throwing away data. If it’s a very small fraction of the data, this may be fine, however in general it’s good to be able to know how to deal with missing values.

There are several approaches for imputing missing values of features, such as filling them in with the mean or mode of the non-missing values of that feature, or a randomly selected value from the non-missing values. One can also build a model outputting the feature in question as the response variable, with all the other features acting as features for this new model, and then predict the missing feature values. These approaches were explored in the first edition of this book (<https://github.com/TrainingByPackt/Data-Science-Projects-with-Python/blob/master/Lesson06/Lesson06.ipynb>). However, since XGBoost typically performs better than other machine learning models for binary classification tasks using tabular data like we’re using here, and handles missing values, we’ll forego more in-depth exploration of imputing missing values and let XGBoost do the work for us.

How does XGBoost handle missing data? At every opportunity to split a node, XGBoost considers only the non-missing feature values. If a feature with missing values is chosen to make a split, the samples with missing values for that feature are then sent down the optimal path to one of the child nodes, in terms of minimizing the loss function.

Summary

In this chapter we’ve learned some of the most cutting-edge techniques for building machine learning models with tabular data. While other types of data, such as image or text data, warrant exploration with different types of models such as neural networks, many standard business applications leverage tabular data. XGBoost and SHAP are some of the most advanced and popular tools you can use to build and understand models with this kind of data. Having gained familiarity and practical experience using these tools with synthetic data, in the following activity we return to the dataset for the case study and see how we can use XGBoost to model it, including the samples with missing feature values, and use SHAP values to understand the model.

Activity 6: Modeling the case study data with XGBoost and explaining the model with SHAP

In this activity, we’ll take what we’ve learned in this chapter with a synthetic dataset and apply it to the case study data. We’ll see how an XGBoost model performs on a validation set and explain the model predictions using SHAP values. As preliminary steps, we need to load the original dataset and replace the samples that had missing values for the PAY\_1 feature.

1. Load the original case study data.
2. Repeat the steps from Chapter 1 where we remove rows that have zeros for all features and the response variable.
3. Reassign undocumented feature values as done in Chapter 1.
4. Now instead of removing samples where PAY\_1 == ‘Not available’, create a new dataframe with only these samples, to combine with the rest of the cleaned data.
5. Load the cleaned data from Chapter 1 and create a list of column names for the features and response variable.
6. Replicate the train/test split from previous efforts, using the same random seed.
7. Copy the dataframe with missing values for PAY\_1 and convert these to np.nan, which XGBoost will recognize as missing values.
8. Split the data with missing values in the same 80/20 proportion as the cleaned data.
9. Concatenate the samples with missing feature values on to the cleaned data separately for training and testing data.
10. Define a validation set to train XGBoost with early stopping.
11. Instantiate an XGBoost model. Use the lossguide grow policy to enable examination of validation set performance for several values of max\_leaves.
12. Create a list of values of max\_leaves from 5 to 200, counting by 5’s.
13. Create the evaluation set for early stopping.
14. Use the same technique as above to loop through hyperparameter values and create a list of validation ROC AUC’s.
15. Create a dataframe of the hyperparameter search results and plot validation AUC against max\_leaves.
16. Observe the number of max\_leaves corresponding to the highest ROC AUC on the validation set.
17. Refit the XGBoost model with the optimal hyperparameter.
18. So that we can examine SHAP values for the validation set, make a dataframe of these data.
19. Create a SHAP explainer for our new model using the validation data as the background dataset, obtain the SHAP values, and make a summary plot.
20. Make a scatter plot of LIMIT\_BAL SHAP values, colored by the feature with the strongest interaction.
21. Save the trained model along with the training and testing data to a file.

Activity 6: Modeling the case study data with XGBoost and explaining the model with SHAP

In this activity, we’ll take what we’ve learned in this chapter with a synthetic dataset and apply it to the case study data. We’ll see how an XGBoost model performs on a validation set and explain the model predictions using SHAP values. As preliminary steps, we need to load the original dataset and replace the samples that had missing values for the PAY\_1 feature.

1. Load the original case study data as follows:

df\_orig = pd.read\_excel( '../Data/default\_of\_credit\_card\_clients\_\_courseware\_version\_1\_21\_19.xls')

1. Repeat the steps from Chapter 1 where we remove rows that have zeros for all features and the response variable:

df\_zero\_mask = df\_orig == 0

feature\_zero\_mask = df\_zero\_mask.iloc[:,1:].all(axis=1)

df\_clean = df\_orig.loc[~feature\_zero\_mask,:].copy()

1. Reassign undocumented feature values as done in Chapter 1:

df\_clean['EDUCATION'].replace(to\_replace=[0, 5, 6], value=4, inplace=True)

df\_clean['MARRIAGE'].replace(to\_replace=0, value=3, inplace=True)

1. Now instead of removing samples where PAY\_1 == ‘Not available’, we create a new dataframe with only these samples, which we’ll combine with the rest of the cleaned data.

missing\_pay\_1\_mask = df\_clean['PAY\_1'] == 'Not available'

df\_missing\_pay\_1 = df\_clean.loc[missing\_pay\_1\_mask,:].copy()

In order to have a similar unseen test set to what we have been working with in previous chapters, we’ll load the cleaned data from Chapter 1 and make the same train/test split. After that we’ll place samples with missing values for PAY\_1 into the train and test sets in the appropriate proportion.

1. Load the cleaned data from Chapter 1 and create a list of column names for the features and response variable:

df = pd.read\_csv('../Data/Chapter\_1\_cleaned\_data.csv')

features\_response = df.columns.tolist()

items\_to\_remove = ['ID', 'SEX', 'PAY\_2', 'PAY\_3', 'PAY\_4', 'PAY\_5',

'PAY\_6', 'EDUCATION\_CAT', 'graduate school',

'high school', 'none', 'others', 'university']

features\_response = [item for item in features\_response if item not in items\_to\_remove]

1. Replicate the train/test split from previous efforts, using the same random seed:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = \

train\_test\_split(df[features\_response[:-1]].values,

df['default payment next month'].values,

test\_size=0.2, random\_state=24)

1. Copy the dataframe with missing values for PAY\_1 and convert these to np.nan, which XGBoost will recognize as missing values:

df\_fill\_pay\_1\_filled = df\_missing\_pay\_1.copy()

df\_fill\_pay\_1\_filled['PAY\_1'] = np.nan

1. Split the data with missing values in the same 80/20 proportion as the cleaned data:

X\_fill\_pay\_1\_train, X\_fill\_pay\_1\_test,\

y\_fill\_pay\_1\_train, y\_fill\_pay\_1\_test = \

train\_test\_split(

df\_fill\_pay\_1\_filled[features\_response[:-1]],

df\_fill\_pay\_1\_filled['default payment next month'],

test\_size=0.2, random\_state=24)

1. Concatenate the samples with missing feature values on to the cleaned data separately for training and testing data:

X\_train\_all = np.concatenate((X\_train, X\_fill\_pay\_1\_train), axis=0)

y\_train\_all = np.concatenate((y\_train, y\_fill\_pay\_1\_train), axis=0)

X\_test\_all = np.concatenate((X\_test, X\_fill\_pay\_1\_test), axis=0)

y\_test\_all = np.concatenate((y\_test, y\_fill\_pay\_1\_test), axis=0)

At this point, we have the training data we want to use for an XGBoost model. Since we’d like to do early stopping, we will need to split off a validation set from the training set. We’ll only use one validation set for this activity, however the interested reader may wish to do k-fold cross validation and average the results across folds similar to what we did in Activity 5.

1. Define a validation set to train XGBoost with early stopping:

from sklearn.model\_selection import train\_test\_split

X\_train\_2, X\_val\_2, y\_train\_2, y\_val\_2 = \

train\_test\_split(X\_train\_all, y\_train\_all, test\_size=0.2, random\_state=24)

1. Instantiate an XGBoost model. We’ll use the lossguide grow policy and examine validation set performance for several values of max\_leaves.

xgb\_model\_4 = xgb.XGBClassifier(

n\_estimators=1000,

max\_depth=0,

learning\_rate=0.1,

verbosity=1,

objective='binary:logistic',

use\_label\_encoder=False,

n\_jobs=-1,

tree\_method='hist',

grow\_policy='lossguide')

1. Search values of max\_leaves from 5 to 200, counting by 5’s.

max\_leaves\_values = list(range(5,205,5))

1. Create the evaluation set for early stopping:

eval\_set\_2 = [(X\_train\_2, y\_train\_2), (X\_val\_2, y\_val\_2)]

1. Use the same technique as above to loop through hyperparameter values and create a list of validation ROC AUC’s.

%%time

val\_aucs = []

for max\_leaves in max\_leaves\_values:

#Set parameter and fit model

xgb\_model\_4.set\_params(\*\*{'max\_leaves':max\_leaves})

xgb\_model\_4.fit(X\_train\_2, y\_train\_2, eval\_set=eval\_set\_2,

eval\_metric='auc',

verbose=False, early\_stopping\_rounds=30)

#Get validation score

val\_set\_pred\_proba = xgb\_model\_4.predict\_proba(X\_val\_2)[:,1]

val\_aucs.append(roc\_auc\_score(y\_val\_2, val\_set\_pred\_proba))

1. Create a dataframe of the hyperparameter search results and plot validation AUC against max\_leaves:

max\_leaves\_df\_2 = \

pd.DataFrame({'Max leaves':max\_leaves\_values,

'Validation AUC':val\_aucs})

mpl.rcParams['figure.dpi'] = 400

max\_leaves\_df\_2.set\_index('Max leaves').plot()

The plot should look something like this:

Text, whiteboard

Description automatically generated

Figure 6.15: Validation AUC versus max\_leaves for the case study data.

Although the relationship is somewhat noisy, we see that in general lower values of max\_leaves result in higher validation set ROC AUC. This is because limiting the complexity of trees by allowing fewer leaves results in less overfitting, and increases the validation set score.

1. Observe the number of max\_leaves corresponding to the highest ROC AUC on the validation set:

max\_ix\_2 = max\_leaves\_df\_2['Validation AUC'] == max\_auc\_2

max\_leaves\_df\_2[max\_ix\_2]

The result should be:

Table

Description automatically generated with medium confidence

Figure 6.16: Optimal max\_leaves and validation set AUC for the case study data

While this is not a perfect comparison to previous results, since here we’ve

* Used the validation set for early stopping,
* Have missing values in the training and validation data, and
* Only have one validation set as opposed to k-folds cross validation,

We note that the validation ROC AUC of 0.779 here is a bit higher than the 0.776 obtained previously with random forest in Activity 5. These validation scores are fairly similar and it would probably be fine to use either model in practice. We’ll move forward with the XGBoost model.

1. Refit the XGBoost model with the optimal hyperparameter:

xgb\_model\_4.set\_params(\*\*{'max\_leaves':40})

xgb\_model\_4.fit(X\_train\_2, y\_train\_2, eval\_set=eval\_set\_2,

eval\_metric='auc',

verbose=False, early\_stopping\_rounds=30)

1. So that we can examine SHAP values for the validation set, make a dataframe of these data:

X\_val\_2\_df = pd.DataFrame(data=X\_val\_2, columns=features\_response[:-1])

1. Create a SHAP explainer for our new model using the validation data as the background dataset, obtain the SHAP values, and make a summary plot:

explainer\_2 = shap.explainers.Tree(xgb\_model\_4, data=X\_val\_2\_df)

shap\_values\_2 = explainer\_2(X\_val\_2\_df)

mpl.rcParams['figure.dpi'] = 75

shap.summary\_plot(shap\_values\_2.values, X\_val\_2\_df)

The plot should look like this:

A picture containing chart

Description automatically generated

Figure 6.17: SHAP values for the XGBoost model of the case study data on the validation set

From Figure 6.17, we can see that the most important features in the XGBoost model are somewhat different from those in the random forest model we explored in Chapter 5 (Figure 5.15). No longer is PAY\_1 the most important feature, although it is still quite important at number 3. LIMIT\_BAL, the borrower’s credit limit, is now the most important feature. This makes sense as an important feature as the lender has likely based the credit limit on how risky a borrower is, so it should be a good predictor of the risk of default.

Let’s explore whether LIMIT\_BAL has any interesting SHAP interactions with other features. Instead of specifying which feature to color the scatter plot by, we can let the shap package pick the feature that has the most interaction by not indexing the explainer object for the color argument.

1. Make a scatter plot of LIMIT\_BAL SHAP values, colored by the feature with the strongest interaction:

shap.plots.scatter(shap\_values\_2[:,'LIMIT\_BAL'],

color=shap\_values\_2)

The plot should look like this:

Chart, scatter chart

Description automatically generated

Figure 6.18: Scatter plot of SHAP values of LIMIT\_BAL and the feature with the strongest interaction

BILL\_AMT2, the amount of the bill from two months previous, has the strongest interaction with LIMIT\_BAL. We can see that for most values of LIMIT\_BAL, if the bill was particularly high, this leads to more positive SHAP values, meaning increased risk of default. This can be observed by noting that most of the reddest colored dots appear along the top of the band of dots in Figure 6.18. This makes intuitive sense: even if a borrower was given a large credit limit, if their bill gets very large, this may signal an increased risk of default.

Finally, we will save the model along with the training and testing data for analysis and delivery to our business partner. We accomplish this using Python’s pickle functionality.

1. Save the trained model along with the training and testing data to a file:

with open('../Data/xgb\_model\_w\_data.pkl', 'wb') as f:

pickle.dump([X\_train\_all, y\_train\_all,

X\_test\_all, y\_test\_all,

xgb\_model\_4], f)