**Extreme Gradient Boosting with XGBoost**

Do you know the basics of supervised learning and want to use state-of-the-art models on real-world datasets? Gradient boosting is currently one of the most popular techniques for efficient modeling of tabular datasets of all sizes. XGboost is a very fast, scalable implementation of gradient boosting, with models using XGBoost regularly winning online data science competitions and being used at scale across different industries. In this course, you'll learn how to use this powerful library alongside pandas and scikit-learn to build and tune supervised learning models. You'll work with real-world datasets to solve classification and regression problems.

#### 01 - Classification with XGBoost

**1. Welcome to the course!**

Hi, my name is Sergey Fogelson and I'm the instructor for Datacamp's course on Gradient Boosted Trees With XGBoost. I'm a data scientist working in the media industry and have used XGBoost extensively on a variety of machine learning problems. I've created this course with DataCamp to help others quickly understand how to use this very popular implementation of gradient boosting. Let's get started.

**2. Before we get to XGBoost...**

In order to understand XGBoost, we need to have some handle on the broader topics of supervised classification, decision trees, and boosting, which we will cover throughout this chapter. To begin, let's briefly review what

**3. Supervised learning**

supervised learning is and the kinds of problems its methods can be applied to. At its core, supervised learning, which is the kind of learning problems that XGBoost can be applied to, relies on labeled data. That is, you have some understanding of the past behavior of the problem you're trying to solve or what you're trying to predict.

**4. Supervised learning example**

For example, assessing whether a specific image contains a person's face, is a classification problem. Here the training data are images converted into vectors of pixel values, and the labels are either 1 when the image contains a face or 0 when the image doesn't contain a face. Given this, there are two kinds of supervised learning problems that account for the vast majority of use-cases: classification problems and regression problems. We will only talk about classification problems here and leave regression to chapter 2.

**5. Supervised learning: Classification**

Classification problems involve predicting either binary or multi-class outcomes.

**6. Binary classification example**

For example, predicting whether a person will purchase an insurance package given some quote is a binary supervised learning problem,

**7. Multi-class classification example**

and predicting whether a picture contains one of several species of birds is a multi-class supervised learning problem. When dealing with binary supervised learning problems,

**8. AUC: Metric for binary classification models**

the AUC, or Area Under the Receiver Operating Characteristic Curve, is the most versatile and common evaluation metric used to judge the quality of a binary classification model. It is simply the probability that a randomly chosen positive data point will have a higher rank than a randomly chosen negative data point for your learning problem. So, a higher AUC means a more sensitive, better performing model. When dealing with multi-class classification problems,

**9. Accuracy score and confusion matrix**

it is common to use the accuracy score (higher is better) and to look at the overall confusion matrix to evaluate the quality of a model.

**10. Review**

Some common algorithms for classification problems include logistic regression and decision trees. If you want a deeper review, check out DataCamp's introductory course on supervised learning.

**11. Other supervised learning considerations**

All supervised learning problems, including classification problems, require that the data is structured as a table of feature vectors, where the features themselves (also called attributes or predictors) are either numeric or categorical. Furthermore, it is usually the case that numeric features are scaled to aid in either feature interpretation or to ensure that the model can be trained properly (for example, numerical feature scaling is essential to ensure properly trained support vector machine models). Categorical features are also almost always encoded before applying supervised learning algorithms, most commonly using one-hot encoding. Finally, other kinds of supervised learning problems exist, so I'll mention them here briefly.

**12. Ranking**

Ranking problems involve predicting an ordering on a set of choices (like google search suggestions),

**13. Recommendation**

and recommendation problems involve recommending an item or set of items to a user based on his/her consumption history and profile (like Netflix).

**14. Let's practice!**

Now that you've been reminded about the basics of classification problems, let's get to work!

Got It!

1. Introducing XGBoost

Now let's talk about what you're actually here for, the hottest library in supervised machine learning, XGBoost.

2. What is XGBoost?

XGBoost is an incredibly popular machine learning library for good reason. It was developed originally as a C++ command-line application. After winning a popular machine learning competition, the package started being adopted within the ML community. As a result, bindings, or functions that tapped into the core C++ code, started appearing in a variety of other languages, including Python, R, Scala, and Julia. We will cover the Python API in this course.

3. What makes XGBoost so popular?

What makes XGBoost so popular? Its speed and performance. Because the core XGBoost algorithm is parallelizable, it can harness all of the processing power of modern multi-core computers. Furthermore, it is parallelizable onto GPU's and across networks of computers, making it feasible to train models on very large datasets on the order of hundreds of millions of training examples. However, XGBoost's speed isn't the package's real draw. Ultimately, a fast but poorly performing machine learning algorithm is not going to have wide adoption within the community. What makes XGBoost so popular is that it consistently outperforms almost all other single-algorithm methods in machine learning competitions and has been shown to achieve state-of-the-art performance on a variety of benchmark machine learning datasets. Here's an example of how we can use XGBoost using a classification problem.

4. Using XGBoost: a quick example

In lines 1-4, we import the libraries or functions we will be using, including xgboost, and the train/test/split function from scikit-learn. Remember, you always build a machine learning model using train/test splits of your data, where some portion of your data is used for training, and the remainder is held-out for testing to ensure that your model doesn't overfit and can generalize to unseen data. In lines 5 and 6 we load our data in from file and split the entire dataset into a matrix of samples by features, called X by convention, and a vector of target values, called y by convention. In line 7 we create our train/test split, keeping 20% of the data for testing. In line 8 we instantiate our xgboost classifier instance with some parameters that we will cover shortly. Lines 9 and 10 should appear familiar to you. XGBoost has a scikit-learn compatible api, and this is it! It uses the fit/predict pattern that you should have seen before, where we fit, or train, our algorithm on the training set, and then evaluate it by generating predictions using the test set and comparing our predictions to the actual target labels on the test set. Lines 11 and 12 evaluate the accuracy of the trained model on the test set and print the results to screen.

5. Let's begin using XGBoost!

Given that XGBoost is this popular, let's get to using it already!

**Got It!**

**1. What is a decision tree?**

Because XGBoost is usually used with trees as base learners, we need to understand what an individual decision tree is, and how it works.

**2. Visualizing a decision tree**

Here is an example decision tree. As you can see, it has a single question that is being asked at each decision node, and only 2 possible choices, at the very bottom of each decision tree, there is a single possible decision. In this example decision tree for whether to purchase a vehicle, the first question you ask is whether it has been road-tested. If it hasn't, you immediately decide not to buy, otherwise, you continue asking questions, such as what the vehicle's mileage is, and, if its age is old or recent. At bottom, every possible decision will eventually lead to a choice, some taking many fewer questions to get to those choices than others.

1. 1 https://www.ibm.com/support/knowledgecenter/en/SS3RA7\_15.0.0/ com.ibm.spss.modeler.help/nodes\_treebuilding.htm

**3. Decision trees as base learners**

The concept of a base learner will be covered more extensively later, but for now, just think of any individual learning algorithm in an ensemble algorithm as a base learner. This is important because XGBoost itself is an ensemble learning method in that it uses the outputs of many models for a final prediction. Anyway, as you saw in the previous slide, a decision tree is a learning method that involves a tree-like graph to model either a continuous or categorical choice given some data. It is composed of a series of binary decisions (yes/no or true/false questions) that when answered in succession ultimately yield a prediction about the data at hand (these predictions happen at the leaves of the tree).

**4. Decision trees and CART**

Decision trees are constructed iteratively (that is, one binary decision at a time) until some stopping criterion is met (the depth of the tree reaches some pre-defined maximum value, for example). During construction, the tree is built one split at a time, and the way that a split is selected (that is, what feature to split on and where in the feature's range of values to split) can vary, but involves choosing a split point that segregates the target values better (puts each target category into buckets that are increasingly dominated by just one category) until all (or nearly all) values within a given split are exclusively of one category or another. Using this process, each leaf of the decision tree will have a single category in the majority, or should be exclusively of one category.

**5. Individual decision trees tend to overfit**

Individual decision trees in general are low-bias, high-variance learning models.

1. 1 http://scott.fortmann-roe.com/docs/BiasVariance.html

**6. Individual decision trees tend to overfit**

That is, they are very good at learning relationships within any data you train them on, but they tend to overfit the data you use to train them on and usually generalize to new data poorly. XGBoost uses a slightly different kind of a decision tree,

1. 1 http://scott.fortmann-roe.com/docs/BiasVariance.html

**7. CART: Classification and Regression Trees**

called a classification and regression tree, or CART. Whereas for the decision trees described above the leaf nodes always contain decision values, CART trees contain a real-valued score in each leaf, regardless of whether they are used for classification or regression. The real-valued scores can then be thresholded to convert into categories for classification problems if necessary.

**8. Let's work with some decision trees!**

Awesome, let's get to working with some decision trees!

**Got It!**

**1. What is Boosting?**

Now that we've reviewed both supervised learning and the basics of decision trees, lets talk about the core concept that gives XGBoost its state-of-the-art performance, boosting.

**2. Boosting overview**

At bottom, boosting isn't really a specific machine learning algorithm, but a concept that can be applied to a set of machine learning models. So, its really a meta-algorithm. Specifically, it is an ensemble meta-algorithm primarily used to reduce any given single learner's variance and to convert many weak learners into an arbitrarily strong learner.

**3. Weak learners and strong learners**

A weak learner is any machine learning algorithm that is just slightly better than chance. So, a decision tree that can predict some outcome slightly more frequently than pure randomness would be considered a weak learner. The principal insight that allows XGBoost to work is the fact that you can use boosting to convert a collection of weak learners into a strong learner. Where a strong learner is any algorithm that can be tuned to achieve arbitrarily good performance for some supervised learning problem.

**4. How boosting is accomplished**

How is this accomplished? By iteratively learning a set of weak models on subsets of the data you have at hand, and weighting each of their predictions according to each weak learner's performance. You then combine all of the weak learners' predictions multiplied by their weights to obtain a single final weighted prediction that is much better than any of the individual predictions themselves. It's kind of incredible that this works as well as it does.

**5. Boosting example**

Here is a very basic example of boosting using 2 decision trees. This example comes from the XGBoost documentation and shows that given a specific example, each tree gives a different prediction score depending on the data it sees. The prediction scores for each possibility are summed across trees and the prediction is simply the sum of the scores across both trees. Here, you can see that whatever it was we were trying to predict, the little boy had a higher predicted score summed across both trees than the old man.

1. 1 https://xgboost.readthedocs.io/en/latest/model.html

**6. Model evaluation through cross-validation**

Since we will be working with XGBoost's learning API for model evaluation next, it's a good idea to briefly provide you with an example that shows how model evaluation using cross-validation works with XGBoost's learning API (which is different from the scikit-learn compatible API) as it has cross-validation capabilities baked in. As a refresher, cross-validation is a robust method for estimating the expected performance of a machine learning model on unseen data by generating many non-overlapping train/test splits on your training data and reporting the average test set performance across all data splits.

**7. Cross-validation in XGBoost example**

So, in lines 1 and 2 we import what we will be using. In line 3, we load in our example dataset. In line 4, we convert our dataset into an optimized data structure that the creators of XGBoost made that gives the package its lauded performance and efficiency gains called a DMatrix. In the previous exercise, the input datasets were converted into DMatrix data on the fly, but when we use the XGBoost cv object, which is part of XGBoost's learning api we have to first explicitly convert our data into a DMatrix. So, that's what we are doing here before we run our cross-validation. In line 5, we are creating a parameter dictionary to pass into our cross-validation. This is necessary because the cv method has no idea what kind of XGBoost model we are using and expects us to provide that information as a dictionary of appropriate key-value pairs. Our parameter dictionary here is bare-bones, only providing the objective function we would like to use and the maximum depth that every tree can grow to. In line 6, we call the cv method and pass in our DMatrix object storing all of our data, the parameter dictionary, the number of cross-validation folds, how many trees we want to build, what metric we want to compute, and whether we want our output to be stored as a pandas dataframe. In line 7 we simply convert our metrics into an accuracy and output the results to screen.

**8. Let's practice!**

Now it's your turn!

**Got It!**

**1. When should I use XGBoost?**

So, given everything we've said about XGBoost, when should (and shouldn't) you use it?

**2. When to use XGBoost**

Given that I've already talked a bit about when and where XGBoost shines, some of this shouldn't come as a surprise to you. You should consider using XGBoost for any supervised machine learning task that fits the following criteria: You have a large number of training examples. Although your definition of large can vary, I intend it to mean a dataset that has few features and at least 1000 examples. However, in general, as long as the number of features in your training set is smaller than the number of examples you have, you should be fine. Finally, XGBoost tends to do well when you have a mixture of categorical and numeric features, or when you have just numeric features.

**3. When to NOT use XGBoost**

When should you not use XGBoost? The most important kinds of problems where XGBoost is a suboptimal choice involve either those that have found success using other state-of-the-art algorithms or those that suffer from dataset size issues. Specifically, XGBoost is not ideally suited for image recognition, computer vision, or natural language processing and understanding problems, as those kinds of problems can be much better tackled using deep learning approaches. In terms of dataset size problems, XGBoost is not suitable when you have very small training sets ( less than 100 training examples) or when the number of training examples is significantly smaller than the number of features being used for training.

**4. Let's practice!**

Ok, let's finish off what you learned in chapter 1 with one last multiple choice question!

#### 02 - Regression with XGBoost

**Got It!**

**1. Regression review**

Congratulations on finishing chapter 1! Now that you've learned how to use XGBoost for classification, you'll learn how to use XGBoost for regression in this chapter.

**2. Regression basics**

Regression problems involve predicting continuous, or real, values. For example, if you're attempting to predict the height in centimeters a given person will be at 30 given some of their physical attributes at birth, you're solving a regression problem. Evaluating the quality of a regression model involves using a different set of metrics than those we described for use in classification problems in chapter 1.

**3. Common regression metrics**

In most cases, we use root mean squared error (RMSE) or the mean absolute error (MAE) to evaluate the quality of a regression model.

**4. Computing RMSE**

RMSE is computed by

**5. Computing RMSE**

taking the difference between the actual and the predicted values for what you are trying to predict,

**6. Computing RMSE**

squaring those differences, computing their mean, and taking that value's square root. This allows us to treat negative and positive differences equally, but tends to punish larger differences between predicted and actual values much more than smaller ones. MAE, on the other hand,

**7. Computing MAE**

simply sums the absolute differences between predicted and actual values across all of the samples we build our model on. Although MAE isn't affected by large differences as much as RMSE, it lacks some nice mathematical properties that make it much less frequently used as an evaluation metric.

**8. Common regression algorithms**

Some common algorithms that are used for regression problems include linear regression and decision trees. It's important to briefly note here that some algorithms,

**9. Algorithms for both regression and classification**

such as decision trees, can be used for both regression as well as classification tasks, which, as we will see, is one of their important properties that makes them prime candidates to be building blocks for XGBoost models.

1. 1 https://www.ibm.com/support/knowledgecenter/en/SS3RA7\_15.0.0/ com.ibm.spss.modeler.help/nodes\_treebuilding.htm

**10. Let's practice!**

Awesome, let's test your regression knowledge with a multiple choice questions.

**Got It!**

**1. Objective (loss) functions and base learners**

Let's talk a bit about objective functions and base learners so we can develop better intuitions about both concepts, as they are critical to understand in order for you to be able to grasp why XGBoost is such a powerful approach to building supervised regression models.

**2. Objective Functions and Why We Use Them**

An objective or loss function quantifies how far off our prediction is from the actual result for a given data point. It maps the difference between the prediction and the target to a real number. When we construct any machine learning model, we do so in the hopes that it minimizes the loss function across all of the data points we pass in. That's our ultimate goal, the smallest possible loss.

**3. Common loss functions and XGBoost**

Loss functions have specific naming conventions in XGBoost. For regression models, the most common loss function used is called reg linear. For binary classification models, the most common loss functions used are reg logistic, when you simply want the category of the target, and binary logistic, when you want the actual predicted probability of the positive class. So, in chapter 1, we were implicitly using the reg logistic loss function when building our classification models in XGBoost.

**4. Base learners and why we need them**

As mentioned before, XGBoost is an ensemble learning method composed of many individual models that are added together to generate a single prediction. Each of the individual models that are trained and combined are called base learners. The goal of XGBoost is to have base learners that is slightly better than random guessing on certain subsets of training examples, and uniformly bad at the remainder, so that when all of the predictions are combined, the uniformly bad predictions cancel out and those slightly better than chance combine into a single very good prediction. Let's look at a couple examples using trees and linear base learners in XGBoost.

**5. Trees as base learners example: Scikit-learn API**

Here's an example of how to train an XGBoost regression model with trees as base learners using XGBoost's scikit-learn compatible API. We will use the Boston Housing dataset from UCI's machine learning repository as an example. In lines 1-5 we import the libraries we need and load in the data. In lines 6 and 7, we convert our data into our X matrix and y vector and split into training and test sets as we've done before. In lines 8-10, we create our XGBoost regressor object, this time making sure we use the reg linear objective function, fit it to our training data, and generate our predictions on the test set.

**6. Trees as base learners example: Scikit-learn API**

And finally in lines 11 and 12 we compute the RMSE and print the result to screen.

**7. Linear base learners example: learning API only**

To use linear base learners, we have to use the learning API in XGBoost. Here's an example. In lines 1-7 we do what we did as before, loading in appropriate libraries and data. In lines 8 and 9 we convert our training and testing sets into DMatrix objects, as is required by the learning API. In line 10 we create a parameter dictionary explicitly specifying the base learner we want as gblinear, and the reg linear objective function we want to use. In lines 11-12 we train our model on the training set and generate predictions using the test set.

**8. Linear base learners example: learning API only**

In lines 13 and 14, we compute our rmse and print to screen, as we did before.

**9. Let's get to work!**

Ok, lets get to work!

**1. Regularization and base learners in XGBoost**

Loss functions in XGBoost don't just take into account how close a model's predictions are to the actual values,

**2. Regularization in XGBoost**

but also take into account how complex the model is. This idea of penalizing models as they become more complex is called regularization. So, loss functions in XGBoost are used to find models that are both accurate and as simple as they can possibly be. There are several parameters that can be tweaked in XGBoost to limit model complexity by altering the loss function. Gamma is a parameter for tree base learners that controls whether a given node on a base learner will split based on the expected reduction in the loss that would occur after performing the split, so that higher values lead to fewer splits. Alpha is another name for L1 regularization. However, this regularization term is a penalty on leaf weights rather than on feature weights, as is the case in linear or logistic regression. Higher alpha values lead to stronger L1 regularization, which causes many leaf weights in the base learners to go to 0. Lambda is another name for l2 regularization. L2 regularization is a much smoother penalty that l1 and causes leaf weights to smoothly decrease, instead of enforcing strong sparsity constraints on the leaf weights as in l1. If you're interested in learning more about regularization, check out DataCamp's Supervised Learning With Scikit Learn Course.

**3. L1 regularization in XGBoost example**

Let's look at an example of how you can tune one of these regularization parameters using XGBoost. As always, in lines 1-4 we import the necessary libraries, load in the data we will be working with, and create our feature matrix and target vectors X and y. In line 5 we convert our X matrix and y vector into a single optimized DMatrix object, and in line 6 we create our parameter dictionary that defines some required parameters for our learner. Specifically, we provide the loss function necessary for regression, and the maximum depth each tree base learner can have. In line 7 we create a list of 3 different l1 or alpha values that we will try, and in line 8 we initialize an empty list that will store our final root mean square error for each of these l1 or alpha values. Line 9 is actually a multi-line for loop where we iterate over each entry in our l1\_params list and do the following. First, we create a new key-value pair in our parameter dictionary that holds our current alpha value. We then run our XGBoost cross validation by passing in our DMatrix object, updated parameter dictionary, number of folds we want to cross-validate, number of trees we want as num\_boost\_round, the metric we want to compute, which is rmse, and that we want to output the results as a pandas DataFrame. In lines 10 and 11, we simply look at the final RMSE as a function of l1 regularization strength. At this point, we've talked about

**4. Base learners in XGBoost**

base learners and regularization quite a bit. Let's finish this off by comparing the two kinds of base learners that exist in XGBoost. The linear base learner is simply a sum of linear terms, exactly as you would find in a linear or logistic regression model. When you combine many of these base models into an ensemble, you get a weighted sum of linear models, which is itself linear. Since you don't get any nonlinear combination of features in the final model, this approach is rarely used, as you can get identical performance from a regularized linear model. The tree base learner uses decision trees as base models. When the decision trees are all combined into an ensemble, their combination becomes a nonlinear function of each individual tree, which itself is nonlinear. At this point, I want to briefly mention how you'll see

**5. Creating DataFrames from multiple equal-length lists**

Dataframes being created in the next couple exercises after you've computed your results. We will use both the zip and list function, one inside of the other, to convert multiple equal-length lists into a single object that we can convert into a pandas dataframe. Zip is a function that allows you to take multiple equal-length lists and iterate over them in parallel, side by side, as shown above. However, in python 3, zip creates a generator, or an object that doesn't have to be completely instantiated at runtime. In order for the entire zipped pair of lists to be instantiated, we have to cast the zip generator object into a list directly. After casting, we can convert this object directly into a dataframe. The point of all of this is, don't feel overwhelmed when you see this pattern in the following exercises, as its very useful and will only make you a stronger Python programmer and data scientist.

**6. Let's practice!**

Now, it's your turn to perform l2 regularization with an XGBoost model in the following exercises. In addition, you'll learn how to visualize feature importances in your model.

#### 03 - Fine-tuning your XG-Boost model

**1. Why tune your model?**

So far, you've learned how to use XGBoost to solve classification and regression problems. **Now, you'll learn how to supercharge those models by tuning them.** To motivate the reason behind this chapter on tuning your XGBoost model, let's just take a look at 2 cases, one where we take the simplest XGBoost model possible and compute a cross-validated RMSE, and then do the same exact thing with a tuned XGBoost model. What do you think the effect of model tuning on the overall reduction in RMSE will be?

**2. Untuned model example**

In lines 1-6, we simply load in the necessary libraries and ames housing data, and then convert our data into a DMatrix. In line 7, we create the most basic parameter configuration possible, only passing in the objective function we need to create a regression XGBoost model. This parameter configuration will be made much more complex as we tune our models. In fact, when performing parameter searches, we will use a dictionary that we typically call a parameter grid, because it will contain ranges of values over which we will search to find an optimal configuration. More on that later. In line 8, we run our cross-validation in XGBoost, passing in the simple parameter grid and telling it to run 4-fold cross validation, and to ouput the rmse as an evaluation metric. In line 9, we simply print the final rmse of the untuned model to screen, which is around 34600 dollars.

**3. Tuned model example**

Now let's take a look at a tuned example. Again, in lines 1-6, we load in the necessary libraries and ames housing data, and then convert our data into a DMatrix. In line 7, we create a more tuned parameter configuration, setting colsample\_bytree, learning\_rate, and max\_depth to better values. These are a few of the more important xgboost parameters that can be tuned, and you will learn more about and practice tuning these parameters later in this chapter. In line 8, we run our cross-validation in XGBoost, passing in our tuned parameter grid, as well as setting the number of trees to be constructed at 200, and again running 4-fold cross validation, and outputting the rmse as an evaluation metric. In line 9, we print the final rmse of the tuned model to screen, which is around 29800 dollars. That's an almost 14% reduction in RMSE!

**4. Let's tune some models!**

Now that you see that you can get a significant improvement in model performance by tuning an XGBoost model, let's have you start doing some tuning yourself!

#### Got It!

#### 1. Tunable parameters in XGBoost

#### Let's now go over the differences in what parameters can be tuned for each kind of base model in XGBoost. The parameters that can be tuned are significantly different for each base learner.

#### 2. Common tree tunable parameters

#### For the tree base learner, which is the one you should use in almost every single case, the most frequently tuned parameters are outlined below. The learning rate affects how quickly the model fits the residual error using additional base learners. A low learning rate will require more boosting rounds to achieve the same reduction in residual error as an XGBoost model with a high learning rate. Gamma, alpha, and lambda were described in chapter 2 and all have an effect on how strongly regularized the trained model will be. Max\_depth must a positive integer value and affects how deeply each tree is allowed to grow during any given boosting round. Subsample must be a value between 0 and 1 and is the fraction of the total training set that can be used for any given boosting round. If the value is low, then the fraction of your training data used would per boosting round would be low and you may run into underfitting problems, a value that is very high can lead to overfitting as well. Colsample\_bytree is the fraction of features you can select from during any given boosting round and must also be a value between 0 and 1. A large value means that almost all features can be used to build a tree during a given boosting round, whereas a small value means that the fraction of features that can be selected from is very small. In general, smaller colsample\_bytree values can be thought of as providing additional regularization to the model, whereas using all columns may in certain cases overfit a trained model.

#### 3. Linear tunable parameters

#### For the linear base learner, the number of tunable parameters is significantly smaller. You only have access to l1 and l2 regularization on the weights associated with any given feature, and then another regularization term that can be applied to the model's bias. Finally, its important to mention that the number of boosting rounds (that is, either the number of trees you build or the number of linear base learners you construct) is itself a tunable parameter.

#### 4. Let's get to some tuning!

#### Now that we've covered the parameters that are usually tuned when using XGBoost, lets get to some tuning!

#### Got It!

#### 1. Review of grid search and random search

#### How do we find the optimal values for several hyperparameters simultaneously, leading to the lowest loss possible, when their values interact in in non-obvious, non-linear ways? Two common strategies for choosing several hyperparameter values simultaneously are Grid Search and Random Search, so it's important that we review them here, and see what their advantages and disadvantages are, by looking at some examples of how both can be used with the XGBoost and scikit-learn packages.

#### 2. Grid search: review

#### Grid Search is a method of exhaustively searching through a collection of possible parameter values. For example, if you have 2 hyperparameters you would like to tune, and 4 possible values for each hyperparameter, then a grid search over that parameter space would try all 16 possible parameter configurations. In a grid search, you try every parameter configuration, evaluate some metric for that configuration, and pick the parameter configuration that gave you the best value for the metric you were using, which in our case will be the root mean squared error.

#### 3. Grid search: example

#### Let's go over an example of how to grid search over several hyperparameters using XGBoost and scikit learn. In lines 1-4 we load in the necessary libraries, including GridSearchCV from sklearn dot model\_selection. In lines 5-7 we load in our dataset and convert it into a DMatrix. In line 8 we create our grid of hyperparameters we want to search over. We selected 4 different learning rates (or eta values), 3 different subsample values, and a single number of trees. The total number of distinct hyperparameter configurations is 12, so 12 different models will be built. In line 9 we create our regressor, and then in line 10 we pass the xgbregressor object, parameter grid, evaluation metric, and number of cross validation folds to GridSearchCV and then immediately fit that gridsearch object in line 11, just like every other scikit learn estimator object we've done this to in the past. In line 12, having fit the gridsearch object, we can extract the best parameters the grid search found, and print them to the screen. In line 13, we get the RMSE that corresponds to the best parameters found, and see that it's ~28500 dollars.

#### 4. Random search: review

#### Random search is significantly different from grid search in that the number of models that you are required to iterate over doesn't grow as you expand the overall hyperparameter space. In random search, you get to decide how many models, or iterations, you want to try out before stopping. Random search simply involves drawing a random combination of possible hyperparameter values from the range of allowable hyperparameters a set number of times. Each time, you train a model with the selected hyperparameters, evaluate the performance of that model, and then rinse and repeat. When you've created the number of models you had specified initially, you simply pick the best one. To finish this lesson off,

#### 5. Random search: example

#### let's look at a full random search example. In lines 1-7, we load in the necessary modules, this time loading in RandomizedSearchCV from sklearn dot model\_selection, and then load in and convert the data we need to a DMatrix object as always. In line 8 we create our parameter grid, this time generating a large number of learning rate values and subsample values using np-dot-arange. There are 20 values for learning\_rate (or eta) and 20 values for subsample, which would be 400 models to try if we were to run a grid search (which we aren't doing here). In line 9 we create our xgbregressor object, and in line 10 we create our RandomizedSearchCV object, passing in the xgbregressor and parameter grid we had just created. We also set the number of iterations we want the random search to proceed to 25, so we know it will not be able to try all 400 possible parameter configurations. We also specify the evaluation metric we want to use, and that we want to run 4-fold cross-validation on each iteration. In line 11 we fit our randomizedsearchcv object, which can take a bit of time. Finally, lines 12 and 13 print the best model parameters found, and the corresponding best RMSE.

#### 6. Let's practice!

#### Ok, now let's have you practice both grid search and random search in the following exercises.

**1. Limits of grid search and random search**

Now that you've done both GridSearch and RandomSearch for hyperparameter tuning on the Ames housing data, let's briefly go over the limits of both of these approaches for hyperparameter tuning.

**2. Grid search and random search limitations**

It should be clear to you that grid search and random search each suffer from distinct limitations. As long as the number of hyperparameters and distinct values per hyperparameter you search over is kept small, grid search will give you an answer in a reasonable amount of time. However, as the number of hyperparameters grows, the time it takes to complete a full grid search increases exponentially. For random search, the problem is a bit different. Since you can specify how many iterations a random search should be run, the time it takes to finish the random search wont explode as you add more and more hyperparameters to search through. The problem really is that as you add new hyperparameters to search over, the size of the hyperparameter space explodes as it did in the grid search case, and so you are left hoping that one of the random parameter configurations that the search chooses is a good one! You can always increase the number of iterations you want the random search to run, but then finding an optimal configuration becomes a combination of waiting randomly finding a good set of hyperparameters. In any case, both approaches have significant limitations.

**3. Let's practice!**

Great, now that you've learned how to tune the most important hyperparameters found in XGBoost, lets move onto the final chapter, where we work through 2 end to end processing pipelines utilizing XGBoost and scikit-learn.

#### 04 - Using XGBoost in pipelines

#### Got It!

#### 1. Review of pipelines using sklearn

#### Let's begin the final chapter in this course by reviewing how pipelines are used in scikit-learn. Refreshing our memory about how pipelines work will allow us to use XGBoost effectively in pipelines going forward. Before working through an example script using pipelines, lets briefly go over how they work.

#### 2. Pipeline review

#### Pipelines in sklearn are objects that take a list of named tuples as input. The named tuples must always contain a string name as the first element in each tuple and any scikit-learn compatible transformer or estimator object as the second element. Each named tuple in the pipeline is called a step, and the list of transformations that are contained in the list are executed in order once some data is passed through the pipeline. This is done using the standard fit/predict paradigm that is standard in scikit-learn. Finally, where pipelines are really useful is that they can be used as input estimator objects into other scikit-learn objects themselves, the most useful of which are the cross\_val\_score method, which allows for efficient cross-validation and out of sample metric calculation, and the grid search and random search approaches for tuning hyperparameters.

#### 3. Scikit-learn pipeline example

#### Now that we've talked about how pipelines work, lets seem them in action. In this example, we will use the Boston Housing dataset. As you've seen many times before, we first import all of the functionality we will need for the example. We will use a randomforestregressor model to predict housing prices, and will import pipeline from sklearn's pipeline submodule. In lines 2-4, we load in our data and create our X feature matrix and y target vector. Lines 5-6 are the ones that do the real work here. In line 5, we create our pipeline, which contains a standardscaler transformer followed by our RandomForestRegressor estimator. Line 6 takes the just created pipeline estimator as an input along with our X matrix and y vector and performs 10-fold cross-validation using the pipeline and the data and outputs the neg\_mean\_squared\_error as an evaluation metric once per fold. As a brief aside, neg\_mean\_squared\_error is scikit-learn's API-specific way of calculating the mean squared error in an API-compatible way. Negative mean squared errors don't actualy exist as all squares must be positive when working with real numbers.

#### 4. Scikit-learn pipeline example

#### Thus, in lines 7 and 8 we simply take the absolute value of the scores, take each of their square roots, and compute their mean to get a root mean squared error across all 10 cross-validation folds. We can see that on average our prediction was off by about 4-point-5 units. In the following exercises, because we will be working with the Ames housing dataset, which is more complex than the Boston housing dataset,

#### 5. Preprocessing I: LabelEncoder and OneHotEncoder

#### some additional preprocessing steps will be required. Specifically, we will do the same preprocessing steps in two different ways, only one of which can be done within a pipeline. The first approach involves using the LabelEncoder and OneHotEncoder classes of scikit-learn’s preprocessing submodule one after the other. LabelEncoder simply converts a categorical column of strings into integers that map onto those strings. OneHotEncoder takes a column of integers that are treated as categorical values, and encodes them as dummy variables, which you may already be familiar with. The problem with this 2-step method, however, is that it cannot currently be done within a pipeline. However, not all hope is lost. The second approach,

#### 6. Preprocessing II: DictVectorizer

#### which involves using a dict-vectorizer, can accomplish both steps in one line of code.The DictVectorizer is a class found in scikit-learn’s feature extraction submodule, and is traditionally used in text processing pipelines by converting lists of feature mappings into vectors. Using pandas DataFrames, we don’t initially have such a list, however, if we explicitly convert a DataFrame into a list of dictionary entries, then we have exactly what we need. For more details on these classes, I encourage you to explore the scikit-learn documentation.

#### 7. Let's build pipelines!

#### You will use both approaches in the next few exercises. I hope you have fun building pipelines!

#### 1. Incorporating xgboost into pipelines

#### Now that you've had some practice using pipelines in scikit-learn, let's see what it takes to use xgboost within pipelines.

#### 2. Scikit-learn pipeline example with XGBoost

#### This example is very similar to what was shown in the pipeline review that began this chapter. To get XGBoost to work within a pipeline, all that's really required is that you use XGBoost's scikit-learn API within a pipeline object. Let's see what that looks like in practice. As always, we first import everything we need for our purposes. We then proceed to load in the dataset and parse it into the matrix of features X and target vector y. At this point lies the only difference between using a scikit-learn native machine learning model and XGBoost. Specifically, we simply pass in an instance of the XGBoost XGBRegressor object into the pipeline where a normal scikit-learn estimator would be. The rest of the script is exactly what you've seen in the past. You compute the cross-validated negative mse using 10-fold cross-validation and then convert the 10-fold negative MSE into an average RMSE across all 10 folds. As you can see, without any hyperparameter tuning, the XGBoost model had a lower RMSE, of ~4-point-03 units, than the randomforest model we started the chapter with, which had an RMSE around 4-point-5.

#### 3. Additional components introduced for pipelines

#### We wanted you to see how a simple case of pipelining with XGBoost works, however, in the final end-to-end example, we will take a dataset that involves significantly more wrangling before it can be used with XGBoost and put it through a pipeline as well. As a result, we will have to work with a library that is not part of the standard suite of scikit-learn tools, as well as work with parts of pipelines that you may not be familiar with. Sklearn\_pandas is a separate library that attempts to bridge the gap between working with pandas and working with scikit-learn, as they don't always work seamlessly together. Specifically, sklearn\_pandas has a generic class called DataFrameMapper, that allows for easy conversion between scikit-learn aware objects, or pure numpy arrays, and the DataFrames that are the bread and butter of the pandas library. Additionally, we will use a class called CategoricalImputer that will allow us to impute missing categorical values directly, without having to first convert them to integers, as is the requirement in scikit-learn. We will also use some uncommon aspects of scikit-learn to accomplish our goals. Specifically, we will use the Imputer class from scikit-learn's preprocessing submodule, that allows us to fill in missing numerical values, and the FeatureUnion class found in scikit-learn's pipeline submodule. The FeatureUnion class allows us to combine separate pipeline outputs into a single pipeline output, as for example, we would need to do if we had one set of preprocessing steps we needed to perform on the categorical features of a dataset and a distinct set of preprocessing steps on the numeric features found in a dataset. The point is, we will introduce these topics at once, but don't want you to feel overwhelmed about what they are doing and how they can be used properly.

#### 4. Let's practice!

#### Hopefully, you just saw that its not particularly difficult to incorporate XGBoost into pipelines. Now, its your turn to practice what you just learned!

**Got It!**

**1. Tuning xgboost hyperparameters in a pipeline**

We are going to finish off this chapter, and the course, by seeing how automated hyperparameter tuning for an XGBoost model works within a scikit-learn pipeline. Once you have this down, you'll be able to make some of the most powerful well-tuned machine learning models today in an automated, reproducible manner.

**2. Tuning XGBoost hyperparameters in a pipeline**

We will again use the Boston housing dataset to motivate our use of pipelines and hyperparameter tuning. As always, we first import what we will be using. The only difference is now we also import RandomizedSearchCV from the scikit-learn modelselection submodule. We then load in our data and create our feature matrix X and target vector y and also create our pipeline that includes both the standard scaling step and a base xgboostregressor object with all default parameters. At this point, you need to create the grid of parameters over which you will search. In order for the hyperparameters to be passed to the appropriate step, you have to name the parameters in the dictionary with the name of the step being referenced followed by 2 underscore signs and then the name of the hyperparameter you want to iterate over. Since the xgboost step is called xgb\_model, all of our hyperparameter keys will start with xgboost\_model\_\_. In the example, we will tune subsample, max\_depth, and colsample\_bytree, and give each parameter a range of possible values. We then pass the pipeline in as an estimator to RandomizedSearchCV and the parameter grid to param\_distributions. Everything else is as you've seen before, with appropriate scoring and cross-validation parameters passed in as well. Once that's done all you need to do is fit the randomizedsearch object and pass in the X and y objects we created earlier.

**3. Tuning XGBoost hyperparameters in a pipeline II**

Finally, once you've fit the randomizedsearchcv object, you can inspect what the best score it found was, and convert it to an RMSE. You can also inspect what the best model found was and print it to screen.

**4. Let's finish this up!**

Ok, last coding exercise of the course, let's finish this up!

**Got It!**

**1. Final Thoughts**

Congratulations on completing this course. Let's go over everything we've covered in this course, as well as where you can go from here with learning other topics related to XGBoost that we didn't have a chance to cover.

**2. What We Have Covered And You Have Learned**

So, what have we been able to cover in this course? Well, we've learned how to use XGBoost for both classification and regression tasks. We've also covered all the most important hyperparameters that you should tune when creating XGBoost models, so that they are as performant as possible. And we just finished up how to incorporate XGBoost into pipelines, and used some more advanced functions that allow us to seamlessly work with Pandas DataFrames and scikit-learn. That's quite a lot of ground we've covered and you should be proud of what you've been able to accomplish.

**3. What We Have Not Covered (And How You Can Proceed)**

However, although we've covered quite a lot, we didn't cover some other topics that would advance your mastery of XGBoost. Specifically, we never looked into how to use XGBoost for ranking or recommendation problems, which can be done by modifying the loss function you use when constructing your model. We also didn't look into more advanced hyperparameter selection strategies. The most powerful strategy, called Bayesian optimization, has been used with lots of success, and entire companies have been created just for specifically using this method in tuning models (for example, the company sigopt does exactly this). It's a powerful method, but would take an entire other DataCamp course to teach properly! Finally, we haven't talked about ensembling XGBoost with other models. Although XGBoost is itself an ensemble method, nothing stops you from combining the predictions you get from an XGBoost model with other models, as this is usually a very powerful additional way to squeeze the last bit of juice from your data. Learning about all of these additional topics will help you become an even more powerful user of XGBoost. Now that you know your way around the package, there's no reason for you to stop learning how to get even more benefits out of it.

**4. Congratulations!**

I hope you've enjoyed taking this course on XGBoost as I have teaching it. Please let us know if you've enjoyed the course and definitely let me know how I can improve it. It's been a pleasure, and I hope you continue your data science journey from here!