Machine Learning Foundations Unit 6: Improve Performance With Ensemble Methods

Table of Contents

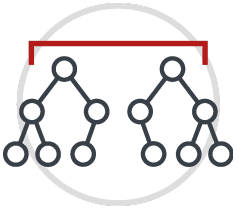
- Watch: Gradient Boosted Decision Trees
- Read: Formalizing Gradient Boosted Decision Trees
- Watch: GBDT Optimization
- Assignment: Building Gradient Boosted Decision Trees
- Tool: GBDT Cheat Sheet
- Ask the Expert: Mehrnoosh Sameki on Addressing Overfitting and Underfitting
- Assignment: Unit 6 Assignment - Comparing Regression Models
- Assignment: Unit 6 Assignment — Written Submission
- Module Wrap-up: Understand Gradient Boosting
- Assignment: Lab 6 Assignment

Unit 6 Overview
Video Transcript

In this unit, we will cover new supervised learning algorithms that can be quite powerful in improving predictive performance and model generalization. These are known as ensemble methods. Ensemble methods are a class of techniques that train multiple models and aggregate them into a single prediction. We will focus on three different ensemble modeling techniques called stacking, bagging, and boosting. Stacking is a more general procedure that doesn't have a specific supervised learning method attached to it. On the other hand, we'll present random forests as a specific implementation of bagging, and gradient-boosted decision trees as a specific implementation of boosting. For each algorithm, I will walk through the typical steps used in constructing the ensemble and also provide code examples and guidelines for how to get the best ensemble predictor from your data.

What you'll do:

- **Explore the bias-variance tradeoff**
- **Improve model performance with ensemble methods**
- **Understand the mechanics of random forests and gradient boosted decision trees**
- **Identify differences among ensemble methods and when to use them**
- **Navigate design decisions and constraints to perform agile model development**
- **Build and tune different models to see how various methods can improve models**



Unit Description

Ensemble modeling is a helpful and important technique used in machine learning. It's a powerful approach to train multiple models and quantify them into a single prediction. There are three commonly used ensemble techniques: stacking, bagging, and boosting. So how do you know which ensemble method to use and when to use it?

In this unit, Mr. D'Alessandro will walk you through the stacking, bagging, and boosting techniques, providing the motivation behind using each and explaining when to use one over the other. There are many tradeoffs with each method due to the fact that each improves results differently. By the end of the unit, you will have observed a number of robust algorithms, such as random forests and gradient boosted decision trees, that employ these methods. You will also have the opportunity to put this new knowledge into action when you practice building and optimizing various ensemble models.

Please note that code snippets in videos may vary from the code in assignments and exercises.

Back to Table of Contents

Tool: Unit 6 Glossary

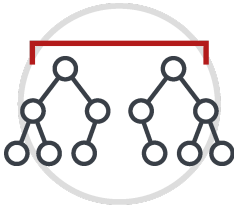
Though most of the new terms you will learn about throughout the unit are defined and described in context, there are a few vital terms that are likely new to you but undefined in the course videos. While you won't need to know these terms until later in the unit, it can be helpful to glance at them briefly now. Click on the link to the right to view and download the new terms for this unit.

Download the Tool

Use this **Unit 6 Glossary** as a reference as you work through the unit and encounter unfamiliar terms.

Back to Table of Contents

Module Introduction: Use Ensembles to Improve Models



In this module, Mr. D'Alessandro introduces an important technique in machine learning called ensemble modeling. There are three commonly used ensemble techniques known as stacking, bagging, and boosting. Mr. D'Alessandro will explain the motivation behind ensemble modeling and how it is used to reduce model estimation errors due to bias and variance. In addition to gaining a high-level overview of each method, you will also see how stacking is implemented in detail.

Back to Table of Contents

Watch: Ensemble Modeling

In order to achieve good generalization, it is essential that a model has low model estimation bias and variance. A technique commonly used to achieve this is ensemble modeling. In this video, Mr. D'Alessandro gives an overview of what ensemble modeling is and how it uses a combination of multiple models to reduce the model estimation bias and variance.

Video Transcript

Now, let's introduce a new approach to model building, which is a technique called ensemble modeling. Ensemble modeling simply involves combining multiple independent models that are predicting the same outcome, and this technique is applicable to both classification and regression problems. The specific models that we combine and the way we combine them form the basis of the different approaches we'll cover. Before presenting specific techniques, though, I want to first present some of the intuition for why we're doing this. Our motivation for combining multiple models into a single prediction stems from model estimation and bias-variance tradeoff. I'll review these concepts again so they're top of mind. Every model's error will be composed of its estimation bias and its estimation variance. Estimation bias is when a model is too rigid to adapt to nuances of the data and estimation variance is the opposite. The model has higher complexity, which makes it flexible and too sensitive often to nuances in the data. The exact contribution of each term will depend on the chosen model. For instance, a logistic regression typically has higher bias, but lower variance, and a decision tree would have the opposite. Every algorithm is going to have its own estimation bias and variance tradeoffs. Even with a single algorithm like decision trees, different hyperparameters that we choose for a specific tree will also have their own bias and variance tradeoffs. Let's assume we had a given problem, and instead of building one model, we built a set of models and we varied aspects of the training process like which algorithms, features, hyperparameters, and even subsets of the training data we'll use. Next, we make a prediction on a single example where we scored this example against all of the models that we have built. Instead of a single prediction, we would get a distribution of predictions like the one similar to this histogram. Here, the green line represents our truth, and the red lines

represent the predictions for four different models. If we average the predictions of these four different models, we'd have the correct prediction, even though each individual prediction was wrong. This process of building multiple models, each with its own bias and variance tradeoff, and then averaging predictions is ensemble modeling in action. When we build an ensemble of models, we choose models with different levels of bias and variance. In some methods, we even manipulate the data to create variations in what different models learn. In the coming lectures, I'll present specific examples of how this is done. With sufficient variety captured across the set of models in an ensemble, our errors will typically cancel out when we combine the predictions. This process, in theory, should ensure that our final prediction is closer to the ground truth than if we were to rely on one single model.

Back to Table of Contents

Tool: Model Debugging Cheat Sheet

This tool provides an overview of the symptoms of high bias and high variance in your models. You'll find information that will help you determine the cause as well as potential solutions to help debug and improve your models.

Back to Table of Contents

Download the Tool

Use this **Model Debugging Cheat Sheet** to help you determine the source and remedy for your model's error.

Ask the Expert: Mehrnoosh Sameki on Bias-Variance Tradeoff

In this video, Dr. Sameki revisits what bias and variance errors are and explains the bias-variance tradeoff.

Note: The job title listed below was held by our expert at the time of this interview.

Mehrnoosh Sameki
Product Manager, Microsoft

Dr. Mehrnoosh Sameki is a Senior Technical Program Manager at Microsoft, responsible for leading the product efforts on machine learning interpretability and fairness within the Open Source and Azure Machine Learning platforms. Dr. Sameki also co-founded Fairlearn and Responsible-AI-widgets and has been a contributor to the InterpretML offering.

Question

What is bias, variance, and the bias-variance tradeoff?

Video Transcript

Bias is the difference between the average prediction of our machine learning model and the correct value, which we're trying to predict. Model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to a high error rate on training and test data. Now, variance is the variability of a model prediction for a given data point or a value, which tells us the spread of our data. Model with high variance pay a lot of attention to the training data and do not generalize on the data which they haven't seen before. And so as a result, such models perform very well on training but have high error rates on test data. Now let's talk about what is this bias-variance trade-off. First of all, let's start by saying that each model that we are training might have a set of parameters that are internal to the model. That is, they're learned or estimated purely from the data during training as the algorithm used tries to learn the mapping between the input features and the labels or targets. Now, if our model is too simple and has very few parameters, then it may have high bias and low variance. On the other hand, if our model has large

number of parameters, then it's going to have high variance and low bias. So we need to find that sweet spot or good balance without overfitting and underfitting the data.

Back to Table of Contents

Watch: Types of Ensemble Methods

In this video, Mr. D'Alessandro introduces three commonly used ensemble techniques: stacking, bagging, and boosting. In stacking, you are combining multiple models and aggregating their output prediction using methods such as taking the average. In bagging, you are using one algorithm but with randomly sampled data with replacement. In boosting, you are building models that aim to reduce errors over multiple iterations.

Video Transcript

We recently introduced that model ensembling is the process of combining different models in an effort to average out the errors that individual models might exhibit. Now, I'll introduce three high-level methods for building ensemble models. These three common ensemble methods are stacking, bagging, and boosting. Bagging and boosting are techniques that are employed by specific machine learning algorithms that focus on building an ensemble of the same type of model, which is usually decision trees. We will introduce you to the bagging algorithm — random forest, and the boosting algorithm — gradient-boosted decision trees. An implementation of stacking, however, can be done using a combination of any common supervised learning algorithms that you've already learned, such as logistic regression or decision trees. Let's start our discussion with stacking. Given a training data set, we would build a varied set of models. We can mix the underlying modeling algorithms, such as using a logistic regression, decision trees, and even K nearest neighbors. Then, when it comes to prediction time, we run the prediction example through each model, and take some aggregation of the scores. The simplest aggregation approach is to just average the predictions. Building from that, we can weigh each prediction separately and take a weighted average over those weighted predictions. But to get to the best performance, we can even take a supervised learning approach where we train a two-stage model to learn these optimal weights. The next method is called bagging, which is a shortening of the phrase bootstrap aggregating, and this is the technique that powers the random forest algorithm. This method relies on this technique called bootstrapping, where bootstrapping is a process where we take multiple different samples from a data set, compute some quantity or statistic on each sample, and

then average them to get our final estimate. For bootstrapping in model building, we run a loop, and in each loop, we sample with replacement from our training data to create a new training set from the original data. Sampling with replacement means we can sample the same example multiple times. We then build a model from the bootstrap sample. For each iteration, we would use the same modeling algorithm. For instance, in random forest, we would build a decision tree at each iteration. After the loop is finished, we will have a collection of models. When we make a prediction, we run an example through all of the models in this collection and simply take the average prediction across these models. The last technique we'll cover is called boosting. This method is the most mathematically rigorous, but it's also quite intuitive. First, we would build an initial model as usual, then we initiate a loop that performs the following procedures. We take the model predictions from the current iteration and compute the error between the prediction and the label. We then build a new model at each iteration that predicts the error instead of the original label, and we add that to the collection of models. After doing this n times, we have an ensemble of models where each one aims to reduce the prediction error. The most common algorithm we'll use to implement this technique is called gradient-boosted decision trees. As is suggested by the name, decision trees are the base algorithm that is used to create this ensemble. If any of these overviews were too high level or fast, don't worry. We will cover each of these methods with multiple videos and provide both technical detail and illustrations to make each method more intuitive.

Back to Table of Contents

Quiz: Check Your Knowledge: Ensemble Methods

You may take this quiz up to three times.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.

Back to Table of Contents

Watch: Stacking

Follow along as Mr. D'Alessandro describes two separate approaches typically used to build a stacking ensemble model. The first approach is using simple aggregation of multiple models. The second approach is to use a two-stage process, which first uses simple aggregation then subsequently uses supervised learning. Mr. D'Alessandro discusses the intuition behind how ensemble models work by smoothing out the variance among multiple models.

Video Transcript

Now, let's go a little deeper on the ensembling technique called stacking. Of the three ensemble methods — stacking, bagging, and boosting — stacking is the one that doesn't have a specific algorithmic implementation. I'll give an overview of this method and then share some nuances on how to train a stacked ensemble model. The stacking procedure is commonly done in one of two ways. First, we start by training k different models where we typically use a variety of methods. In this picture, I illustrate this first step with three different models, but in reality, you can choose as many as you wish. Also, we can use any type of model here. Now, the second step is applied when we make a prediction. Here, I show it mathematically. This equation shows a weighted sum of s different models, with each model represented by the f of x symbol. The aggregation method is very straightforward. This is just a weighted sum of the individual model predictions. The nuance, though, is in how you determine the weights. The simplest approach is to weight each algorithm equally, which amounts to taking an average of the individual predictions. We can also take a supervised learning approach where we train another model to set the weights. The supervised learning approach to setting weights is a two-stage process. The diagram here lays this out, and I'll walk us through it. The first stage is the core stacking approach we just presented. Starting with our original training data, we build a set of models. Any modeling algorithm will work here, but the key is to have models with sufficient diversity. You might want to include simpler models like logistic regression and then decision trees with different depths, for instance. Then to start the second stage, we build a new training data set where the individual model predictions from step one are the features, and the label is from the original data set. Then the second stage

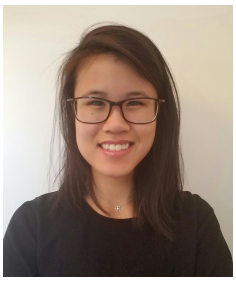
would be to train another model on this new data set. Any type of algorithm works so long as it is appropriate for the label. In practice, simpler linear models tend to be favored for the second stage. The result would be a two-stage prediction model. Scikit-learn offers a class for running this entire procedure in its ensembles package, so while it is good to understand the method overall, in practice, you don't have to implement it from scratch. I also want to share a bit of insight as to why stacking helps. Now, these insights actually apply to all ensembles in general. Here's an example scatterplot of points colored by their label. The lines represent classification boundaries of some learned model. The green line shows a classifier that is very over fit. Now imagine we had a bunch of different models where each one over fit, but in different ways. By averaging these models, we smooth out the green lines here and end up with something more like the black line. Taking weighted sums of individual models increases the chances that we can cancel out the error of any individual model, and then this would arrive at a model that generalizes better. But for this process to work, there is one fundamental rule we need to follow. It is very important that each individual model in this sum is different. If we trained the exact same algorithm on the same data using the same hyperparameters and the same features, each model would likely produce the same predictions. This means that our model errors likely would not cancel each other out. We would be incurring the cost of training many models, but without any of the benefit. So to get more independence and variety in our base models for stacking, we need to vary the algorithm, the features, and the hyperparameters used. When I cover random forests later, you will see that we also induce more variation in the base models by also varying the training data at each iteration. With sufficient variation in the base models, the stacking procedure should produce better results for you.

Back to Table of Contents

Ask the Expert: Annie Wong on Using Ensemble Models

How do you know when to use an ensemble model to solve your ML problem? Follow along as Ms. Wong describes when you should use ensemble models over models, as well as the advantages and disadvantages of using them. Ms. Wong also recounts a time when she used an ensemble model.

Note: The job title listed below was held by our expert at the time of this interview.



Annie Wong
Senior Manager - Data Science, Verizon Wireless

Annie Wong is a Senior Manager leading a team of data scientists at Verizon. Ms. Wong works with the business to utilize data more in their day-to-day decision making. Technology is exciting for her because it is constantly growing and that enables her to learn every day!

Question 1

How do you know when to use an ensemble model over another model?

Video Transcript

So there's a bunch of things to consider. I'm going to name two things. The first one is high variance in your data or a bias. If a single model is over fitting or under fitting, you can use an ensemble model. For example, if you combine multiple models that have been trained on different sets inside your data set, it can help you reduce some of that variance. And the other piece is really thinking about performance when considering what you're optimizing for, like accuracy, precision, or recall. If the performance of one model is just not working, an ensemble model might be a good approach. So I guess to summarize, ensemble models are great with diversity in data sets and driving diversity in models because there's one, if the training data set is too diverse, ensembles can capture the complexity and more of the nuances there. And then also, the diversity in models when they aren't too similar can definitely help with improving performance.

Question 2

Can you provide an example of when you chose to use an ensemble model?

Video Transcript

Great concrete example. Our team actually recently worked on this text classification problem. Essentially, we were just trying to determine whether a tweet or comment about Verizon was network-related, billing-related, promotion-related, or something else. We tried a bunch of different algorithms and each had their strengths and weaknesses. So just to give you a couple of examples, for logistic regression, it performed really well on tweets and short comments, and another example is support vector machine performed well on multi-issue or category tweets and comments. So we essentially took note of these kind of pros and cons of each algorithm, and once that team understood each model's strengths and weaknesses, we used an ensemble model with a basically simple voting approach where each model's prediction was weighed and kind of that final prediction was based on a majority vote.

Question 3

What are the advantages and disadvantages of using an ensemble model?

Video Transcript

Ensemble models have several advantages. They help improve performance and robustness. They have the ability to deal with diverse data sets, as mentioned earlier. Disadvantages, though, come in that complexity. Training time of the model. It can be a lot more computationally expensive, requiring a lot more resources and being able to interpret the model may be a little bit less straightforward. While ensemble models can provide interpretability, that final prediction might be a little bit more challenging to understand than maybe using one single model.

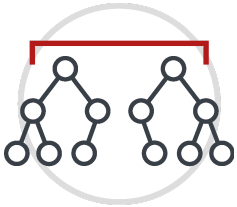
Back to Table of Contents

Module Wrap-up: Use Ensembles to Improve Models

In this module, Mr. D'Alessandro presented an overview of three commonly used ensemble modeling approaches: stacking, bagging, and boosting. While bagging and boosting are typically implemented with the random forest algorithm and the gradient boosted decision tree algorithm, respectively, the stacking method may be implemented with any supervised learning algorithms. In addition to the overview, Mr. D'Alessandro took a deep dive into the implementation of stacking, where he explained the two approaches used as well as the pros and cons of these approaches.

Back to Table of Contents

Module Introduction: Understand Random Forest



In this module, Mr. D'Alessandro introduces the random forest algorithm which uses the bagging ensemble method to reduce model estimation variance. A random forest model is comprised of multiple decision trees. You will discover how to make each tree within a random forest sufficiently different by using random sampling and by using only a set of all features during training. Finally, Mr. D'Alessandro will describe the tradeoff between performance and scalability.

Back to Table of Contents

Watch: Random Forest Overview

In this video, Mr. D'Alessandro goes over the detailed approach of implementing a random forest. A random forest is comprised of multiple decision trees of the same configuration. In a random forest model, the training data is sampled at random with replacement. Follow along as Mr. D'Alessandro explains how to make predictions with a random forest in both classification and regression settings.

Video Transcript

We are now going to discuss the second ensemble modeling method, bagging. In the next few videos, we're going to focus on random forests, which is the most popular ensemble algorithm that uses the bagging technique. At a high level, bagging helps because taking averages over different but similar models is an effective way to reduce a model's overall estimation variance. Bagging is one of the primary ways that random forests get their effectiveness. There is a bit more to random forests than just the use of bagging, though. I'm going to present a general overview of the structure of random forests and how the prediction works. One of my favorite things about the random forest approach to ensemble modeling is its simplicity. Most of the sophistication of the algorithm is in learning a decision tree. The rest is mathematically simple and pretty intuitive. The random forest is nothing more than a set of decision trees. Each decision tree is expected to be different because we vary the training data by using different examples and subsets of the features. When training the forest though, we typically use the same exact configuration of decision tree hyper parameters. A typical random forest can consist of dozens to hundreds of different trees. The total number of trees is one of the primary hyper parameters you'll consider, and you'll find the right now using standard model selection techniques. Once your random forest is trained, the model object will store the collection of decision trees. To make a prediction, you input your feature vector into each tree to get a set of individual predictions. In each individual tree, the standard rules of decision trees apply. If this is a classification task, the individual trees can output either a class label or probability of belonging to a specific class labeled one. If this is a regression, the output would be the average value of the label. The final prediction will be a simple average of the individual decision tree predictions. If we want a class label

instead of a probability for classification problems, we can use the most common class label amongst the set of trees as the prediction. The key thing to remember is that this is just a set of decision trees, so anything that decision trees can do, random forest can also do, but often with better results.

Back to Table of Contents

Watch: Bagging (Bootstrap Aggregation)

In this video, Mr. D'Alessandro discusses how the random forest algorithm implements bagging, which is also known as bootstrap aggregation, and explains the bootstrap sampling technique. He presents several plots to illustrate the idea of bagging and how it is used to average out the variance of multiple models.

Video Transcript

Now, I'm going to cover the bagging procedure in more detail with an emphasis on how it is used in the random forest. The word bagging itself might seem like a strange choice of word, but it actually is a contraction of the two words bootstrap and aggregating. Bagging relies on this technique called bootstrapping. To understand bagging, we primarily need to understand how the bootstrapping procedure works. We have already discussed bootstrapping at a high level. Here, I'm going to present bootstrapping with pseudocode. The first thing we need to do is define how many bootstrap iterations we'll use. For random forest, this is equivalent to the number of trees in the ensemble. Bootstrapping is a more generalized technique, but I'm going to focus specifically on how it is used in machine learning and specifically for random forest. The first step is to sample the data with replacement. If you had a training data set with 1,000 examples, for instance, we would create a new training data set with 1,000 examples. The sample with replacement part means that each example can be added to the bootstrap sample more than once provided you end up with the same training set size. In this case, that would be 1,000. After that, we build a decision tree on this data. Then the last step in the loop is to store this result. Once we have all of our models, after going through the loop n times, we can use this ensemble to make predictions. Here's an example of this process in some randomly generated data. The first plot with the blue dots is our training data. Each subsequent plot shows a bootstrap sample taken from the blue dots, of course, sampled with replacement. Also, we plot a regression curve that used a decision tree regressor. I'm using regression here instead of classification because it's a bit easier to visualize the differences between each sample this way. We can see that the shape of each bootstrap sample is similar to the original data. There are small differences, of course, and this is how we induce variation between the individual models. It is hard to tell

exactly what points are different with the visual inspection, but the red regression curve does make it clear that each bootstrap sample contains a slightly different signal. Now, let's take the same example and work on the aggregation step. Remember, at this stage we have iterated through 11 bootstrap samples where in each one, we generated a regression against the bootstrap data. The green lines here represent the output of each bootstrap decision tree regression. The red line is what we get by simply averaging the predictions of the gray lines at each value of x. The green line represents the ground truth. We can see here that averaging the different bootstrap predictions bring us much closer to the actual ground truth, and this is, again, the core goal of bagging. We often expect single individual models to have some level of variance, but that variance means that the predictions can be both under or over the ground truth. If we have k models, then we expect the errors to be randomly distributed around 0. Taking the average cancels out the individual model errors to bring us closer to the actual ground truth. This here is a decision tree regression example, but the same principle applies to classification. This is close to how a random forest algorithm is implemented, but there are a few differences that will be introduced in another video.

Back to Table of Contents

Watch: Choosing Decision Tree Model Candidates

Beyond using randomly sampled data to train a random forest, there is another difference that ensures the individual trees are sufficiently different from one another. In this video, Mr. D'Alessandro explains how using only a different subset of the features for each tree can lead to better performance. In addition, he describes the tradeoff between model performance and scalability when choosing the number of trees to include in a random forest.

Video Transcript

Now, we will cover the full random forest algorithm and then cover some practical tips to consider while working with them. We have already learned about decision trees and also bagging. So there is one more element of the full algorithm to introduce. For ensemble models to be effective, we need the individual models within the ensemble to be independent of each other. With random forest, we do not vary the algorithm of the base models, so we need to find ways to make the individual decision trees more different from each other. I've already introduced one way this base model independence is created, and that is through bagging or bootstrapping the data with each individual tree. The other method is to take samples of the features that go into each tree. When the random forest algorithm loops through each tree, it performs a simple but effective trick on the features. Rather than use all features for each tree, the algorithm randomly selects a subset of them. So while each tree can pull from the same overall set of features, it only learns using a subset of them. The algorithm here is essentially the same as the bootstrapping algorithm I presented in an earlier lecture, but I added the random feature selection for completeness. So to train, we run a loop n times where n is the number of trees we want in the forest. In scikit-learn, the number of trees is controlled by a hyperparameter called n_estimators. Within each loop, we first prepare our training data, which consists of taking a bootstrap sample and then randomly select a subset of the features. After that, we build a standard decision tree and store the result. Last, for the prediction step, we just take an example, run it through each tree, and then get an average over the predictions of the individual trees. As with all algorithms, we have to tune the random forest to get optimal out-of-sample performance. Here, I present the scikit-learn class that is used

to build a random forest classifier. The key hyperparameters to consider can be grouped into forest level parameters and individual tree level parameters. The tree level parameters will be the same for each tree in the forest. There are a lot of options here, but in my experience, one only really needs to focus on the n_estimator's hyperparameter, which is a forest level hyperparameter. For the rest, I tend to rely on the defaults, which are shown here. One reason for this is that we generally want to overfit the individual decision trees. You can see here that the max depth is set to none and the other tree level hyper parameters are set very low. Random forests have a very unique property to consider when tuning the n_estimator's hyperparameter. For most hyperparameters, you'll want to find a value that is neither too small nor too large. This is because the hyperparameters usually control the balance between over- and underfitting. The n_estimator hyper parameter is unique because increasing it too much doesn't usually lead to overfitting. The left-hand plot here shows the training data AUC on a particular data set, and the right-hand plot shows test data AUC. What we really care about is the test AUC. So let's start there. We can see that the AUC never decreases as we increase the number of trees. But at the same time, it doesn't really go up. So the trade-off here is between having good test performance versus the execution time of the random forest. We can see here that having 1,000 trees has generally the same performance as having 100, but it will take 10 times longer to train 1,000 trees. Now let's look at the training data performance. Random forests will severely overfit the training data. In most cases, here the training data AUC is 1, which means perfect predictions. Usually, this is actually a bad sign, but this is actually common for random forests, so it's not something to be concerned about. The test AUC is much lower here, but this is actually as good as it gets for this particular data set. The takeaway again is we tune the n_estimator's hyperparameter to get the right balance between model performance and model scalability.

Back to Table of Contents

Formalizing Random Forest

Bias and Variance in Decision Trees

Recall that the complexity of a supervised learning model is controlled by its hyperparameters. When a model is overly complex, it tends to overfit and leads to a high variance error. Conversely, when a model is overly simple, it tends to underfit and leads to a high bias error.

A decision tree is susceptible to high variance. The optimal balance of low bias and low variance is typically achieved through an ensemble of decision trees, also known as a random forest. The table below shows the advantage of using a random forest versus just a single tree.

☆ Key Points

Random forest is used to achieve the optimal balance of low bias and low variance.

Random forest is an ensemble of decision trees that generalizes better than an individual decision tree.

Although a random forest provides better performance than a single tree, it takes more time and resources to train.

Number of Trees	Tree Depth	Model Estimation Error	Behavior
1	No limit	High variance, low bias	Overfitting
1	Shallow	Low variance, high bias	Underfitting
High	No limit	Low variance, low bias	Good fitting

Random Forest Illustrated

Let's now take a look at how random forest works with the help of an illustration. Figure 1 below shows how a random forest works with three bootstraps. On the left side of the image, we start off with a full data set split into a training set (red) and a test set (blue). We then take three bootstrap samples by randomly sampling the training set into three training subsets. We subsequently train a decision tree on each bootstrap. When we make a prediction on the test set, we would make a prediction from each of the three trees and combine the results. The combined tree will tend to cancel out the error due to variance among one another, leading to good generalization.

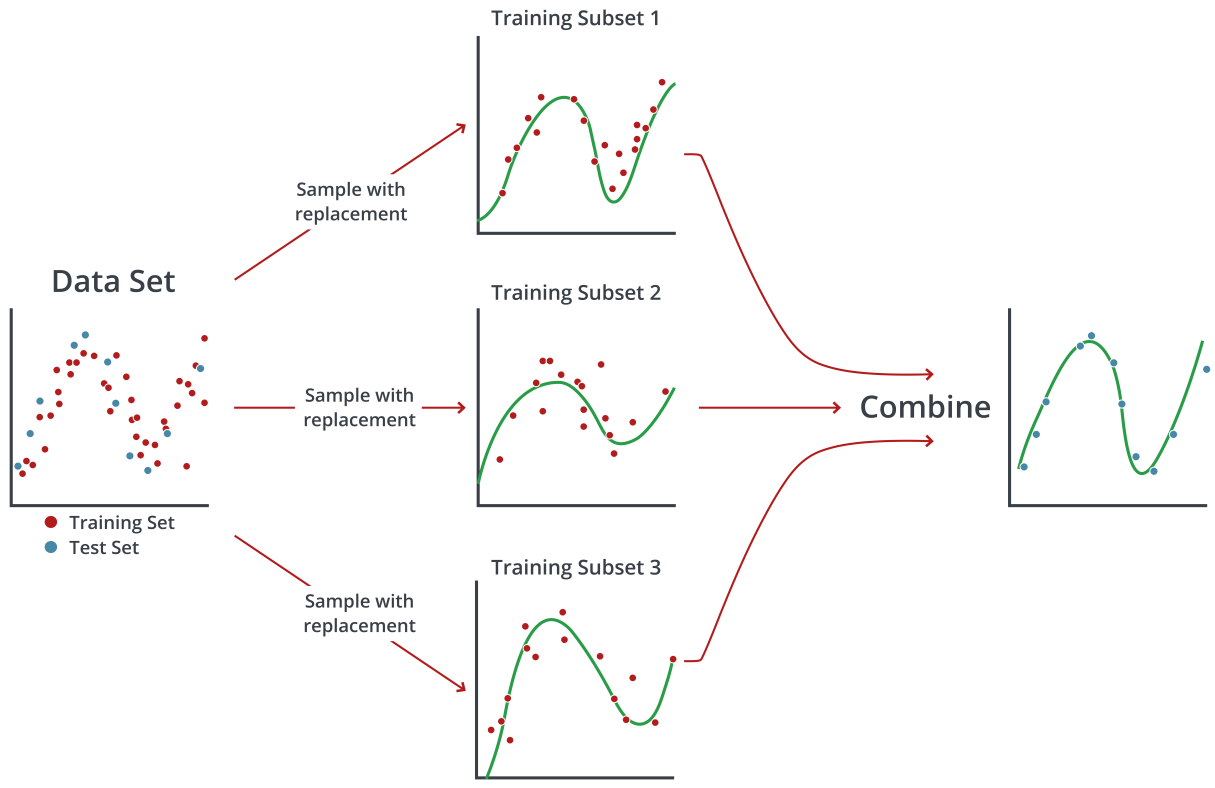


Figure 1. Random forest training illustrated

While this example uses a single feature (x-axis), a real data set would have many features. Another important property of the random forest is that the individual trees will be using only a subset of the features. This allows for sufficient differences

between the trees. Without some diversities in the trees, the resulting random forest will end up with a higher bias since they all have similar configurations.

Random Forest Pseudocode

Training

For $i \rightarrow$ number of estimators:

1. Bootstrap data = sample N examples randomly, with replacement
2. Randomly select a subset of features
3. Build a decision tree with bootstrap data and add it to the ensemble

Prediction

For a given X, get predictions from all the decision trees in the ensemble and average them.

Classification vs. Regression

In the case of classification, the resulting outputs from each tree are generally aggregated using a majority vote, the mode of the ensemble predictions. In regression, the resulting outputs from each tree are generally aggregated using mean.

Tradeoff

While random forest tends to do a better job at generalization than a single tree alone, the training time as well as the prediction time are more costly than a single tree alone. For this reason, we cannot simply have an arbitrarily large number of trees. As a machine learning engineer, part of your job is to find the balance between cost and performance.

Back to Table of Contents

Tool: Random Forest Cheat Sheet

As you have seen, random forest creates an ensemble of decision trees that were trained independently on data bootstraps using a subset of random features for each split. Use this cheat sheet to review the details of random forest when implementing the algorithm yourself.

Download the Tool

Use the **Random Forest Cheat Sheet** to review how the algorithm functions.

Back to Table of Contents

Assignment: Building Random Forests

In this exercise, you will train two random forest classifiers and compare their performances.

This exercise will be graded.

When you finish your work:

1. Save your notebook by selecting the **Save and Checkpoint** entry from the **File** menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education —> Mark as Completed** in the upper left of the Activity window: 

3. This assignment will be auto-graded and can be resubmitted. After submission, the Jupyter Notebook will remain accessible in the first tabbed window of the exercise. To reattempt the work, you will first need to click **Education —> Mark as Uncompleted,** then proceed to make edits to the notebook. Once you are ready to resubmit, follow steps one and two.

This exercise will be auto-graded.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.

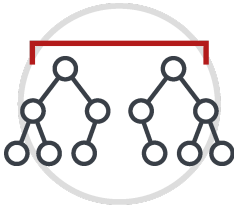
Back to Table of Contents

Module Wrap-up: Understand Random Forest

In this module, Mr. D'Alessandro discussed a powerful ensemble algorithm known as random forest and described the design of it along with other practical considerations for its usage. Remember that random forest can be used for both the regression and the classification problems. Mr. D'Alessandro also explained how to make the trees sufficiently different by using random sampling with replacement and using only a subset of all features. You discovered the tradeoff between performance and scalability then had the chance to put your knowledge to use by building a random forest in a practical exercise.

Back to Table of Contents

Module Introduction: Understand Gradient Boosting



The third ensemble learning algorithm is the gradient boosted decision tree, also known as GBDT. In this module, Mr. D'Alessandro discusses the differences and similarities between a GBDT and a random forest. He will also demonstrate how a GBDT is trained and optimized. After studying the theoretical foundation of GBDT, you will solidify your understanding by building, tuning, and comparing different models through a practical exercise.

Back to Table of Contents

Watch: Gradient Boosted Decision Trees

The third type of ensemble algorithm is the gradient boosted decision tree (GBDT). Here, Mr. D'Alessandro describes the steps for training a GBDT and explains the pros and cons of using an ensemble model. As you watch this video, think about how this algorithm compares to a random forest.

Video Transcript

Let's now begin our discussion on the third ensemble modeling technique called boosting, and an algorithm associated with boosting called gradient boosted decision trees. The name here is somewhat of a mouthful but we can start with the familiar. Gradient Boosted Decision Trees, or GBDT for short, is an ensemble method consisting of individual decision trees. It is different from random forests in almost every way except that both are ensembles of decision trees. The math behind this algorithm is pretty sophisticated but I think the intuition is very straightforward. I'll cover the basic structure of the model and give a qualitative overview of how it is trained. Here is the structure of the GBDT model. As we can see here, this resembles our typical ensemble structure. The base models are individual decision trees, and the full model is a weighted sum over the trees. In random forest, we would take a simple average over individual decision trees. With GBDT, each tree has a weight that the learning algorithm determines. I want to draw more comparison to random forest here with a deeper look at the base decision trees. In random forest, each decision tree uses a subset of features, a bootstrap sample of the data, and is typically overfit. With GBDT, all of this is different. We leverage all features and generally try to create shallow trees. The last point is something you will control with the max depth tree parameter. Probably the biggest difference in the idea that actually defines the gradient boosting part is that each tree is trained on the cumulative prediction error instead of the original label. This last part is where this gets mathematically sophisticated, but it can be illustrated nicely with some plots. Let's start with a regression problem. The plot here shows a relationship that a simple linear model would have a hard time fitting. To build a GBDT, we need to first start with a simple model. This simple model is reflected by the red line here, which is actually just a uniform guess of the average value of the target variable or label. Like I said, simple is okay here. The next stage of the learning

algorithm is a series of steps that we repeat in an iterative way. After starting with a simple first model, the next stage is to compute the residual or error. In regression, this is simply the difference between the true value of Y and the prediction up until this point. The left-hand chart here is the residual between the data and the first model, which was just the average value of the target variable. The next step then would be to fit a decision tree to this residual. This is illustrated in the right-hand chart. You can see that we've identified the general shape of this residual curve, but it's still not a perfect fit. After finding a weight for this new tree, we'll add this tree to the ensemble. Then we repeat this entire thing. In other words, we compute the residual again, this time computing it with the latest tree added. This process repeats itself until we hit our stopping criteria, which is usually just the number of trees we specify when building the algorithm. I usually find this process easier to illustrate and define with regression problems because the error we use in the training is just the simple subtraction of the prediction and the label. We'll mostly use GDBT in classification problems, though. I'll try to illustrate that as well. Here are some simulated data where the positive class in red is part of an inner circle surrounded by negative class points. The right-hand side shows the decision boundaries for the first nine trees built on this data. Each tree is a simple model that individually doesn't look very interesting. But when we start adding them together, we can see the power of this method. This plot shows the decision boundaries for different iterations of this process. The top left is iteration 1 and they go up in orders of magnitude up to about 700. We can see as the number of trees grow, we're starting to capture the circular shape we should expect. As we get higher, the broader circular shape is there, but we can see a lot of color variation within the inner circle. Remember that the colors here represent the predictions the model is making. When we see small patterns of blue embedded in the red, this is a sign of overfitting. The middle band here is where our sweet spot is. It has enough trees to capture the general shape of the original data but not so many that it is fitting the noise within that shape. In random forests, increasing the number of trees usually always improves performance but again at the cost of scalability. This is not true for gradient boosting decision trees. Increasing the number of trees too much can result in overfitting as well as reduce scalability. This is a hyperparameter we have to be careful to tune. Last, now that we've covered gradient boosting and random forests, it is important to get a sense of when to use these. In general, there are no specific domains or applications where it is always better to use either

algorithm or better to use ensembles in general over non-ensemble techniques. Choosing the ensemble methods is typically an empirical question and it is common to test both random forest and GBDT against methods like logistic regression or decision trees. But I can present some general rules of thumb for cases in which the ensemble techniques might be better. For one, these techniques require more data. If you have a reasonably large data set, you may find ensembles are going to be better. But they also tend to be slower, both to train and to make predictions. If you're working on a real time application that requires very fast predictions, something like a logistic regression might be better. Last, while well-fit ensembles typically generalize better, we also lose interpretability. Interpretability here just means our ability to understand why a particular prediction was made given an example's features. If you work in an environment with strict model interpretability rules, which is often true in health care or financial services, ensemble methods might not be appropriate. Overall, we should start by thinking about the design constraints on the problem, such as scalability and interpretability. If there is no reason to rule out ensemble methods, it then becomes a matter of predictive performance, which we can establish using common model selection techniques.

Back to Table of Contents

Read: Formalizing Gradient Boosted Decision Trees

Overview

At a high level, a gradient boosted decision tree works very similarly to gradient descent in logistic regression. We typically start off by initializing a very simple (shallow) decision tree and compute the error (also known as residual) of this tree with respect to the training data set. We then train another tree against this residual so that this new tree represents the amount of error in this tree. Next, we use this new residual tree as "gradient" and move our original tree toward the direction of this residual tree by a small step. Finally, we repeat this process over and over again until we have reached the number of trees that we desire. You can more easily visualize this process by examining Figure 1.

☆ Key Points

GBDT is another ensemble learner that is capable of achieving low bias and low variance.

GBDT trains shallow trees against the error of the model and uses the resulting trees as gradient to improve the ensemble model.

There are many similarities and differences between GBDT and random forest; in practice, you will need to experiment with both.

GBDT Training Illustrated

In Figure 1 we start with a very simple model $f(X) = T_0(X)$, one that always produces a prediction of 0 regardless of the value of X. We subsequently calculate how much each point is off from the data, X, to get our residuals, as represented by the blue dots. We then train a shallow decision tree $T_1(X)$ against the blue dots to get our "gradient" tree. Next, we produce a new $f(X)$ by adding $\gamma \cdot v_1 \cdot T_1(X)$ to our previous version of $f(X)$, where γ is a learning rate and v_1 is an optimized weight. The above steps repeat until we reach the number of trees desired.

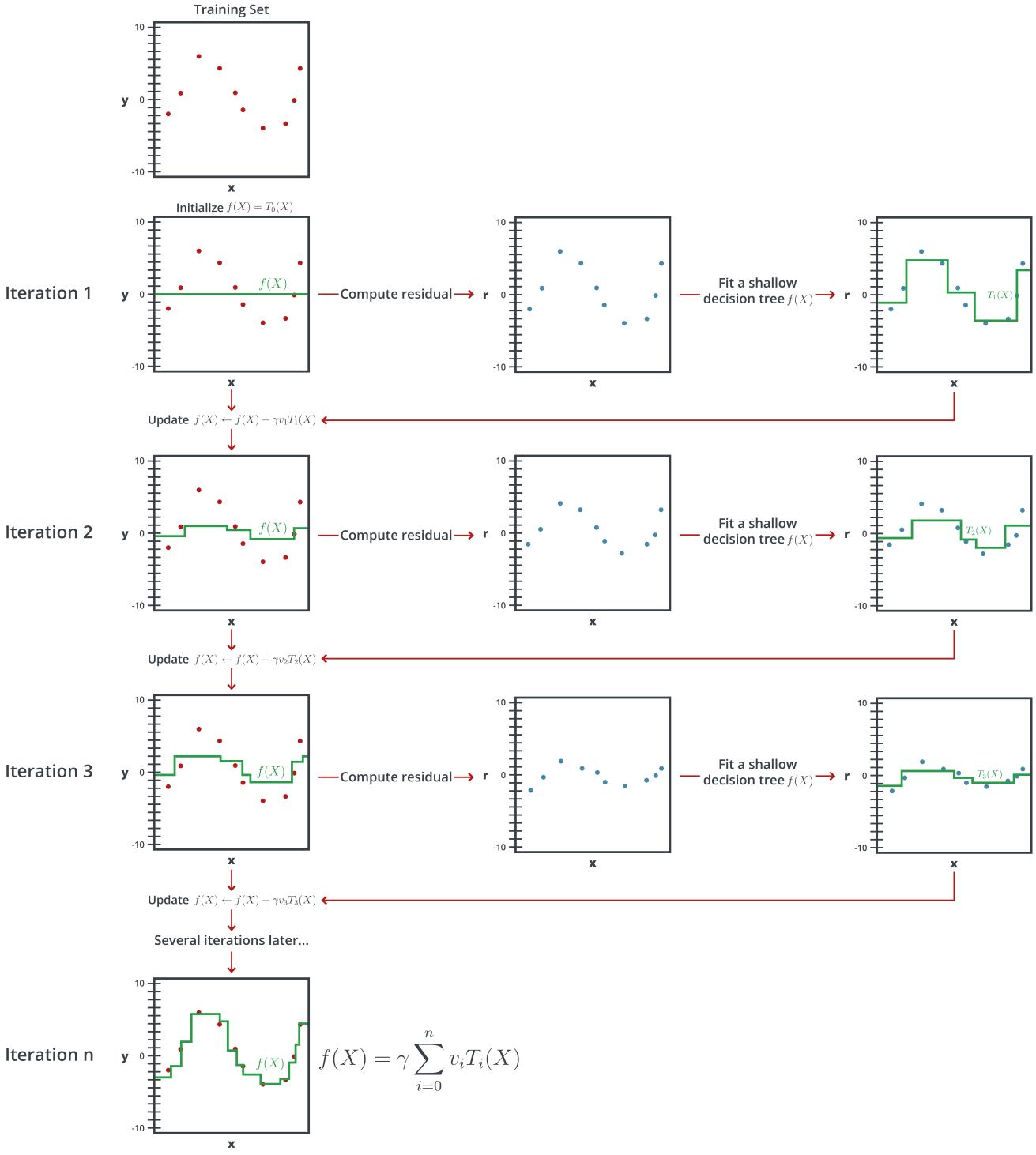


Figure 1. GBDT training illustrated for regression

As you can see, the residual graph gradually flattens out as the error of $f(X)$ is reduced. Also, observe that our ensemble model $f(X)$ becomes more and more fitted against the training set as the number of iterations increases.

GBDT Pseudocode

Initialize $f(X) = T_0(X) = 0$, the average of y.

For i from 1 to N (the total number of boosted trees desired):

1. Compute the residual.
2. Train a shallow tree T_i(X) against the residuals.
3. Identify the optimal weight v_i.
4. Update $f(X) \leftarrow f(X) + \gamma \cdot v_i \cdot T_i(X)$.

Random Forest vs. Gradient Boosted Decision Trees

Random forest and GBDT are similar in that they are both ensemble methods consisting of multiple decision trees. Both are excellent at reducing errors due to bias and variance. In practice, you will likely want to experiment with both models before choosing one over the other. Table 1 highlights their main differences in implementation.

Random Forest	GBDT
Consists of deep trees	Consists of shallow trees
Trains on randomly sampled data	Trains on residuals
Uses a subset of features	Uses all features

Table 1. Comparison between random forest and GBDT

Back to Table of Contents

Watch: GBDT Optimization

As with all machine learning algorithms, GBDT has various hyperparameters that influence its performance. In this video, Mr. D'Alessandro introduces the three most important hyperparameters for GBDT — n-estimators, learning rate, and max depth — and describes how to perform a comprehensive search of the ideal hyperparameters using loops and scikit-learn. To further reinforce your understanding, Mr. D'Alessandro explains with an example of overfitting when the n-estimator grows too large.

Video Transcript

The gradient boosted decision tree method is a powerful choice of algorithm, but it is also susceptible to overfitting. The typical way to mitigate overfitting risk is to run a thorough, hyperparameter search to find that middle ground between over- and underfitting. GBDT is no different, but I'll have to admit that this process is a little more involved than what we've seen in our other supervised learning algorithms. There are many more hyperparameters to tune, and since this is an ensemble method with many decision trees, hyperparameter optimization can take much longer. As usual, we'll use scikit-learn to build a gradient boosted decision tree. There are many hyperparameters to consider, but I'm going to highlight what I consider to be the most consequential. These are n_estimators, learning rate, and max depth. The first two in this table are ensemble level hyperparameters, and the last is for the individual trees. For the tree level hyper parameters, each individual tree will have the same parameters applied to them. In each of these, the higher the number, the more complexity your model will have. In other words, for the higher ranges, your model is more likely to overfit. The meaning of n_estimators and max depth have already been covered. I'll focus more on the learning rate. This is a parameter that reduces or shrinks the weight of each decision tree in the ensemble. For each of these parameters, I have presented some recommended ranges to test. These are just a starting point, so I won't guarantee they're optimal for every single problem, though. You can see that for the learning rate, we have pretty small values, like 5 or 10%. This means that after we build a tree on each iteration, we're only adding about 5 or 10% of that tree's prediction value to the total sum. There are natural tradeoffs between these hyperparameters as well. To get the same fit, we would need more estimators when the learning rate is

lower. Similarly, if max depth is higher, then we may need fewer estimators overall. A common practice is to loop through all combinations of these hyperparameters and perform standard out of sample validation. Here is an example of this process using scikit-learn. At this point, we should hopefully see some common patterns. The structure of this code is a set of nested loops. This is an explicit way of doing this, although scikit-learn does provide more convenient ways to achieve the same thing. The main thing that is GBDT specific in this code is the actual specification of the model. Here, we're using the gradient boosting classifier class and initializing it with the method specific hyperparameters we've discussed. After that, we'll call the standard fit, prediction, and evaluation methods. Now, remember, this procedure can take time. This specific loop structure has 18 iterations, and in each iteration, we're training between 100-300 decision trees. This naturally will be time consuming. For a reasonably sized data set, you can expect this to take an hour or more to finish. Although it takes time, I want to emphasize how important this hyper parameter search is. Here are example evaluation curves on both training and test data sets. The x axis is the number of trees in the ensemble. As this number increases, our training loss gets lower and lower, which is represented by the blue curve. The red curve is our out-of-sample, or test set loss. It quickly decreases in the first 50 trees, but slowly increases after that. This is a classic example of overfitting. I'll also illustrate here the zones of under- and overfitting. It is pretty easy to avoid underfitting with this method, but more challenging to stay out of the overfitting zone. While it's a very powerful algorithm, always take the time to tune gradient boosted decision trees appropriately.

Back to Table of Contents

Assignment: Building Gradient Boosted Decision Trees

In this exercise, you will train two gradient boosted decision tree classifiers and compare their performances.

This exercise will be graded.

When you finish your work:

1. Save your notebook by selecting the **Save and Checkpoint** entry from the **File** menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education —> Mark as Completed** in the upper left of the Activity window: 

3. This assignment will be auto-graded and can be resubmitted. After submission, the Jupyter Notebook will remain accessible in the first tabbed window of the exercise. To reattempt the work, you will first need to click **Education —> Mark as Uncompleted,** then proceed to make edits to the notebook. Once you are ready to resubmit, follow steps one and two.

This exercise will be auto-graded.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.

Back to Table of Contents

Tool: GBDT Cheat Sheet

As you have discovered, a set of weak models with high bias can be combined to form an ensemble that has low bias. Use this cheat sheet to review the details of gradient boosted decision trees when implementing the algorithm yourself.

Download the Tool

Use the **GBDT Cheat Sheet** to review how the algorithm functions.

Back to Table of Contents

Ask the Expert: Mehrnoosh Sameki on Addressing Overfitting and Underfitting

You want to ensure your model is performing properly with well-suited data. Sometimes you may experience good model performance on the training data but poor generalization; this means your model is overfitting. Other times you may have both poor model performance and poor generalization; this means your model is underfitting. In both cases, you need to refine your model. In these videos, Dr. Sameki explains how you can improve your model when it is overfitting or underfitting.

Note: The job title listed below was held by our expert at the time of this interview.

Mehrnoosh Sameki
Product Manager, Microsoft

Dr. Mehrnoosh Sameki is a Senior Technical Program Manager at Microsoft, responsible for leading the product efforts on machine learning interpretability and fairness within the Open Source and Azure Machine Learning platforms. Dr. Sameki also co-founded Fairlearn and Responsible-AI-widgets and has been a contributor to the InterpretML offering.

Question 1

How did you improve your model when you experienced overfitting?

Video Transcript

So, in this particular case, there are more solutions that have been proven to improve. One of them is cross-validation. It's a very effective preventative approach against overfitting. And so it's when you are trying to create this like train/test splits and then train on the specific data and then try to test on some other specific data. We also have things like cross-validation or k-fold cross-validation, where we divide the data into k different subgroups and then we try to repeatedly train on k-1 folds with the remaining one serving essentially as your test data set. So that's one of the most effective ones that I've seen. Obviously, always a great idea to bring more training data. More training data means that you are having better representation of

subgroups. So, say, different ethnicities, different age groups, or different -- any characteristic you're going to have better representation of them. And so the model tends to understand the patterns better. Data enhancement and data augmentation could be also another one. Necessarily you might not need more features; sometimes you could use data augmentation or feature engineering to come up with more complex features that are more descriptive of the characteristics of your data. And the other thing is regularization. It essentially refers to various strategies for pushing the model to be a little bit simpler. And so the approach you choose will be determined by the learner that you're using in your machine learning. So for example, if you're using a decision tree, you could prune a decision tree and perform -- If you're using a neural network, you could perform dropout on a neural network. Or if you're using, say, something else, you can add a penalty parameter to a regression cost function so that it's not like super complex. Ensembling is another good method when you're bringing together predictions from numerous different models and you're combining them in order to provide one prediction. There is also -- the two most prevalent techniques of ensembling are boosting and bagging, which -- one works by increasing the collective complexity of basic base models and so it educates many weak learners in a series, and each learner learns from the mistakes of a learner before them and so keeps improving. And then bagging works by training a large number of strong learners in a parallel pattern and then merging them to improve their predictions. If you're working with a neural network, sometimes early termination is a good one. When training a learning algorithm iteratively, you might assess how well each model iteration performs. And so in order to avoid overfitting and being able to generalize well, sometimes it's great to refine the model until a specific number of iterations is reached.

Question 2

How did you improve your model when you experienced underfitting?

Video Transcript

So usually underfitting is happening when you have a high bias in your answers and low variance when your size of training data set is not enough. So that's where the model doesn't know exactly what are the patterns or characteristics or relationship between different features or features and outputs when the model is too simple.

While you have a lot of complexity in the pattern on your data, you've chosen a model with a simple architecture and that architecture has limitations of not adapting well to the complexity of your data, and then when the data is not cleaned and has so much noise in it. And so one of the most effective ones is sometimes when you increase model complexity and increase the number of features by performing the right feature engineering, then you start seeing a lot of improvements in terms of underfitting. And then sometimes you have to be careful because adding a lot more parameters to the model obviously adds to its complexity, and so it needs to be a little bit of a balance of to what extent you're going to go with more complex models in order to see an improvement. But another good thing that works there is make sure that you're cleaning your data well. So there is noise that is removed from your data set that you're then throwing at the model for training. And then if you are working with maybe more like deep neural network models, you can sometimes increase the number of epochs or increase the duration of training in order to get better results with your model.

Back to Table of Contents

Assignment: Unit 6 Assignment - Comparing Regression Models

In this assignment, you will work toward solving a regression problem. You will train and evaluate multiple regression models to predict a continuous label.

This assignment will be graded.

When you finish your work:

1. Save your notebook by selecting the **Save and Checkpoint** entry from the **File** menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education —> Mark as Completed** in the upper left of the Activity window: 

3. Note: This assignment will be manually graded by your facilitator. You cannot resubmit.

This assignment will be graded by your facilitator.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.

Back to Table of Contents

Assignment: Unit 6 Assignment — Written Submission

In this part of the assignment, you will answer five questions about using ensemble methods.

The questions will prepare you for future interviews as they relate to concepts discussed throughout the week. You've practiced these concepts in the coding activities, exercises, and coding portion of the assignment.

Completion of this assignment is a course requirement.

Instructions:

1. Download the **Unit 6 Assignment document.**
2. Answer the questions.
3. Save your work as one of these file types: .doc or .docx. No other file types will be accepted for submission.
4. Submit your completed Unit 6 Assignment document for review and credit.
5. Click the **Start Assignment** button on this page, attach your completed Unit 6 Assignment document, then click **Submit Assignment** to send it to your facilitator for evaluation and credit.

Back to Table of Contents

Module Wrap-up: Understand Gradient Boosting

In this module, Mr. D'Alessandro discussed the third ensemble learning method — the gradient boosted decision tree, or GBDT for short. You discovered how GBDTs are built iteratively with decision trees that are trained on the model's residuals. You also explored the three common GBDT hyperparameters: n-estimators, learning rate, and max-depth. Finally, you were able to further solidify your knowledge by building, tuning, and comparing different models through a practical exercise.

Back to Table of Contents

Lab 6 Overview

In this lab, you will implement the stacking ensemble method to solve a regression problem. You will also implement individual regressors and compare the resulting performances of the individual models and the ensemble model. You will be working in a Jupyter Notebook.

This three-hour lab session will include:

- **10 minutes** - Icebreaker
- **30 minutes** - Week 6 Overview and Q&A
- **20 minutes** - Breakout Groups: Big-Picture Questions
- **10 minutes -** Class Discussion
- **10 minutes - Break**
- **30 minutes** - Breakout Groups: Lab Assignment Working Session 1
- **15 minutes** - Working Session 1 Debrief
- **30 minutes** - Breakout Groups: Lab Assignment Working Session 2
- **15 minutes** - Working Session 2 Debrief
- **10 minutes -** Concluding Remarks and Survey

By the end of Lab 6, you will:

- Use the stacking ensemble method to train four regression models.
- Train and evaluate the same four regressors individually.
- Visualize and compare the performance of the stacked ensemble model to that of the individual models.

Back to Table of Contents

Assignment: Lab 6 Assignment

In this lab, you will continue working with the Airbnb NYC "listings" data set.

This assignment will be graded.

When you finish your work:

1. Save your notebook by selecting the **Save and Checkpoint** entry from the **File** menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education —> Mark as Completed** in the upper left of the Activity window: 

3. Note: This assignment will be manually graded by your facilitator. You cannot resubmit.

This lab assignment will be graded by your facilitator.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.