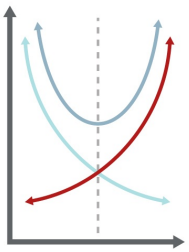
Debugging and Improving Machine Learning Models

What you'll do

- **Identify the cause of high prediction error by recognizing high bias or high variance**
- **Mitigate the negative impact of a bad bias-variance tradeoff on your model**
- **Analyze how ensemble methods reduce bias and variance in order to improve the predictive model**
- **Implement bagging and boosting to improve the predictive model**



Course Description

Making predictions from data is not as easy as running pre-written code out of the box. A data scientist has to make many decisions, such as which algorithm is best suited, which pre-processing steps to take, or what hyperparameters to choose. Many choices create more opportunities for error; it is therefore important to know how to debug suboptimal settings.

This course will provide you with different strategies to improve and fine-tune machine learning models in a principled fashion. Professor Weinberger will introduce methods such as bias-variance decomposition, bagging, and boosting. The bias-variance decomposition helps analyze the error made by learning algorithms with respect to three different parts: bias, variance, and noise. You can determine which part dominates and apply appropriate tools to improve your ML setup. You will analyze each technique and practice implementing them to make your classifier stronger and more accurate.

Kilian Weinberger
Associate Professor
Computing and Information Science, Cornell University

Kilian Weinberger is an Associate Professor in the Department of Computer Science at Cornell University. He received his Ph.D. from the University of Pennsylvania in Machine Learning under the supervision of Lawrence Saul and his undergraduate

degree in Mathematics and Computer Science from the University of Oxford.

During his career, Professor Weinberger has won several best paper awards at ICML (2004), CVPR (2004, 2017), AISTATS (2005), and KDD (2014, runner-up award). In 2011, he was awarded the Outstanding AAAI Senior Program Chair Award and in 2012 he received an NSF CAREER award. He was elected co-Program Chair for ICML 2016 and for AAAI 2018. In 2016 he was the recipient of the Daniel M. Lazar '29 Excellence in Teaching Award.

Professor Weinberger's research focuses on machine learning and its applications. In particular, he focuses on learning under resource constraints, metric learning, machine-learned web-search ranking, computer vision, and deep learning. Before joining Cornell University, he was an Associate Professor at Washington University in St. Louis and previously worked as a research scientist at Yahoo! Research.

Table of Contents

Read: Formalize Bootstrapping
Watch: Reduce Variance With Bagging
Read: Formalize Bagging
Watch: Benefits of Bagging
Watch: Random Forest
Read: Random Forest
Tool: Random Forest Cheat Sheet
Random Forest
Module Wrap-up: Reduce Variance With Bagging

Module 3: Reduce Bias With Boosting

Module Introduction: Reduce Bias With Boosting
Watch: Gradient Boosted Regression Trees
Read: Gradient Boosted Regression Trees
Watch: Gradient Descent in Functional Space
Read: Gradient Descent in Functional Space
Tool: GBRT Cheat Sheet
Gradient Boosted Regression Tree
Module Wrap-up: Reduce Bias With Boosting
Read: Thank You and Farewell

Course Resources

Glossary

Q&A

Live Labs: Building a Community of Learners
Live Labs Q&A

Back to Table of Contents

Live Labs: Building a Community of Learners

Building a Community of Learners

This course will include two live (synchronous) video sessions called **Live Labs**. Live Labs will be offered each week as an opportunity to connect with your peers and instructor to address questions about the coursework and dive deeper into the material. Each hour-long session will be scheduled in advance. You are highly encouraged to take advantage of this chance to connect with your community of learners to build relationships and make this course experience even more rewarding.

This short Q&A section of the course contains a discussion called **Live Labs Q&A** where you can post questions and interact with your peers at any time as you work through this course. Common questions and important topics brought up in this discussion may be addressed in the Live Labs sessions.

In order to review the session schedule and join a session, you'll need to navigate to the **Live Labs** link under the **Course Shortcuts** section in the navigation menu on the left.

Back to Table of Contents

Live Labs Q&A

The Live Labs Q&A discussion is a place for you to post questions and interact with your peers throughout your work in this course. Common questions and important topics brought up in this discussion will likely be addressed in a Live Labs session.

Instructions:

Read through existing posts to find out what other students are discussing. Upvote posts that you find interesting or contain questions that you also have by "liking" it with the thumbs-up button at the bottom of the post.
If you have something to contribute to an existing discussion, reply to the thread and get involved in the conversation.
If you have new ideas, questions, or issues, post them in this Live Labs discussion board. If you have potential answers or suggestions that may resolve a question thread, feel free to share your understanding in a reply. This will help guide the instructor's feedback and discussion during their live session.

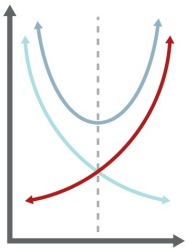
The questions that are the most upvoted or replied to that haven't been resolved will likely be addressed at the scheduled time by the instructor.

Back to Table of Contents

Module 1: Decompose the Generalization Error

Back to Table of Contents

Module Introduction: Decompose the Generalization Error



You have encountered linear models such as perceptron and nonlinear models such as classification trees. After training these models on a training data set and applying the model to a test data set, you obtain the test error. If the test error is reasonably small, then you are all set. However, what do you do if the test error is high?

There are two main reasons for high test error: Either your model is too simple (i.e., it cannot even learn the training data sufficiently) or your model is too complex (i.e., it is memorizing the training data). In either case, the model won't generalize to testing data. It is hard to tell whether the model is too simple or too complex by just looking at the error. You therefore need a principled way to differentiate the two settings, and that is what we are going to cover in this module: the bias-variance decomposition.

You will start with a review of validation error, followed by decomposing the error into three parts: bias, variance, and noise. Professor Weinberger will explain each of these parts, their negative effects on prediction accuracy, and how you can mitigate issues caused by these sources of error. You will practice improving your classifier in various ways by reducing the negative impact of high bias and high variance.

Back to Table of Contents

Watch: Define Bias-Variance Decomposition

You can quantify the inaccuracy of your algorithm through validation error. Validation error decomposes into three parts: bias, variance, and noise. In this video, Professor Weinberger will define these terms and explain how they affect machine learning algorithms.

Video Transcript

In machine learning, we train a classifier on a training data set that ultimately should do well on test data. We've already talked about how we approximate the test data with validation data and we look at the validation error. But what if your validation error is too high? So you train your classifier, you evaluate it, even with the best hyperparameter setting you're just not doing well, right? Your validation error is too high; you know if you deploy this model, it's not going to be good. That's a problem. As a data scientist, you should be able to diagnose this problem and identify what the cause of the problem is. In fact, it turns out that high validation error, or high test error — same thing, essentially — decomposes into three possible causes. So the error actually is a sum of three terms: the variance, the bias, and the noise. These three terms, and only these three terms, contribute to your error. So it's very important that you understand what these three terms mean and how to diagnose which one of them is too high, because there's different ways to lower these terms. Let's first unpack variance and bias. Let me picture machine learning as a game of darts, where when you train a machine learning algorithm, you throw a dart, and when you evaluate it you see how close you got to the bullseye. In the ideal setting — that's the low-bias/low-variance setting — you throw your darts and every single dart just hits the bullseye. Right? That's great. That's basically — you have no bias in any direction, you have no variance; you have no problem. You want to get into that setting, but often you are not. One problem is it could be, if your error is too high, that you have high variance. What does high variance mean? High variance means that depending on what training data set you train on, you get a very different classifier. So, in the dart sample, that would be every time you throw a dart — that means you take a training data set, you train a classifier, and you evaluate it — it ends up somewhere else. Right? So depending on what training data set you trained on, you can get a very different classifier. On average, you're doing well, but unfortunately, because you trained on a very specific data set, you overfitted to that data set, so you got dragged off in one direction or the other direction. That's a setting with high variance and low bias. So you have data points that are kind of around the bullseye, but they're spread out, and none of them is really, really good. Another scenario is the setting with low variance and high bias. Low variance means all your dots are very close together. So in machine learning that

means no matter what training data set you train your classifier on, they all turn out very similar. But the high-bias setting means, unfortunately, they're all off, right? All your darts are close together, but they're just not hitting the bullseye. You hit always to the right or always too high or something. When can that happen? That happens when you make the wrong assumptions, right? Let's say you train a linear classifier on a data set that's just not linearly separable. Every time you train it, right, no matter what the data set is, you will always be off, because the assumption that the data is linearly separable is just not right. You can never hit the bullseye. It's impossible with a linear model. You're always off, consistently. That's a high-bias scenario. The worst case, of course, is if you have high bias and high variance. That's when you're off, and depending on what data set you train on, your final classifier varies a lot depending on the data set. There's a third term, and that is high noise. High noise, in the dart example you could imagine that there's an earthquake and the dartboard just shakes around. You throw the dart, but there's nothing you can do if the bullseye moves around. In machine learning, essentially that means you can't trust your labels or you can't trust your features. They may just be really noisy. Imagine the label, for example, is wrong, or the feature is wrong.

Back to Table of Contents

Watch: Formalize Bias, Variance, and Noise

Now that you are familiar with decomposing error into bias, variance, and noise, Professor Weinberger formalizes each term in greater detail and introduces the corresponding mathematical definitions.

Video Transcript

All right; let's formalize these three terms: bias, variance, and noise. For this we need a few terms, and the first thing we start off with is hD. hD is a classifier that is trained on data set D. So D is our training data set, and we assume it's a random variable. So, we basically — there's some distribution that we don't know, right? That's the distribution of the true data. And from this distribution we draw some training data. Imagine, for example, you're trying to predict house prices from some feature vectors that describe a house; for example, how many bathrooms the house has, how many rooms it has, what's the square footage, etc., what's the school district, the crime rate, etc., in a certain area. So, there's some distribution out there over houses and their prices, and we take a bunch of them and just look up their statistics and the house prices. This is our training data set D. Then we take this data set and learn a classifier hD. We can now make a prediction for this test point x. And so the ultimate error that we're trying to minimize is the expected error for a new test point x with label y. So we call this the error of the algorithm A. So essentially we say if we train a classifier D on some training data set, and then we draw a new data point x with a label y, how far off would we be; what would be the expected test error of this algorithm? That's all you want to minimize, ultimately, if you design a new algorithm. This is the term that we will now unpack. To do this, we first define the average label, or the expected label. So for a certain test point x, the label may actually vary, right? So for example, you may have multiple identical houses, and they won't sell exactly for the same price. So that means we have some probability distribution P of y given x. That means for a specific house, there is some distribution over the label y. This could be either down to noise or just random fluctuations, but it could also be because of things that are not captured in our x. So maybe there are some additional things about this house that we don't know. Let's say a famous celebrity lives in that house. That would make it more expensive, but it's just not captured in our feature representation; we have no dimension that says, "Did a famous celebrity live in that house?" So that would make it more expensive than we thought it would be, or less expensive if the person is notorious.

Essentially what we are trying to compute, what we are trying to aim for, is the expected label, saying, "Well, over all the houses that are exactly like this, what is the average price that this house would go for?" That's basically all we can do. The stuff

that's not captured in the feature we can't possibly predict. We call this y-bar. y-bar is the expected label of a certain data point x. Then we also have h-bar. h-bar is the expected classifier that we would get if we had many, many trained data sets. So imagine basically you had infinite amount of data, although you could take many, many data sets. And we now compute the expected classifier you would get from all of these data sets. Essentially it's the average classifier; that we call this h-bar. And with these different terms, we can now actually formalize variance, bias, and noise. The first thing is noise. So noise is the expected difference between the label of a data point and its expected label. So if, for example, a data point — we have houses that always sell with exactly the same price, and this would have very little noise; there is very little uncertainty. That's where the dartboard is totally in place and not moving at all. If some kind of houses vary a lot depending on some factors that we don't know, right, then we have a lot of noise. That means for the same x, the price that is actually — the label y that actually is associated with that x, varies widely across different data points. This is the noise term. The noise term usually we can't address with a machine learning algorithm. We can address by changing the data, by cleaning up the data, for example, because the label may be wrong or the features may be wrong, or by adding additional features that we didn't incorporate before. The second term is the variance. The variance says, how much do our classifiers vary if we train them on different data sets? So we basically say if we train our data set classifier hD on data set D, how different is that prediction from the average prediction or the expected prediction you would get if we had infinite amount of data? So what we are measuring here is, how much does the classifier essentially overfit to a specific data set? If this varies a lot, then we have high variance. If this varies very little, then basically it doesn't really matter what training data set you train it on; the classifier is pretty similar every single time. The last term is the bias. The bias is the prediction of the average classifier minus the average label squared. So that essentially tells us even if you had unlimited amount of data and you would average all these classifiers — so you get a really good classifier, right — this is the error that even that classifier would make. So this captures, how off are your assumptions? If you, for example, assume that the problem is linear and use a linear regression algorithm, it doesn't matter how much data you have, right? You will always be wrong if your data is not actually linear, right? You cannot possibly capture this. So the bias is the error that you will always have even if you have unlimited amount of data. The variance is the error that you get because you trained on one specific data set and not infinite amounts. And the noise is the error that you get because your data is not very clean or some labels are wrong or there's some attributes, some factors in the data that you just don't represent in your feature vectors and you couldn't possibly get right.

Back to Table of Contents

Read: Formalize Bias, Variance, and Noise

A classifier's error can be decomposed into three parts: bias, variance, and noise.

Noise refers to intrinsic variation in the data that even the best classifier cannot capture (because they are not captured by the features, or due to erroneous labels).

Variance offers a measure of overfitting, while bias tells us whether our model is structurally misaligned with the underlying mechanism producing our data.

Given a data set , drawn independently and identically distributed from some distribution , and assuming a regression setting (i.e.,), let's decompose the test error of a classifier into three rather interpretable terms. We first define several terms then formally state the bias-variance decomposition.

Expected Label (given)

First, let us consider that for any given input , there might not exist a unique label y. For example, if your vector describes features of a house (e.g., number of bedrooms, square footage) and the label its price, you could imagine two houses with identical descriptions selling for different prices. So, for any given feature vector , there is a distribution over possible labels. We therefore define the following, which will be useful later on:

The expected label denotes the label you would expect to obtain, given a feature vector .

Expected Test Error (given)

Now imagine that we have a machine learning algorithm trained on this data set to learn a hypothesis (aka classifier). Formally, we denote this process as . For a given , learned on data set with algorithm, we can compute the generalization error (as measured in squared loss) as follows:

We use squared loss here because it has nice mathematical properties and because it is the most common loss function.

Expected Classifier (given)

Now, remember that itself is drawn from and is therefore a random variable. Further, is a function of and is therefore also a random variable, which means that we can compute its expectation:

where is the probability of drawing from . Here, is a weighted average over functions.

Expected Test Error (given)

We can also use the fact that is a random variable to compute the expected test error only given , taking the expectation also over .

To be clear, represents our training points and the pairs are the test points. We are interested in exactly this expression, because it evaluates the quality of a machine learning algorithm with respect to a data distribution .

Bias-Variance Decomposition

With a few standard operations, we can decompose the expected test error as defined above into three meaningful terms:

Variance: Captures how much your classifier changes if you train on a different training set. How "overspecialized" is your classifier to a particular training set? If we

have the best possible model for our training data, how far off are we from the average classifier?

Bias: What is the inherent error that you obtain from your classifier even with infinite training data? This is due to your classifier being "biased" to a particular kind of solution (e.g., linear classifier). In other words, bias is inherent to your model.

Noise: How big is the data-intrinsic noise? This error measures ambiguity due to your data distribution and feature representation. You can never reduce it algorithmically; it is an inherent aspect of the data. (You might, however, be able to add more features that capture this seemingly random variability.)

Back to Table of Contents

Watch: Debug Your Model

When it comes to actually debugging your model, you must first determine the root cause of the error. In the three videos below, Professor Weinberger explains how you determine whether bias or variance is driving up your error. He then outlines both scenarios and provides ideas for improving your model.

Determine the Issue

When your validation error is too high, you must determine if your error is the result of high bias or high variance. Professor Weinberger explains how to determine the source of validation error.

A key skill of a data scientist is to not only train the right algorithm for the right data set, but also to know what to do when things are not working the way they should, or if the error that we finally get on the validation set is too high. We've already looked into model selection. Let's, for example, take a decision tree. Decision tree we have a knob that we can change that increases the complexity; that's the maximum depth of the tree. So as we make the tree deeper, we increase the complexity of the tree. What happens as we increase the tree depth? Well, the training error goes down very, very nicely; the deeper the tree is, the more complex my algorithm is, the more I can memorize my data set essentially, and my training error will go down to zero very, very nicely. What about the testing error? The testing error will first go down as I get a more and more complex model, it can learn more and more complex decision boundaries, but then eventually it will go back up. This is when I start overfitting, start memorizing things in the training data set that don't actually generalize, that are not general trends, but are just true in this particular data set. We've already talked about finding the sweet spot; if you kind of look at this curve for using cross-validation, we're using a validation set. But what if you find the sweet spot and your validation error is still too high? That's a common setting. If your validation error is low enough, then you can ship your product, but if it's too high, you still have to do something; you have to somehow improve your model. So the key is to look into this, and from the perspective of bias and variance. We now know that we can take the testing error and decompose it into bias and variance. There's also the noise term, but the noise term is just a function of the data. For now, let's just focus on the algorithms. So the noise is independent of the algorithm setting and just increases everything. So we can take the test error and decompose the exact key into bias plus variance. As we increase the complexity of our model, the bias goes down very nicely, but the variance creeps up. But the key to understand between bias and variance is that when bias goes down, the

variance doesn't have to go up the same amount, right? The test error is not flat; the test error goes down and then up. So as bias goes down, variance first goes up a little bit, but then variance goes up so much that basically it overwhelms the error. So what we need to know is when you find your sweet spot setting and you evaluate the classifier on the validation set, is your variance too high or is your bias too high? Depending on which one of these two is the case, you should apply very different measures to decrease your error. How can you figure out if your variance is too high or whether your bias is too high? And the way to do this is quite simple: You examine how your classifier behaves as you change the amount of training data. So let's say you take your training data set, and you first train your classifier on just a small percentage; 5% or something. What happens then? Well, if you train your classifier on very little data, your training error will be close to zero, right? it's very simple to get a few data points right. But your testing error will be really high because you haven't learned anything that generalizes here. Now you increase your training data set and you track your training error and your testing error, and what will happen? The training error will go up, because the data set becomes more and more challenging, because it becomes larger, and your testing error goes down and down and down. And you stop the moment you run out of data. Now, there's two different settings you could be. You know that your testing error is above — your validation error is above E, otherwise you would be done. But it could be that your training error is lower than E, and there's a big gap between the two, so your training error does well. Or your training error could be high and could be above E, and your testing error could be very close to it. Those are the two different settings, and those are the bias and the variance settings.

High Variance

High variance occurs when the training error is low but the testing error is high. Professor Weinberger suggests some methods to help keep training and testing error below the expected error.

So the first setting is the setting of high variance. That's the setting where your testing error is high, and your training error is very low. So lower than E. E is the value that we would try to get. In this case, what can you do? Well, so what's happening is you're basically memorizing your training data set, but you're not generalizing; you're overfitting. So one thing you can do is you can add more training data; if you can somehow afford it, buying more data, stick it in, retrain. More training data will have the effect that your training error will go up — that's OK because it's lower than E; who cares, right — and your testing error will go down. Hopefully, the two will kind of meet up below E, or it will push the testing error below E. Another thing you can do is reduce the model complexity. Your model is too complex; it memorizes the training data set, but it doesn't generalize. So you could add a regularizer, or if you have a

decision tree, you could shrink the amount of nodes that you're learning, or you could do other methods that somehow reduce the complexity of your data; for example, feature selection. Finally, you could also do bagging. Bagging is something we will cover in this class. Bagging is very, very effective of taking classifiers that have high variance and reducing the variance of these classifiers.

High Bias

High bias occurs when both the validation error and training error are higher than expected. In this video, Professor Weinberger suggests different methods to lower the two errors.

The second setting that we could be in is the setting of high bias. Here again our validation error is too high; higher than E. That's why we have a problem in the first place. But this time, our training error is also higher than E. So what's happening here is your classifier can't even get the low error on the training data set; even on the samples where you give it the training labels, it can't get the error down. That means your classifier is not powerful enough. What do you do in such a case? Well, in this case, more data does not help. I've seen this many times that people add more data, because companies invest a lot of money in adding more data to the training data set, but they have a high bias setting. It's nonsense. More data would only increase the training error, which is already a lower bound of the testing error. So you would just shrink the gap further but you will stay above E. No; in this case, you have to increase the number of features, for example. So maybe you don't have enough features that describe your data. So let's say you're classifying houses. Maybe there's more features like school district or, you know, the number of bathrooms, etc. There's something you can add in on your data to make it more expressive. If you just look at the algorithmic component, your classifier is not complex enough. So maybe you can, if you have a decision tree, train it for a deeper depth. If you have a regularized classifier, decrease regularization. Or if you have a linear classifier, make it nonlinear, either through Kernelization, or you could learn a decision tree and have the nodes be linear classifiers. Finally, another thing that we will cover later on in this class is boosting. Boosting is an algorithm, a meta-algorithm, that can turn classifiers that have high bias and combine them into a classifier that has very low bias.

Back to Table of Contents

Tool: Model Debugging Cheat Sheet

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

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

Watch: Find Bias and Variance on Sample Data

We know that the testing error has three terms but we do not know what fraction of the error is contributed by each one of them. However, if we knew the true distribution, we could compute the exact mathematical terms in the decomposition. Although knowing the underlying distribution for real data is typically not possible, we do know it precisely if we use artificial data sets.

In this demo, Professor Weinberger describes what happens step by step. Examine the trend lines to find the best level of complexity for the classifier. The bias, variance, and noise should match the testing error.

Video Transcript

We now know that the testing error of a machine learning algorithm decomposes into three terms: variance, bias, and noise. In practice, we don't know exactly what fraction of the error is variance, what's bias, and what's noise, but we have rules of thumb that tell us indicators whether variance may be dominating everything or bias may be dominating everything, and then we apply methods or apply remedies to reduce one of these terms. However, if we had knowledge of the true data distribution, then we could actually compute these terms exactly, or approximately to arbitrary exactness. And we can do this if we have artificial data sets. So in the project that you're doing for this particular module, we will actually assume that the data set is drawn from known Gaussian distributions, where we know all the parameters. So in this particular case, we have two classes — blue and red; plus one and minus one — and they're both drawn from Gaussian distributions that have a small offset; one of zero mean, the other one has a mean of 2, 2. And one thing you can observe — here you can see the blue points and the red points drawn from this Gaussian. And one thing you can see, there are some points — for example, here is our blue points that are actually squarely in the red regime. So these are points that you will always get wrong if you have a good classifier. Because if you see a point here, almost certainly that is a red point, right? It would be wrong to predict it's a blue point. Right? But occasionally it is blue. There's nothing you can do about it; that's just noise in the data. So that's exactly where noise surfaces, right? These are data points that we can't really get right. What you're supposed to do in this particular project is you're supposed to write a function that actually estimates the bias, the variance, and the noise. So in particular what we will do is we will ask you to use regression trees to classify this data set. And the way you estimate bias, variance, and noise is essentially by sampling many, many data sets. So if you actually have hundreds of data sets, you could estimate the variance quite easily, right? For each one of these hundred data sets, you train a different

regression tree, and then you compute the average prediction, and you compute the average squared difference from the prediction across all these hundred different trees for every single data point; that's the variance of your classifier. And similarly, the bias, you can actually compute the average prediction and compute the squared distance of the average prediction from the average label.

And similarly, you can compute the noise by just putting the squared difference of the actual label and the expected label. One thing you can then do is you can plot these three terms as a function of the tree depth, so we can do the whole thing for classifiers of different depth. So the first thing we observe is, well, the green line, the noise, is roughly constant, right? So as the tree depth changes, the noise does not change, because the noise is not a function of the tree; it's just a function of the data. You see a little fluctuation here, and that's just due to the random sampling. The second thing you observe, that if we have a tree depth of zero, that's just a single node, that basically just means we predict the average label. That has zero variance — that doesn't vary at all if you sample different data sets; that makes perfect sense — but has very high bias, because it's biased to a very specific classifier which predicts the average label. As we now increase the tree depth and make our classifier more complex, you see that the bias goes down very, very nicely. Immediately it drops and essentially becomes close to zero, as the tree becomes more and more complex. However, while we're doing this, the variance creeps up, right? So initially the variance is very, very low. As we increase the tree depth, the variance goes up, goes up, goes up, goes up, goes up, goes up. Noise stays roughly constant. The last thing we are plotting, the blue thing here is the testing error, and the dotted line — the black dotted line is bias plus variance plus noise. And one thing you can immediately observe is that if your code is correct, the bias plus variance plus noise should pretty much match the testing error exactly. And the second thing you can see is that these trend lines kind of follow what's happening with bias and variance. So initially, the bias is so high that pushes the testing error up, and you do really, really badly. Then as you keep going, the bias goes down, and your testing error goes down with it. However, eventually the variance creeps up too much, and the testing error goes back up. And the best, probably, setting in this specific data set is a tree depth of 3, where the testing error is the lowest. This is the best tradeoff between bias and variance.

Back to Table of Contents

Bias-Variance Tradeoff

Back to Table of Contents

Module Wrap-up: Decompose the Generalization Error

Now that you have completed this module, you are familiar with bias, variance, and noise, as well as how they can impact the validation and testing error of your machine learning model. You have reviewed the underlying derivations that yield this decomposition and explored options for detecting and mitigating the various causes of error. You've had the chance to apply your conceptual understanding of these terms, and you've completed a project where you implemented these concepts with code.

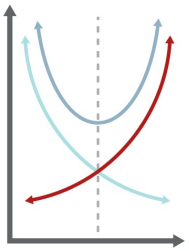
With this experience in hand, you are now prepared to determine the source of error in your model. But knowing where the error comes from is only half the battle. You'll now need to explore ways of mitigating the error once you've been able to isolate it. In the next module, Professor Weinberger shows you how to improve your model when high variance is the issue.

Back to Table of Contents

Module 2: Reduce Variance With Bagging

Back to Table of Contents

Module Introduction: Reduce Variance With Bagging



Bagging is an ensemble method that data scientists use to improve their predictive models. In this module, Professor Weinberger will explain how bagging is an effective and efficient method for reducing the variance of a classifier. He will first explain bootstrapping, an approach to estimate the variance of a classifier. The same principle can then be used to reduce the variance in bagging. You will also explore a new algorithm, Random Forest, that is an adaptation of the CART algorithm you've seen before. At the end of this module, you'll have a chance to implement Random Forest yourself.

Back to Table of Contents

Watch: Estimate Variance With Bootstrapping

We can use bootstrapping to estimate the variance of a classifier. Professor Weinberger provides an explanation of how bootstrapping works to create artificial data sets and how it generates an estimate of variance.

Video Transcript

Bootstrapping is a way to estimate the variance of a classifier. The variance of a classifier can be very, very informative. Number 1, it's important to know because you know your error decomposes into bias, variance, and noise. If you know how much of it is variance, then you know your error is at least that high. Also, it can tell you if it's worth it to apply some methods to reduce the variance of your classifier; for example, reduce the complexity. The variance of a classifier is also very informative about the certainty of a prediction. Imagine, for example, you're in a medical setting and you make a crucial prediction about a patient that informs you whether you should use a certain procedure on that patient — yes or no — that may be risky. The variance tells you how similar the prediction would have been if you had only trained your classifier on a different training data set, right, that essentially tells you how sure the classifier is, whether this prediction is right or wrong. So how does bootstrapping work? How do you estimate the variance of a classifier? It would be easy if you had many different data sets, right? So imagine you, for example, had a thousand different data sets all drawn from the original data distribution. Then you could just train a classifier on each one of these thousand data sets. You have a thousand classifiers, you average their predictions; now you get the expected classifier — an approximation thereof — and you can compute the average squared difference of a prediction with the average classifier. Unfortunately, you can't do this because you don't have a thousand data sets; in practice, you typically only have one data set. So how can you estimate the variance from a single data set? And that's exactly where bootstrapping comes in. Bootstrapping essentially promises the impossible. You cannot estimate the variance from a different data set, and that's why it's called bootstrapping.

Bootstrapping comes from actually the expression of pulling yourself up on your own bootstrap. It comes from a German aristocrat who was a notorious liar, Baron von Munchausen, who tried to impress the ladies by telling them that he had pulled himself up on his own bootstrap out of the snow. And the lady said, "Wow. You know nothing about Newtonian physics." So, in practice, though, you can use bootstrapping to get pretty good estimates of your variance. And here's how it works: Imagine you have a data set of N data points. This data set is drawn from the original data distribution, which you don't know. And what you do is, the first thing is you create new data sets,

and you create artificial data sets by resampling the data that you have. And the way you do this is you just sample with replacement. So imagine your data set is a big bag, all right? And now I create a new data set by just opening up the bag. Without looking, you grab a point out of that bag; you say, "Oh, that point I add to my data set." Now you place it back into the bag because you use replacement, you sample with replacement, and now you pull another data point and — grab another data point and add it to your data set. And you do this N times. So your original data set has N data points, and then you N times draw a data point uniformly at random from this original data set and add it to a new data set. The new data set now has the same number of data points as the original data set, but they're not identical. Why not? Because you could have picked some data points — or you will have picked some data points multiple times, and some data points you may have not picked at all. So now you have a new data set that has some overlap, like basically has some repetitions and consists of all data points that are originally from your training data set. And what you can do is you can now do that many, many times. So instead of just creating one data set, you can actually create a hundred data sets, right, over and over again, right; keep drawing data points, stick them in a data set, draw a new data point, stick them in a data set, etc. So let's say you create a thousand data sets. What you can now do is you can train your classifier on each one of these thousand data sets, and then you use these thousand resulting classifiers to estimate the variance of your algorithm. Of course, that's not exact, right? It's not magic. You can't actually pull yourself up on your own bootstrap, although I invite you to try it out. But what you get is a lower estimate of the variance. So what you will get is always a value that's a little lower than the true estimate. And that's still very useful to know because if you do your own bootstrapping and get a certain estimate of the variance, you know, well, at least that much, possibly more — or probably more of your error is due to variance. And when you make a prediction, you know something about the uncertainty; the uncertainty of your classifier is at least the variance that you obtained as your estimate through bootstrapping.

Back to Table of Contents

Read: Formalize Bootstrapping

Bootstrapping gives us the ability to estimate the variance of a classifier.

Sampling with replacement from the training sample results in the same distribution as the original data distribution (however, samples are no longer independent).

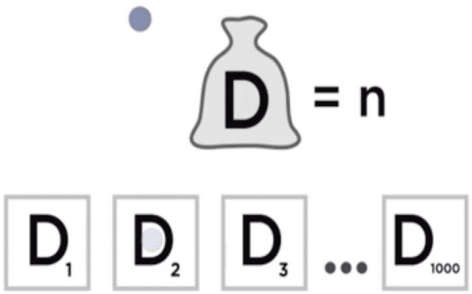
By training models on "bootstraps," we can calculate the mean prediction, then calculate the variance of the model.

One way to estimate the variance of a classifier would be to sample many data sets from the original data distribution, train a classifier on each one of them, and estimate their variance directly (as you have done in the demo in the previous module). However, this only works if you have access to the original data distribution, which you don't in most real-world settings.

We can, however, simulate drawing from the data distribution by drawing uniformly with replacement from the training data set D. Formally, let be a probability distribution that picks a training sample from uniformly at random.

More formally, . with .

We sample the set ; i.e., , and is picked **with replacement** from . This way we obtain data sets, , each one with elements subsampled from . Note, however, that these data sets are *not* all identical to . The reason is that we may pick some samples multiple times, and others not at all. In fact, in expectation, the intersection of any and is only 63%, and the remaining samples of are repetitions of each other. Because each is picked with replacement from , they are also drawn from the same data distribution, but the samples are *not independent*.



In order to estimate the variance of our algorithm, we train one classifier on each data set . We can then compute the average classifier of these bootstraps: .

The variance of our algorithm is then estimated to be , where denotes the validation set.

Notice that . In other words, the average of these classifiers does not converge to the true expected classifier , which we would obtain if we had truly sampled different training data sets from the original data distribution. The reason is that our data sets are not independent of each other (as they are all subsampled from the original data set .) However, in practice, bagging still reduces variance very effectively.

Back to Table of Contents

Watch: Reduce Variance With Bagging

Bagging, also known as bootstrap aggregating, is an effective way to reduce high variance. In this video, Professor Weinberger explains how to use this method to create an ensemble of classifiers that typically have very low variance.

Video Transcript

When you're in a setting with high variance, a very effective way to reduce the variance is through bagging. Let us just take a step back and revisit the variance term of a classifier. So variance is basically the expected value, where the expectation is the training set of the test point x, and inside the expectation is the square loss, the square difference of the hD of x — that's the prediction of the classifier trained on D that it makes on x — minus the expected prediction that the expected classifier would make. So essentially there's some value in the prediction the expected classifier makes, and your hD, depending on D, makes some predictions that are somewhat close, somewhat around this expectation. The variance measures how far are you off in a squared sense from this expected value. You can, of course, reduce the variance if you had many different data sets. If you had like a thousand data sets, then you could train a thousand classifiers hD. And what you could do is you can say, "Well, I could find a new h-hat," which is the average of these predictions. And one thing you know is as the number of data sets close to infinity — so if I go from 1,000 to 10,000 to 100,000, and average more and more with these classifiers, then my h-hat becomes closer and closer to the expected classifier. And what will happen is that the variance will naturally go to zero, because h-hat will become exactly h-bar. Well, that's in fantasyland, but in the real world, we only have one training data set. So we don't have thousands of data sets. However, now we know one new trick, and that is bootstrapping. Bootstrapping is a way of obtaining multiple data sets from a single data set. So we have one training set, that D, and what we can do with bootstrapping is we can generate m different data sets out of this one training data set that are drawn from the same distribution as the original data set; however, they are not independent. That's OK. So let's just do this; let's just draw m such data sets. For each one of these m data sets, we train a classifier — let's call it h1 to hm — and what we do is we just say, "OK, our final classifier is just the average prediction of all of these classifiers." This new average does not become exactly the expected classifier because we're using bootstrapping. You're not estimating the variance exactly. All our data sets are drawn around the one training data set that we have. However, there is a certain amount of variance that originates from the fact that some points are present or missing, etc., and that's exactly what we're capturing with bootstrapping, right? Because sometimes we are taking data points twice, sometimes we are omitting data

points, right? A good chunk of the variance can just be attributed to that kind of stuff happening. So what we are doing here is essentially when we are averaging these classifiers, we get a much more stable classifier that varies a lot less across different original data sets D. And that's exactly what bagging is. So what we do in bagging is you take a classifier that has high variance, you generate more of these data sets — use bootstrapping to generate data sets. you average the obtaining classifiers, and this ensemble of the classifiers is a classifier that tends to have pretty low variance. The nice thing is typically you don't increase the bias to bagging, so it's actually a very, very effective method that you can typically use in almost all settings and that's very effective at reducing the error of your classifier.

Back to Table of Contents

Read: Formalize Bagging

The weak law of large numbers tells us that the sample mean of a random variable approaches the population mean as the number of samples goes to infinity.

We apply this to classifiers and average many classifiers trained on different data sets. If the data sets were truly sampled independently from the original data distribution, the variance term would vanish.

As we do not have access to the underlying data distribution, we sample bootstraps instead. Although the resulting training sets are not independent, the resulting averaged classifier still has much reduced variance.

Remember the bias-variance decomposition:

Our goal is to reduce the variance term: .
For this, we want .

Weak Law of Large Numbers

The weak law of large numbers says (roughly) for independent and identically distributed (i.i.d.) random variables with mean , we have,.

In other words, as the number of samples approaches infinity, the mean of a sample approaches the true mean of a random variable.

Apply this to classifiers: Assume we have m training sets drawn from . Train a classifier on each one and at run time take the average result obtained across all classifiers:

We refer to such an average of multiple classifiers as an **ensemble** of classifiers.

Idea: If , the variance component of the error must also vanish; i.e., .

Problem: We don't have data sets ; we only have D. Recalling the previous lectures, how can we use the one data set we have to simulate many different data sets?

Solution: We use bootstrapping to obtain the data sets by sampling them with replacement from the original data set . Once again, we train a classifier on each one of these bootstraps and compute their average . This average is the **bagged** classifier.

Notice that although gets closer to , we do not have full convergence as the weak law of large numbers suggests; i.e., . In other words, the average of these classifiers does not converge to the true expected classifier , which we would obtain if we had truly sampled different training data sets from the original data distribution. The reason is that our data sets are not independent of each other (as they are all subsampled from the original data set .) However, in practice, bagging still reduces variance very effectively.

Back to Table of Contents

Watch: Benefits of Bagging

Bagging has two appealing aspects that make it practical in the real world: You can train all of your classifiers in parallel, and you can determine an estimated validation error without having to do cross-validation. In this video, Professor Weinberger further explains the bagging method and why it is such a useful approach.

Video Transcript

So bagging is an extremely effective method to reduce the test error of a classifier that has high variance. But bagging also has two really compelling aspects to it that can make it very, very practical in real-world settings. Number 1 is that when you do bagging, you can actually train all the different classifiers that later on you ensemble in parallel. The reason is that we subsample all these data sets, and then on each data set we train the classifier independent of the other classifiers. So If you, for example, have a cluster of many computers, or you have a computer with multiple cores, you could just launch off many jobs, train them all in parallel, at the end get all your classifiers back. And often it can be that you train thousands of classifiers, and only takes a little longer than training a single classifier, depending on how many computing resources you have. A second advantage of bagging is the out-of-bag estimation of the test error. And this is a little more subtle, but it's really, really cool and it's pretty unique to bagging. When we do bagging, we essentially take the original training data set and subsample m data sets with replacement. However, one thing that happens is every time we subsample a data set, there's also some data points that we didn't pick, right? Some data points we did pick from the training data set into our subsample data set, but some data points we held out. We call this a leave-out bag. And this is a little similar to a concept that we are very familiar with, and that is the splitting of a data set into train and validation. So every single time — you can view this out-of-bag set as a validation data set, and every single time you create a training data set, you automatically create a validation set, this out-of-bag data set of data points that we didn't pick, that we didn't include in this particular data set. So when we create m data sets, we also automatically, implicitly, create m validation data sets.

The nice thing is now that we can actually get an estimate of the validation error without ever sacrificing any data. So what we do is for any given data point Xi, we just look at all the data sets where Xi fell into the validation portion. So for example, here we have five different data sets that we created; they lead to five different classifiers, h1 to h5. And we also have five validation sets; this particular data point Xi falls in the validation set three times. So what we do is we only take these three classifiers, h2, h4, and h5. Those were trained on data sets that did not include the data point Xi. And

if we average — just ensemble these three classifiers and apply it to Xi, then this is a legitimate validation error of Xi. And you can do this for every single data point. So for every single data point, we look which classifiers were not trained on this particular data point, or which classifiers was this data point in the validation set? And then we only ensemble this subset of classifiers, and what we get is an estimate of the validation error. And if we average all of these errors, then we actually get the true estimate of the validation error. The beautiful thing is typically when we obtain a validation error, we have to take our data set and split it in training and validation. So there is some portion of the data that we are not using during training. The amazing thing with bagging is that we actually train our full classifier on all the data, yet we get this leave-out estimation of the validation error for free. And we can basically estimate the test error without ever having to deal with cross-validation or any of these things. In practice, this can be very, very useful, and it's also a good way to decide when do you want to stop bagging? So as you add classifiers to your ensemble and you monitor the leave-out error, and at some point that plateaus, then you know it's time to stop; I'm not getting any further improvement out of more classifiers.

Back to Table of Contents

Watch: Random Forest

Random Forest, a new algorithm Professor Weinberger will introduce here, is a variation of the CART algorithm that is bagged with one small modification: You only consider a subset of the features, instead of all of the features, when making a split. In this video, Professor Weinberger explains the process and describes the aspects of Random Forest that make it effective.

Video Transcript

In principle, bagging works with any kind of classifier that has high variance. But some classifiers are better suited than others. And in particular, CART trees are extremely well suited for bagging. In fact, they're so well suited that there's a new kind of algorithm called Random Forest, which is essentially a variation of CART tree that is bagged. Why are CART trees so well suited for bagging? It's very simple. Number 1, they have really high variance. If you build a CART tree all the way to the bottom without depth limit, they essentially always get zero training error. Number 2, they basically have no bias. So if you can reduce the variance, there's essentially no bias error left so we can really reduce the overall error of the classifier a lot. And number 3 — and this is crucial — CART trees are really fast to train and really fast to evaluate. So if we bag a thousand trees, it doesn't take us that long to train them and it doesn't take us that long to evaluate them. So how does Random Forest work? Random Forest is essentially just a bag tree with one small modification. And that is every time you build trees, you consider the best split as you're building the tree in the tree function. And when you look at the best split, you don't look at all the dimension but you subsample k dimensions randomly and you only look at these k dimensions. So let's say, for example, I have a hundred dimensions and k is 10, then every time I make a split I just pick 10 dimensions at random. And only in those 10 dimensions do I look for the best split, and that's the split I'm making. The next time I make a next split in the tree, I again pick 10 new dimensions randomly and look for the best split. The reason we are doing this is because this increases the diversity of the trees and captures the variance that we get by splitting our different dimensions.

So the key with bagging is that we try to capture as much variance as possible and then reduce it by averaging. One part of the variance is obtained through bootstrapping, which basically says, "Well, the classifiers may vary because we include or exclude certain data points." That's exactly what we capture with bootstrapping. And the second thing is we could split on different dimensions. That's what we capture by randomly subsampling dimensions. One side effect is also that it's faster, right, because you only look at k different dimensions so you don't evaluate as

many splits when you build a tree. Random Forests are extremely popular, and actually they are one of the best machine learning algorithms in my experience. There's two really nice aspects about Random Forests. Number one is they're insensitive to any feature scaling, etc. So, for example, one dimension is — in one unit let's say we have data, and one dimension is the blood pressure, the other one is the gender, etc, All of this doesn't matter. Other classifiers you have to normalize these features and make sure they all are within the same region. In Random Forests, you just split anywhere; it's extremely insensitive to this kind of stuff. So it typically works out of the box; you just throw a data set at it and it learns the right thing. Number 2, Random Forests are naturally non-linear so you don't have to worry about, you know, "Is this a linear problem," etc.; you just run Random Forest and typically it works really well. The third thing is — and this is a key aspect of Random Forest — is that it only has two hyperparameters. The first one is m; how many trees do you average? How do you set m? Well, it's very simple: The more, the better. So you just set m as large as you can afford. That's very easy to set. It's not a tradeoff. The second thing is k; how many features do you have to subsample every single time you make a split? And here we have a really good rule of thumb, which has some theoretical justification. What you do is you just set k to be the square root of the number of dimensions. So for example, if you have 100 dimensions, you set k as a square root of 100, is 10. Now you have an algorithm that essentially has no more hyperparameters, right? You just let it run as long as you can afford and you set k to be the square root of d. So, in practice, what I do when I get a new data set? I just run Random Forest out of the box and see how well it does, and typically that's a pretty good estimate of how well you can do in machine learning.

Back to Table of Contents

Read: Random Forest

Random Forests harness the power of bagging to reduce the variance of decision trees (which are very high-variance/low-bias classifiers).

Random Forests introduce randomness in two stages: sampling the data with replacement and sampling a subset of features at each split.

The final output of a Random Forest averages the predictions of all the component trees.

Let's dive a bit deeper into how Random Forest works. A Random Forest is essentially nothing else but bagged decision trees, with a slightly modified splitting criteria. The algorithm works as follows:

Sample data sets from with replacement.
For each , train a full decision tree (max-depth=) with one small modification: Before each split, randomly subsample features (without replacement) and only consider these for your split. (This further increases the variance of the trees.)
The final classifier is .

The Random Forest is one of the best, most popular, and easiest to use out-of-the-box classifiers. There are two reasons for this:

The RF only has two hyperparameters, and . It is extremely *insensitive* to both of these. A good choice for is . (where denotes the number of features). You can set as large as you can afford.
Decision trees do not require a lot of preprocessing. For example, the features can be of different scale, magnitude, or slope. This can be highly advantageous in scenarios with heterogeneous data; for example, the medical settings where features could be things like blood pressure, age, or gender, each of which is recorded in completely

different units.

Useful Variants of Random Forest

Split each training set into two partitions , where . Build the tree on and estimate the leaf labels on . You must stop splitting if a leaf has only a single point in in it. This has the advantage that each tree and also the RF classifier become consistent.
Do not grow each tree to its full depth; instead, prune based on the leave-out samples. This can further improve your bias-variance tradeoff.
Random Forests are also extremely useful to determine the importance of a particular feature. To this end, one can keep track of the accumulated reduction in loss that is achieved by splitting on a particular feature. Many variants of this approach are known and used in practice.

Back to Table of Contents

Tool: Random Forest Cheat Sheet

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

As you have seen, Random Forest creates an ensemble from CART classifiers 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.

Back to Table of Contents

Random Forest

Back to Table of Contents

Module Wrap-up: Reduce Variance With Bagging

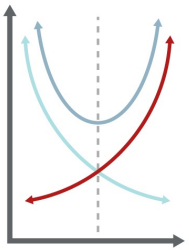
After completing this module, you should have a good understanding of how to apply bootstrapping and bagging to your algorithm. You also explored a new algorithm, Random Forest, which you had a chance to implement yourself. Ensemble methods allow you to mitigate the shortcomings of machine learning algorithms. In this module, you learned how to use bias to reduce variance. In the next module, Professor Weinberger introduces yet another highly practical ensemble method; this time, however, to reduce bias.

Back to Table of Contents

Module 3: Reduce Bias With Boosting

Back to Table of Contents

Module Introduction: Reduce Bias With Boosting



Another ensemble method, this one known as boosting, can help you improve your model. In this module, Professor Weinberger explains the method of gradient boosting and how you can use it in the form of yet another algorithm, Gradient Boosted Regression Trees (GBRT), to improve the accuracy of your model. Gradient boosting is similar to bagging, except that instead of reducing the variance of a classifier, it reduces its **bias**. You'll have a chance to explore the underlying concepts of GBRT and implement it yourself in the final notebook project of this course.

Back to Table of Contents

Watch: Gradient Boosted Regression Trees

Before we dive into the more general (and slightly more abstract) gradient boosting, Professor Weinberger describes a specific instance: Gradient Boosted Regression Trees (GBRT). GBRT combines trees in ensembles, similar to bagging. However, instead of using full trees (with high variance and low bias), here we train trees of very limited depth (with low variance and high bias). GBRT is extremely powerful and popular, especially in web search engines where we have many different features describing a page, the query, and the relationship between the two, and we need to predict if a user will click on a certain page or not. In this video, Professor Weinberger shows how the Gradient Boosted Regression Trees algorithm works with an example and compares them to Random Forests.

Video Transcript

In principle, you could take any loss function and any kind of weak learner and combine them for gradient boosting. But some combinations are so popular that they have gotten their own name. One such example is Gradient Boosted Regression Trees. As the name suggests, here our weak learners are regression trees. Typically, we take small regression trees of maybe size depth 4 or 5 or something. And the loss function that we are using is the square loss. So, what we are doing, our algorithm takes this input of data sets and a CARD algorithm. And initially, we say our ensemble is all zeros, so we predict zero for every single data point. And now we iterate through the data set. And there's a very, very nice interpretation of this particular algorithm. This comes out the derivation of gradient boosting. If you plug in the square loss, you will see that actually you will obtain exactly this algorithm. What we get is that every single iteration, we compute the residual of the label and the current predictions. So what that means is Ri for every single data point denotes the difference of Yi, the true label, minus the current prediction. So essentially what we're saying is, here's our current prediction. Here's a true label. They're not the same, right, because we're not done yet. So the difference is what we still need to predict. And we are basically slowly closing that gap between our prediction and the true label. So, Ri is what we call as residual; that's basically what we still have to predict, still have to add to our classifier. And then we learn a new CARD tree that predicts for every single data point not the true label, but instead that residual. And then we add that classifier to our ensemble. Here, in this case, the learning rate is fixed, and typically it's a value of 0.1 or 0.01 or something; we also call it the shrinkage factor. And essentially what we are doing is we're saying each weak learner is not very good. So we're adding it to our ensemble, but just a teeny little bit. The weak learner points us in the direction of a good solution, and we just take a small step in this direction. And then we learn a new direction, and

we take a small step in that direction. Very much like gradient descent.

I can also walk you through a little example. So what you can see here is a function that we are trying to learn with gradient boosted trees. But our gradient boosted trees only have a depth of 3, so they're not very good. Here you can see how the function is approximated with the tree of depth 3. So, that's not terrible, but it's also not great either. So what we're doing is, with this approximation, we shrink it by . So let's say is 0.1. So we shrink it down. And essentially what we are doing is we subtract this amount from our label Yi. Now we get a new target function, and we learn a new weak learner. And once again, we shrink this weak learner, and we subtract it from our target function. And we do this over and over and over again until eventually the residual is zero. The function that we're trying to learn is zero. That means at this point, we've learned the function completely. There is nothing left. That means that the prediction matches the label. Gradient boosting is really, really popular; in particular, in search engines. In search engines, we have many different features that describe a web page and a query and the relationship between the two. And the nice thing about gradient boosted trees is it's a very fast algorithm so you can run through it on many, many web pages very quickly if someone types in the query. And the other thing is, similar to Random Forest, it makes very few assumptions on the particular types of features that you have. So people can have very diverse features because trees are so robust with respect to that, and it's a very, very flexible algorithm. In my experience, gradient boosted trees typically work a little better than Random Forests, but they also need a little more love, because the number of trees that you're adding is actually a true parameter; more is not always better. And also the step size is a parameter you need to tweak. So, you're doing a little better, but at the cost of two hyperparameters that you need to tweak.

Back to Table of Contents

Read: Gradient Boosted Regression Trees

Gradient Boosted Regression Trees (GBRT) is an algorithm that results from gradient boosting applied to regression trees — similar to Random Forests, which is bagging applied to regression/classification trees.

Gradient boosting reduces the bias of a classifier, analogous to the way bagging reduces its variance.

GBRT learns an ensemble of trees, and each tree is trained to correct the *residual* (the remaining error) of the existing ensemble.

CART trees have many strong advantages: They are extremely fast to build and to execute, they are insensitive to feature scaling, and they require very little storage. However, they are typically not competitive by themselves, and the reason is that it is hard to find the right tradeoff between high variance and high bias. If the tree is built all the way to its full size, it will almost always (unless two samples have identical features but different labels) achieve zero training error but will strongly overfit to the training set. In other words, the classifier is *low bias* but *high variance.* If, on the other hand, a tree is trained with limited depth (e.g., maxdepth=4), the classifier will have a hard time modeling the training data set and we will obtain a classifier with *very high bias* but *lower variance*. Unfortunately, setting the maximum depth (or even the number of splits) is a crude way of trading off bias with variance. Because the depth is inherently integral in nature, we cannot perform the fine-grained adjustment that is typically necessary to get the balance between bias and variance "just right."

Random Forests, as introduced in the previous module, allow us to circumvent this problem. Instead of worrying about the bias-variance tradeoff during tree construction, we train many full trees on bootstraps (with a slightly modified splitting criterion) and reduce their average by averaging their predictions (through bagging).

Gradient Boosted Regression Trees (GBRT) is a similar algorithm, but it does not use **bagging to reduce variance**, instead using **boosting to reduce bias.**

Boosting Weak Learners

Boosting is a general method to reduce the bias of *weak learners.* Weak learners are essentially classifiers that cannot achieve zero training error on a given data set — a typical sign for high bias. Similar to bagging, boosting combines many such classifiers together into an ensemble. However, this time we assign weights to each weak learner; more specifically, let our individual classifiers be with weights , where each . Then our final classifier becomes

The key to boosting is to understand how the classifiers and weights are obtained.

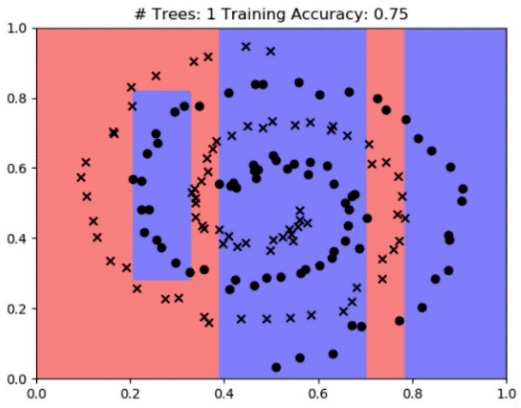
Implementing Gradient Boosted Regression Trees (GBRT)

Boosting is a general concept (similar to bagging), and there are many different specific boosting algorithms. Here we will focus on one of the most popular algorithms, Gradient Boosted Regression Trees (GBRT). Similar to Random Forests (which is essentially bagging for CART trees), GBRT is gradient boosting for CART trees.

In GBRT, each tree is trained not to predict that the label directly, but to predict the *residual* of the current ensemble. In other words, we are trying to learn the difference between the label and the current prediction . Initially, we set the classifier to predict for all inputs. This means that the very first target is for all inputs . However, because the trees are high bias, we know they have high training error. In other words, they won't be able to predict the training labels correctly, and there will be a residual. The second trees are trained to predict exactly this remainder and added to the trees, and so it continues with each iteration. Note that the weights are set to a constant identical for all trees.

Here is the pseudocode for GBRT and a visualization of a GBRT classifier trained on a binary "spiral" data set.

GBRT($(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, D, T, α)
H $\leftarrow 0$;
for *t=1:T* **do**
  for *i=1:n* **do**
    $t_i \leftarrow y_i - H(\mathbf{x}_i)$;
  end
  $h_t \leftarrow CART((\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_n, t_n), maxdepth = D)$;
  H $\leftarrow H + \alpha h_t$;
end
return H



Back to Table of Contents

Watch: Gradient Descent in Functional Space

Gradient boosting is an approach used to reduce a loss function through an ensemble of weak learners. As it turns out, this approach is very similar to gradient descent, except that in each iteration we update the *ensemble* instead of the model parameters. Professor Weinberger explains this connection and how this works in general terms.

Video Transcript

Remember gradient descent; there we had a loss function — for example, the logistic loss or the squared loss — and we had a linear classifier; that was the context in which we introduced it. And in every single iteration, what we did is we computed the gradient of the loss with respect to the parameters W, and then we said W becomes W minus times this gradient, where is a step size and the gradient just basically says we take the loss function and we approximate it locally as a line. Which direction would we have to follow to decrease this line? Turns out gradient boosting is quite similar to that. So essentially what we have is we have a function H which is our ensemble, and in every single iteration we update H to become H plus times some new h, but that approximates the gradient of the loss function with respect to our H. So essentially what we are saying is we would like to minimize the loss function. And what we're doing in every single iteration, we adding h to it, which points in the direction that reduces the loss. If you formalize this, you can basically say, "Well, every single iteration we want to find the H that minimizes the following term." We sum over all the data points i, and for every single data point, we have a loss of this particular data point, which is H of Xi, which is the ensemble we've had so far, plus times h of Xi. This is what we are adding in this situation. s the step size; how far do we want to go? And H is basically the new weak learner that we're adding and hopefully points in the direction that decreases the loss. Similar to gradient descent, what we can do is we can say locally this function is linear, then we can do Taylor's approximation. Taylor's approximation essentially decomposes this term inside into the loss of H of Xi, plus times, and then comes the gradient of the loss with respect to H of Xi times h of Xi at every single data point. And this is not very hard, right? So essentially what we're saying is while we want to — the first term doesn't actually depend on little h so it doesn't matter. And the second term basically says we compute the gradient of the loss function with respect to the prediction at every single data point, and basically we see how aligned that is with the weak learner that we want to find. So we want to find the weak learner that is maximally aligned to this. As it turns out, when we pick the loss function to be the exponential loss, then we can exactly derive AdaBoost that way. The gradient of the loss with respect to H of Xi becomes exactly the weight that we are basically computing to weigh the different data points. And what we obtain is

we are minimizing the weighted training error. But the beautiful thing of this framework is that we are not restricted to the exponential loss; we can use any kind of loss function. And that's exactly what people have done. So, a common one, for example, is the square loss or the logistic loss, etc., and each of these leads to different algorithms with different properties. AdaBoost is always a little special because it was the first one that was invented and is a very neat way of setting the step size in close form. So essentially now we have an interpretation of this. What we are doing is we find the step size that actually minimizes the function. For the other algorithms, you have to pick the step size as a hyperparameter. One thing that's nice, if you use, for example, the squared loss, then you get a boosting algorithm for regression.

Back to Table of Contents

Read: Gradient Descent in Functional Space

Gradient boosting (as the name suggests) is highly related to gradient descent.

In each iteration of boosting, we add a weak learner that adjusts the training predictions to be closer to the true labels (similar to a gradient step).

Boosting is a very general concept with many instantiations, some of which have specialized names such as GBRT, LogitBoost, and AdaBoost.

Gradient Descent

Remember gradient descent? We used it to minimize a loss function with respect to some parameters (for example, to learn a logistic regression classifier). Each iteration we updated the parameters with a gradient update:

Here, was the step size and the gradient of the loss function with respect to the parameters .

Gradient Boosting and Gradient Descent

Gradient boosting is actually quite similar to gradient descent. We are trying to minimize a loss function ; not with respect to a parameter vector, however, but with respect to a classifier . We assume that this classifier is an ensemble:

Boosting is an iterative algorithm, and each iteration we learn a new function that we

add to our ensemble. In other words, the iterative update per iteration is

.

If you compare this equation with the update from gradient descent, you should realize an undeniable similarity. So what should our be in each iteration? By a very similar argument as we have made to derive gradient descent, should resemble the negative gradient; i.e.,

Gradient With Respect to a Function

It may seem strange to think of the gradient of a loss function with respect to a function . How exactly are we supposed to do this? Well, as it turns out, we don't have to. At the end of the day we are only evaluating on the training data points, so we can pretend that a function is really just an -dimensional vector of all the values it predicts on our training points; i.e., . To illustrate this, let us take a step back and reconsider the loss function . A loss is really a measure of how well a classifier is performing on the inputs in a given data set. For example, we can define the squared loss as

.

Note that it evaluates the function only on the training inputs. So as far as the loss is concerned, we can view as an -dimensional vector. Suddenly it is no longer so weird if we take the derivative ; it is nothing else but the derivative with respect to a vector — similar to the vector in gradient descent.

Gradient Approximation

So we regard our function as an -dimensional vector , where the i-th dimension corresponds to the evaluation of the function on input . Now we can compute the gradient , where . As an example, let us consider to be the squared loss as defined above. Here, we obtain

This quantity is easy to compute for each training input.

Now, if functions really were just -dimensional vectors, we could simply perform an update and subtract from . Unfortunately, that's not true. is still a function, and we can only add functions together (not functions and vectors). So what we need is a function that behaves just like the negative gradient ; in other words, should have a similar effect as if we performed the gradient descent step in our -dimensional vector

space . To this end, we try to find a function that mimics across all our training points. Formally, we are trying to find

where , the negative gradient w.r.t. ; i.e., . The resulting function will be very similar to the negative gradient across all training points, so if we add it to our ensemble , we are essentially taking a gradient step to bring the training predictions closer to the training labels.

Note: *There are other ways to find the weak learner that is most like the negative gradient. For example, one can also use . In the case of regression trees, these two approaches are essentially equivalent.*

Gradient Boosted Regression Trees

In GBRT, we use the squared loss as our loss function . The gradient then becomes and . This term has a very nice interpretation: The negative gradient is the residual of the current classifier ; in other words, it measures the difference between the current prediction and the label for each training data point. The next weak learner will try to predict that difference, and if we add it to our ensemble, we are taking a small step towards the label across all dimensions (i.e., for each training point). One beautiful aspect of GBRT is that the search for becomes a simple call of the CART algorithm, with inputs .

Other Boosting Algorithms

There are many variations of boosting. For example, if the loss function is the logistic loss function, then we obtain **LogitBoost**. Or if the loss function is the exponential loss , we obtain **AdaBoost** (where the step size is found via a closed-form line search). In AdaBoost the weak learner is typically found by optimizing . The negative gradient takes on the form . If the labels and the outputs of the weak learners are restricted to +1,-1, this can be interpreted as minimizing the weighted training error — another intuitive interpretation of boosting (although the details are slightly beyond the scope of this module).

Remark About Loss Functions

One aspect of gradient boosting that can be confusing is that there are **two** loss functions. There is the global loss function that we are trying to minimize, but there is also the loss that the weak learning algorithm minimizes in order to find the next classifier that best aligns with the negative gradient. Often both of them are the

squared loss, but they don't have to be. For example, you could choose the logistic loss for and then use standard regression trees (that minimize the squared loss) to find the weak learner in each iteration to match the negative gradient. This allows you to minimize a *classification loss* using *regression trees*.

Back to Table of Contents

Tool: GBRT Cheat Sheet

Use the GBRT Cheat Sheet to review how the algorithm functions.

As you have seen, a set of weak classifiers 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 Regression Trees when implementing the algorithm yourself.

Back to Table of Contents

Gradient Boosted Regression Tree

Back to Table of Contents

Module Wrap-up: Reduce Bias With Boosting

In this module, you explored the second learning algorithm to improve CART, boosting. Boosting is, in fact, a general method that reduces the bias (and sometimes even the variance) of classifiers. It works best when classifiers suffer from high bias and are fast to evaluate, as it creates an ensemble of many classifiers that each need to be evaluated during testing. A very popular boosted algorithm is Gradient Boosted Regression Trees (GBRT), which boosts limited-depth regression trees. Regression trees are a perfect fit for boosting, as they are high bias (when limited in depth) and extremely fast to evaluate. In this module, you implemented your own version of GBRT. You now have the tools to improve your model accuracy using bagging and boosting.

Back to Table of Contents

Read: Thank You and Farewell

Kilian Weinberger
Associate Professor
Computing and Information Science
Cornell University

Congratulations on completing "Debugging and Improving Machine Learning Models."

In this course, you have learned about the bias-variance decomposition, bagging, and boosting. Personally, I find these to be among the most important topics in machine learning. Understanding the bias-variance tradeoff, identifying whether an algorithm suffers from high bias or high variance, and being able to apply the right tools to improve the results are what distinguishes true data scientists from amateurs. Bagging and boosting (in particular, Random Forests and GBRT) are amongst my favorite algorithms that I use all the time for real-world applications.

From all of us at Cornell University and eCornell, thank you for participating in this course.

Sincerely,

Kilian Weinberger

Back to Table of Contents

Glossary

Bagging

Bagging (bootstrap aggregating) reduces the high variance of a classifier by averaging several models. You create m datasets by bootstrapping, train a classifier on each, and for the final classifier use the average prediction of all the classifiers. Bagging does not increase bias, but can substantially reduce the error from variance. Bagging is most effective with high variance classifiers and works best with classifiers that are fast to evaluate, as it otherwise increases the running time substantially (due to the additional model evaluations). Bagging is one of two meta-algorithms that make the CART algorithm highly competitive. The other is Boosting.

Bias

One of three terms that contribute to test error, which is the sum of bias, variance, and noise. Bias expresses the error that the expected classifier would make. If you trained on an infinite amount of data and averaged all the classifiers, the bias error would remain. This happens, for example, when you make the wrong modelling assumptions, such as training a linear classifier on a non-linearly separable dataset.

Boosting

Boosting reduces the bias (and also sometimes the variance) of a classifier. Similar to bagging, it creates an ensemble of models, however, these are not trained independent of each other. Instead, each new ensemble member is trained explicitly to reduce the error of the existing ensemble. Boosting works best with classifiers that have high bias and are fast to evaluate. Boosting is one of two meta-algorithms to make the CART highly competitive. The other is Bagging.

Bootstrapping

A way to estimate the variance of a classifier. You create artificial datasets by resampling the data you have, using sampling with replacement--drawing a data point, and adding it to a new dataset. This sampling is performed N times, where N is the number of data points in the original set. You create a second set of the same size, with different data, because you can pick some data points multiple times and others not at all. You can create as many datasets as you need in this way and train a classifier on all of the sets. You can then estimate the variance of your model by computing the variance of the predictions across models. This estimate is an under-estimate of the true variance, so you will know that the error induced by variance is at least the amount estimated through bootstrapping.

Noise

One of three terms that contribute to test error, which is the sum of bias, variance, and noise. Noise is the expected difference between the label of a data point and its expected label. When there is noise, the label of a given feature vector $\mathbf{X}$ can be rather uncertain. In machine learning, noise indicates you

can't trust your labels, features, or your features are not sufficient. You can reduce noise by cleaning up the data or adding more features. A common case for noise is that an important factor that influences the label is not captured by the features. For example, when trying to predict house prices without information about school district in any of the features, the changes in sale price due to school district will appear to be random for any given $\mathbf{X}$.

Random Forest

A variation of a bagged classification and regression trees (CART). In Random Forests, each split of each CART tree considers only a randomly selected subset of the features and ignores all other features (which increases the variance of the individual CART trees). Random Forests are particularly useful as they are very resistant to overfitting, completely insensitive to hyperparameters, unaffected by feature scaling, and can provide unbiased estimates of the testing error from the training data (without any validation data).

Variance

One of three terms that contribute to test error, which is the sum of bias, variance, and noise. Classifier variance expresses how consistent the predictions of a classifier are if it is trained on different training data sets. High variance is a sign that the classifier is overfitting to the particular data set it is trained on.