# Module 10 Transcripts

## Video 1: Introduction to module ten [02:43]

Welcome back to module ten, the second part of our discussion of Decision Trees. In the previous module, we have learned how to construct individual trees. We have seen that trees are very powerful and that they can learn highly nonlinear relationships. But its power comes at a price. It is very easy to over fit trees to the training data. We have learned how tree pre- and post-pruning can be used to control this overfitting. However, in this module we will learn an alternative and often more powerful approach to avoid overfitting. Instead of constructing a single tree, we will construct a whole forest of trees, which we call a Tree Ensemble. How do multiple trees help to reduce overfitting though? Each individual tree in such a Tree Ensemble, will differ from the other trees by perturbing the training data. We will use bootstrapping for this purpose. This will make these trees more robust against noise because each tree captures a different perturbation of the input data. For any new data point, we will then classify or predict the output variable of that point through either majority vote in the case of classification trees or by averaging in the case of a regression tree. So, if Tree Ensemble tend to be better at avoiding overfitting, why would we ever want to use individual trees? It turns out there is a downside from going over from individual trees to tree ensembles, and that downside is a loss in interpretability. Interpretability is a major selling point for individual trees. It is often the case that insights that can be understood by practitioners are much more likely to be adopted by them as well. So, ultimately, it will be up to you to decide what matters more in the application, the predictive performance, or the interpretability of the model. At the end of this module, we will also take some time to study how an individual tree as well as tree ensembles can be constructed in Python, and we will apply what we have learned to real life data set.

## Video 2: Bagging [09:16]

The same trees are very powerful, but they can suffer from a high variance. In fact, a simple experiment is the following. Imagine you split up your training data randomly into two halves. And you look at the decision trees that you get from classifying each half. In many cases, those two decision trees, we'll look substantially different. Phis indicating a high presence of variants in that estimator. How can we reduce the variance? It turns out that we can reduce the variance by going over from one decision tree to a collection of decision trees. And the key idea here is to combine bootstrapping with averaging, leading to the so-called bagging algorithm. Let's first have a look at why averaging might be able to help us reduce the variance. And to this end, I want to draw on an analogy of simpler random

objects, not trees, but random variables.

Let's consider the case where we have IID independent and identically distributed random variables, X1 up to Xn and day follow a mean mu and a variance sigma square, neither of which we know. And let's imagine we want to estimate the mean mu of these random variables. It turns out that any of these random variables, whichever one I take, is an unbiased estimator for the mean mu. So, in other words, I could take the realization of any of these random variables and use that as an estimator for my mean mu that I would like to learn. Turns out we can do better than that. And the reason we can do better than that is we can average the random variables. To send. Let's first recognize that the variance of our individual estimator here would be sigma square. That's how we define sigma squared. Now imagine instead, we would use the following estimator for the mean mu. We would take 1 over n times X1 plus X2 plus all the way up to xn. So, we take the average of these random variables. By the properties of the expectation operator, we know that we can take out the one over n term. And since the expectation operator is additive, we can write this as the following expression here.

Now we know that each of these individual expectations here is going to be Mu. All of those are unbiased estimators. So, we have 1 over n times n times Mu, which is going to be mu. So, in other words, once the average as expected is also an unbiased estimator. Now let's have a look at the variance of the average of these random variables. Meaning how and how much does our estimator fluctuate? The variance of the average of all these random variables is equal to one over n squared. This is a property of the variance times the variance of X1 plus the variance of X2 plus all the way up to the variance of xn. We can do this because the covariances are 0. We assume that those random variables form independent draws. This now is equal to one over n squared times n times Sigma squared because each of these individual variances is sigma squared as we assumed before. So, in other words, we have average is an unbiased estimator, but it has a variance that is much smaller. It is smaller by a factor of n, then any individual random variable. The analogy is the following. If we train one single decision tree, we are in this setup here where we get an unbiased estimator for the concept that we want to learn. But a single decision tree is going to have a high variance. If we take many decision trees and in some sense average over them, we potentially get into this regime here where we still have an unbiased estimator. But an estimator with a much smaller variance.

Question is now, how can we average over multiple trees when we only have one data set from which we can only construct one single tree? The answer is found by the bagging algorithm. The idea is that we combine the concept of averaging with the concept of bootstrapping from statistics. Here's how we go about this. We take our training data points that we have and we repeatedly for each of the decision trees that we want to create, for example, 100 of them. We take our n data points and we sample n data points from these n data points with replacement. Meaning I

sample one of the data points, randomly, pull it back afterwards, sample the next data point, and so on. We do this for each decision tree and then we grow a complete tree from each of these sets of n samples. Let's have a look at how this would work in practice.

What we do is we predict the outcome which could be a category if we have a classification problem or a number if we have a regression problem of any new data point that we'll want to consider in each of these capital B many trees. Subsequently, we then take a majority vote. If we face a classification problem or an average, if we face a regression problem across all those trees. Let's again go back to our example. Imagine we would have grown 100 trees and we face a classification problem. It could be that for the new data point that we want to consider, 63 trees predicted that a new data point is of category 1. And 20 trees predicted category two. And the remaining 17 trees would predict category free. In that case, our bagging algorithm would go with the majority vote, meaning it would categorize the new sample that we have seen as being of category 1. Similarly, in a regression problem, imagine that we had 100 trees. The first one might have predicted the outcome 17.32. The second one could have predicted 15.27, and so on. The last one might have predicted 7.98. What we do is we take the average of all those 100 numbers. That is going to be our prediction from the bagging algorithm. The key idea here is that instead of going one single tree, we grow many different trees. And we afterwards average in some sense, our opinion over those trees.

**Video 3: Disadvantages of bagging [06:03]**

The idea of bagging is almost magical in some sense. We have taken one dataset, and we have created many different datasets from it using bootstrapping, and then afterwards, we have created different decision trees that are all based on the same data, but in some sense, we expect the variance of our overall prediction to have shrunk. This is not always necessarily the case, as the following example will show.

Let's go, again, back to our concept of the random variables. Consider again random variables x1 up to xn. But now, let's assume that they have mean mu, the variance of sigma squared, as well as a pairwise correlation coefficient of rho, where the correlation coefficient rho is defined as the covariance between any of those two random variables xi and xj, divided by the square root of the product of their variances. The correlation coefficient rho here essentially tells us how correlated, how closely related to different random variables are. And keep in mind, our decision trees that we get, since we essentially take the same data set and just change it marginally by sampling with replacement, these decision trees are going to be highly correlated. So, what happens if we take a random vote over those random variables when they are in fact highly correlated? So, in other words, what

we want to do is, first of all, we want to verify that if we average over those random variables, we are still getting an unbiased estimator for the mean that we are interested in, and surely that's the case, because these correlations here have no impact on the mean.

The question now, however, is, what is the variance of the average now that these random variables are no longer pairwise independent, but they are correlated? It turns out that, again, as we saw, we can write this as 1 over and n squared times the variance of the sum of these random variables. However, now the variance of these sums -- the sum of these random variables is not going to be equal to some variances anymore due to the presence of these correlations. In fact, the correct formula here is we need to sum up over all pairs *i* and j the core variances of *xi* and *xj*. And this can be expressed as the following. First of all, whenever *i* is equal to *j*, we get the variance of the random variable *xi*. However, whenever i is not equal to *j*, we get the covariance of xi and xj, and this covariance of *xi* and *xj*, by definition of our correlation coefficient, is going to be rho times the square root of the variance of xi times the variance of *xj*, which is nothing else than rho times sigma squared. So, in total, what do we get? We get 1 over n squared times n times this expression here, which is sigma squared. So, we get sigma squared over n. And now we get *i* is not equal to j in exactly n times *n* minus 1 cases. So, we get plus n times n minus 1 divided by n squared times rho times sigma squared. So, we get sigma squared divided by *n* plus n minus 1 divided by n times rho times sigma squared. If rho is 0, meaning the random variables are indeed independent of another, or uncorrelated from another, then this expression here would be completely removed, and we get back to the earlier result of averaging. Whereas, if rho tends towards 1, meaning that the different random variables are perfectly correlated, then we actually would end up -- if rho is equal to 1, we would actually end up with 1 over n plus n minus 1 over n times sigma squared. We would actually get to an estimator that is no better than a single random variable.

And this is one of the key difficulties with bagging. If we create many trees, as we do in bagging, but these trees are highly correlated, which they are going to be, because they are essentially based on the same training data sets, then the correlation between these trees is going to be high and averaging over these trees is going to help us somewhat but not as much as we were hoping for.

**Video 4: Random forests [02:21]**

We have seen that the classification and regression trees resulting from bagging are highly correlated. This is undesirable because that does not allow us to reduce the variance as much as we want to. So, our goal now is going to be to decorrelate the trees resulting from bagging. And this results in two different very popular classes of algorithms, namely so-called random forests and boosting algorithms.

Let's first discuss random forest -- the key idea in random forest is to decorrelate

the decision trees by forcing them to look at different subsets of predictors in each level. So, here's how we do that. We again consider a number of decision trees, let's say 100 of them, and for each of these 100 trees, we sample n of our training data points with replacement, just as we have done with bagging. However, now we again grow a complete tree just as with bagging, but in each splitting step, we only consider a random sample of the predictors. For example, the square root of the number of possible predictors. This is an interesting concept. We're clearly introducing bias here. Rather than splitting on the best predictor, we split on the best or the subset of these predictors. Clearly, that is going to give us decision trees, which individually are not going to be as good as they can be, but these decision trees look considerably different. We decorrelate these decision trees and then a similar technique, as in bagging, such as a majority vote or an averaged outcome, will typically give us better results. Again, this is a manifestation of the bias-variance trade-off. We are introducing a bias, but we hope that the introduction of this bias is far outweighed by the reduction of the variance that we achieve by decorrelating the trees.

**Video 5: Boosting [02:07]**

A different way to decorrelate the trees resulting from bagging is achieved by the boosting algorithm. The key idea here is to attach weights to each training sample. The higher the weight, the more important that training sample. Initially, we set the weights of all training data points to be the same. Then we grow our first decision tree, which is typically a very shallow decision tree, only to depth one or two. We use decision tree pre-pruning here. Once we have grown that tree, we look at the data points and decide which ones have been misclassified. Those that have been misclassified, I'll get a higher weight, whereas those that have been correctly classified get a lower weight. And then we grow the second decision tree. Again, a very shallow one, but now with a new set of weights that will put more emphasis on correctly classifying those samples that have previously been incorrectly classified. And we do that over and over again. Typically for very large number of trees, such as a 1000 or even more. Once we have grown all these trees, we can then again proceed just as a bagging algorithm. We take majority vote. If we have a classification problem, all we take average votes. If we have a regression problem. What is often done by the way is that the contribution of each tree and the boosting algorithm is it's out weighed by the misclassification rate of that tree. So, we give more importance to those trees, of the many thousands of trees that we have created that worked well then to those that did not perform well on the training data.

**Video 6: Summarising module ten [01:39]**

This brings us to the end of module ten, and the topic of decision trees. We have seen in this module three different ways to construct decision tree ensembles. We have discussed bagging, which combines bootstrapping with averaging in the case of regression trees, or majority votes in the case of classification trees. We then went over to random forests, which increase the viability of bagging by considering random subsets of splits in each stage of the tree construction. And we have seen boosting, when each tree we give a greater way to samples that the previous trees did poorly on, again, resulting in a greater variability than the standard bagging algorithm. Remember, tree ensembles reduce variance at a cost of a higher bias, since we no longer take optimal splits in each tree node. Oftentimes, this is desirable as a reduction in variance far outweighs the increase in bias. However, as a result, we also reduce the interpretability which is a major advantage of individual decision trees. Which criterion is more important, predictive performance or interpretability? Ultimately depends on the application area.