Module 14: Decision Trees

Video Transcripts

Video 1: Introduction

In this module, we're going to talk about a new classification algorithm called a decision tree. A decision tree is simply a tree of questions that must be answered in a sequence to yield a predicted classification. For example, here I show a classifier for animals. It starts by asking how many legs the animal has. If the answer is 0, we immediately say Snake. If the answer is 4, the model is not quite sure. So, we ask another question. Does it purr? If the answer is Yes, the animal is a Cat, and if the answer is No, the animal is a Dog. Now, the other possibility is that the number of legs was 2. In that case, as you see here, we have up to two additional questions we may need to ask before deciding if it is a Kangaroo, a Human, or a Parrot.

Now, obviously this decision tree classifier does not cover all animals. And in principle, with the right sequence of questions, we could build these absolutely gigantic decision trees that could differentiate a vast number of animals. For this module, we're going to focus on an iris flower dataset. This dataset consists of 150 flower measurements from three different species. These three species are Versicolor, Setosa, and Virginica. Now, each row of this table gives a single observation about a flower that was picked for the dataset. The observation includes which species the flower belongs to, as well as the sepal_length, sepal_width, petal_length, and petal_width of each flower, where the petal and the sepal are two different parts of the flower.

Here's a two-dimensional scatterplot of that dataset, showing only the petal length and the petal width. This plot ignores the other two dimensions, the sepal length and the sepal width, that are in the dataset. Now, as you've seen before, a logistic regression model trained on only these two features will have linear decision boundaries. The model seems to do a pretty good job and, with some higher-order combinations of these features, we could imagine getting even better accuracy without going into overfitting territory.

Now we could have, of course, also fit a k-nearest neighbors model and it would have its own decision boundaries. Though, I do not show such a figure here. For this module, we'll explore how a decision tree model behaves when attempting to fit this data.

Video 2: Building Decision Trees Visually

The plot seen here is just the same plot as before. We see the width and the length of the petals of each flower on a scatterplot. Let's consider how we could build a decision tree manually—just by looking at this picture. We might start by setting up a question whose answer is Yes only for the setosas. And that should be pretty easy as they're off in this corner by themselves. There are many possible rules that we could select. And the one that I have chosen arbitrarily is: Is the petal_width less than 0.75? And is the petal_length also less than 2? If the answer is Yes, then our decision tree is done. We know what kind of flower it is, it's a setosa.

So, on the scatterplot, I highlight here the region covered by this decision rule. Note that we could have selected many other possible rules. For example, is petal_length less than 2? That would also work. Or we could have asked simply, is the petal_width less than 0.75? So, my choice of rule

was entirely arbitrary and—like many others—it does a perfect job of identifying the setosas. So, from here we need to come up with another rule. There are an infinite number of possible rules, but the one that I have chosen arbitrarily is the simple question: Is the petal_width greater than or equal to 1.75? So, the Yes side represents everything with petal_width above 1.75 and the No side represents everything below. Now, since everything above this line is a virginica, if we answer Yes to this question, we are done. Or in other words, this space at the top of the figure is predicted to be entirely virginicas by our decision tree model. If the answer was No, by contrast, then we have more work to do.

The next totally arbitrary rule that I selected is: Is the petal_width less than or equal to 1.55, and the petal_length also less than or equal to 4.95? If you'd like, pause the video and think about the implications of a Yes or No answer to this question. This time, a Yes answer gives us a region that only includes versicolors. And so, a Yes answer would mean that we're done. Now, note that this space is not quite a rectangle as part of the space—at the bottom left of the figure—was already reserved by the decision tree rule at the very start that separated out the setosas.

Now by contrast, if the answer to this new question is No, then we find ourselves in the uncolored space that remains. We observe that the space looks like it includes two versicolors and five virginicas. And so, our remaining decision rules, they're going to attempt to differentiate between these remaining seven flowers. So, the next rule that I pick arbitrarily is a two-part question. Is the petal_width less than or equal to 1.65? Or is the petal_length greater than or equal to 4.95? Note that this time I'm using the

word 'or' instead of 'and.' And again, if you'd like, pause the video and try to predict the color and the shape of the resulting space.

Now here, because we use the word or, we end up with a space which is disjoint. There's this thin strip of colored space on the left and a larger block of colored space on the bottom right. This space includes only virginicas. So, we assign our virginica color, leaving only a small part of the original feature space unclassified. And in the unclassified space now, we see now there's only three flowers remaining: Two versicolors, and one virginica. If you'd like, try to come up with a rule that will differentiate the three remaining flowers. The rule I picked to differentiate is: Is the petal_length less than or equal to 5.55? Naturally, this rule splits the remaining space into two halves. The left, or Yes side, is colored for versicolor and the right, or No, side is colored for virginicas. So that, my friends, is an example of how we can build a decision tree by eye.

Here's a question to ponder. What is the training error for our model on this data set? And is that good or is that bad? Well, it seems to me—just by eyeballing it—that our model seems to get every single data point correct. And I would argue that this is in fact a bad thing. That suggests that our model is prone to overfitting. Our decision boundary is somewhat arbitrary and complicated. And it seems plausible that, if we gathered additional flower observations that we did not have access to when we trained the model, that we would make significant errors of the boundary of the versicolor and virginica regions that we might not incur with a little more caution. We will discuss more about overfitting in decision trees later. But first, let's see how we can build and visualize decision trees for classification using scikit-learn.

Video 3: Building Decision Trees in Scikit-Learn

Fitting a decision tree model in scikit-learn is exactly like fitting any other scikit-learn estimator. We create a DecisionTreeClassifier object. And then we call its fit function on our data. And of course, making predictions using such a decision tree model is exactly like before as well. We simply call the predict function on an observation or a DataFrame of observations, and we get back the predicted class for each observation. Earlier, I created a visualization of the individual rules for each decision tree. So, that crude visualization that I created, was manually put together using an illustration program. Now, I really like these decision tree diagrams, showing each of the rules. And luckily there's a way to automatically generate them.

The scikit-learn dot tree model provides a plot_tree function. And, to use it, we provide a decision_tree_model, a list of feature_names, a list of class_names. And, in my case, a couple of aesthetic arguments—rounded = True and filled = True—that are just some Boolean arguments. Now unfortunately, the default layout for the figure you get back is not particularly good. And if you look at the figure that was generated, many of the arrows are hard to see.

Now luckily, there's an even better library, called export_graphviz, which uses the AT&T Graphviz library to make a much nicer layout. And code for doing so is shown on the screen. Now, looking at this visualization, we see that each of the box have lots of annotations. For boxes which represent yes or no questions, the first thing in the box is the question. For example, at the root node, we have the question: Is the petal_length less than or equal to 2.45? Looking at all of the boxes, you'll see that all of the rules only ever

use one of the features. Unlike the decision tree that we manually created earlier—which had rules that had conditions like, is petal_length less than three and petal_width greater than two—scikit-learn decision trees only ever consider a single feature at a time. The next thing that we see in each box is the entropy, which we'll discuss in a later video.

Next, let's consider the line in each box that says samples. In the root, samples is equal to 150. This means that when the decision tree was trained, there were 150 total samples. In the node to the left of the root, we see there are only 50 samples left. These are the 50 samples for which the rule above was true. By contrast, the right child of the root node has the other 100 samples. These are the samples whose petal_length was not less than or equal to 2.45.

Below the samples line in each box is a list called value. This provides the number of samples in each class that remain after applying all of the rules above the current node. For example, for the node with 50 samples that we just considered, all 50 samples belong to the left class and zero samples belong to the middle and right classes. By contrast, in the 100-sample node that we just considered, zero samples are in the left class, 50 samples are in the middle class, and 50 samples are in the right class.

The last label in each node—class—represents the most likely class, given the knowledge that is available to us by the time we reach this node in the decision tree. For example, consider this node, which has 54 samples: 49 in the middle class and five in the right class. The most likely class is the middle class. After all, 49 out of 54 samples belong to this class. And what class is that? It is the versicolor class. Thus, this node is labeled versicolor.

Next, consider the color of each node. The color reflects the most likely class for each node. The darker the color, the more confident the model is in that classification. Note that two of the nodes have absolutely no color whatsoever. This is because there is no majority class. In other words, there is a tie.

Let's compare the decision tree boundaries of a scikit-learn logistic regression model with the decision tree model. We observe that unlike the logistic regression model boundaries, the decision tree boundaries are non-linear. Furthermore, we have observed that they seem to be getting 100% accuracy. In the next video, we will actually compute the accuracy of our decision tree model rather than just looking at it by eye.

Video 4: Measuring Our Decision Tree's Training Error

To assess the accuracy of our model, we can use the accuracy_score function that we saw in an earlier module. So, when we run this code to compare our decision tree model's predictions to the actual species in the dataset, we get only 99.3% accuracy. Now to understand why our tree does not have 100% accuracy as it seemed before, let's look carefully at the visualization of our decision tree's rules.

There's this one terminal decision point in the tree where, at the very end, there's more than one possible right answer. Pause the video and see if you can find that node. So, this node is very special. We see that it has three samples and they're not all from the same class. And we call such a node impure. So, for some reason it seems like scikit-learn just gave up and it didn't create a rule that can differentiate these samples. To understand why we can call .query, echoing each of our decision tree's rules in turn.

So, we say: Hey, DataFrame, give me all of your data points where the petal_length is greater than 2.45, and where the petal_width is greater than 1.75, and where the petal_length is less than 4.85. When we do this, we get back three data points. And if we look closely, we see that all three of these have the exact same petal_length and petal_width; 4.8 and 1.8 respectively. However, not all three of these flowers are of the same species. And so, given just these two features, it is impossible to differentiate these flowers.

We can generalize this idea with the following statement. Scikit-learn decision trees always have perfect accuracy on the training data, except when there are samples from different classes with the exact same features. So, if the versicolor in our table here had a petal_length of 4.8001, we would have 100% training accuracy. This tendency for these type of models to have perfect training accuracy should give us grave concern about overfitting.

Video 5: Decision Tree Overfitting

Here we see a scatterplot of the exact same Iris dataset. Only now I'm using the sepal_width and sepal_length instead of the petal_length and petal_width. Now in this space, the versicolors and virginicas are much more intermixed. So, if we train a decision tree model, it will do its best, earning 100% training accuracy—except when the data points overlap exactly.

Here, we see the resulting decision tree boundaries. Observe they are highly erratic. At least a few of the data points live in their own personal bubble. In other words, our classifier has effectively just memorized these specific

data points. Given the complexity of these decision tree boundaries, they are quite unlikely to reflect any sort of useful physical reality. Now if you look very carefully, you'll see at least a couple of points that are misclassified. That only occurs when two samples from different classes overlap exactly. Other than such cases, we will get 100% training set accuracy.

Another way to observe that our model seems to be overfitting, is to look at the decision tree diagram. Here, we see that the decision tree is much more complicated than the tree we saw when we used the petal_length and petal_width. This tree is 11 layers deep and quite wide. Zooming in a bit, if you squint, you might be able to tell that there are only a very small number of samples in some parts of this tree. And in effect, the model has memorized those data points.

We saw earlier that including a large number of features can lead to overfitting. Let's see what happens if we try to use all four of our petal and sepal measurements. Since the prediction space is four-dimensional, we cannot easily visualize our decision boundaries. However, by looking at the visualization of the decision tree rules, we can compare how our new four-dimensional model will work to our earlier models, which used only the petal_length and the petal_width or the sepal_length and the sepal_width.

So here, on the left, we see the decision tree diagram from before, which used only the petal attributes. And on the right, we see a new four-dimensional model, which uses all four attributes. You might be surprised to see that this four-dimensional model does not overfit, even though it has access to all of the features. Instead, these two models are extremely

similar. And in fact, our four-dimensional model only makes use of the sepal features exactly once. And, actually, that's to resolve that tricky case where there were these three overlapping virginica and versicolor flowers. This showcases that for decision tree models, more features doesn't necessarily lead to overfitting. Especially if a small subset of the features do a good job of resolving the difference between the classes.

So, which of these two models is better? Well, in the real world, I doubt if there's any perceptible difference. I'd say they're about the same. Now, going back to our model trained on only the two sepal features, this tree seems to be badly overfit. So, to understand how to prevent such overfitting, we should first understand how this arises. And to do so, we'll need to talk about the algorithm that sklearn uses to generate these decision trees.

Video 6: An Intuitive Look at Decision Tree Boundaries

The traditional decision tree generation algorithm is as follows. All of the data starts in the root node, and we repeat the following until every node is either pure or unsplittable. First we pick the best feature x and the best split value β. For example, x equals petal_length, β equals two. We then split the data into two nodes: One where x is less than β and one where x is greater than or equal to β. A pure node is defined as a node that has only samples from one class. And we define an unsplittable node as a node that has overlapping data points from different classes and thus cannot be split.

So far, we have not yet defined when we say best split. Let's explore the notion of a best split intuitively. When we start, all of the samples are in the root node—50 from each of our three classes. And there are many potential

splitting lines we could draw. For example, we could draw the line separating petal_width less than 1.5 and petal_width greater than or equal to 1.5. This yields on the Yes side, 0 setosas, 15 versicolors, and 49 virginicas. And the remaining 86 samples are on the No side. Note that if you try to count the number of flowers of each type from this figure, you won't arrive at those numbers because there are many overlapping data points that you can't see. For example, there are multiple versicolors with the exact same length and width as each other.

So, is this choice of split good? Well, it's clear that it's at least a little useful. For example, it sticks most of the virginica in a single class. Now we could also try a different choice. For example, this vertical line, separating points with petal_lengths greater than or equal to 4. Arguably, this line is better than our previous choice of line, and it feels intuitively better to me because all of the virginicas are in one node. But another split we might try, is whether the width is greater than or equal to 0.5 or not. This split seems intuitively even better. It keeps 48 of the setosas together and the remaining 102 samples are on the other side of the split. And of course, as we did at the very beginning, we can choose a horizontal or vertical line that totally separates the setosas, for example, width greater than or equal to 0.8. Intuitively, this choice seems like the best one.

So, we'd like an algorithm that can automatically conduct this process of considering different splits. But before we can do that, we need some sort of rigorous definition for a good split.

Video 7: Entropy

Let us introduce the notion of node entropy. This is going to feel really arbitrary and confusing at first but, after a few examples, it'll become clear. So first, we're going to define p_C as the proportion of data points in a node that have label C. For example, for the node at the top of the decision tree, p_0 is 34 divided by 100 or 0.31. p_1 is 36 divided by 100 or 0.33. And p_2, which is 40 over 110, is 0.36. So next, we'll define the entropy S of a node with this equation. S equals the negative of the sum of p_C times log base two of p_C summed over all classes C. So, for example, S for this top node is: −0.31 times the log_2, of 0.31 minus 0.33 times log_2 0.33 minus 0.36 times log_2 0.36.

Now, computing these logs and multiplying out these values, we get 0.52 plus 0.53 plus 0.53 equals 1.58. That's the entropy. Now this definition of entropy isn't hard to understand. But you might find it a little hard to follow in the middle of a lecture video. So, let's do a quick exercise to reinforce your understanding. Consider the node on the left, which has 31, 4, and 1 sample in each class respectively. I'd love for you to pause the video and compute p_0, p_1, and p_2. Once you have those values, then compute the entropy of the node using those p-values. This will take a couple of minutes. And if you're stuck, go back and see how I computed the entropy: S equals 1.58 for the top node.

So here we see the answers to this question, which hopefully you've paused and worked out. p_0 is 0.86, p_1 is 0.11, and p_2 is 0.028. Applying our formula for entropy to these three values as shown on the screen, we compute an entropy value of S equal to 0.68 for this node.

So, what is entropy? We can think of entropy as how unpredictable a node is. Low entropy means more predictable and high entropy means less predictable. So, for example, if we're working our way through a decision tree, at the root node shown we have very little idea what class the sample belongs to and thus the entropy is relatively high. By contrast, by the time we reach the left node, we know that it is very likely a setosa, and thus the entropy is lower. If we look back at the entropy of the nodes in our very first decision tree that we built on the petal_length and petal_width, we can see that as we work our way down the decision tree—learning more and more about the likelihood that our sample belongs to a given class—the entropy decreases. For nodes where all of the samples belong to the same class, observe that the entropy becomes 0.0.

Let's reflect on some of the numerical properties of entropy. A node where all data are part of the same class, has zero entropy, as we just saw. Because the log_2 of 1 is 0. A node where data are evenly split between two classes has entropy 1. This is because −0.5 times log_2 0.5, −0.5 times log_2 0.5 is 1. A node where data are evenly split between three classes, as they were at the beginning of our decision tree generation process, have entropy 1.58. And this is because 3 times −0.33 times log_2 0.33 is 1.58. And more generally, a node where data are evenly split into C classes, has an entropy of log_2 of C, which we can show by just continuing the trend we observe for these evenly split data for the two and three class cases.

Now at this point, you may wonder why we're learning about this mysterious numerical measure of uncertainty. And we'll see why in the next video.

Video 8: Using Entropy to Select Decision Trees

We saw earlier that in any given point in the decision tree generation algorithm, we have choices we have to make. For example, do we want to split based on the question: Is petal_length greater than or equal to 4? Or do we want to use something like: Is sepal_length greater than or equal to 0.5? Or something else? How do we decide which splitting rule is better? Scikit-learn uses a concept known as the weighted entropy as part of its decision-making process.

The weighted entropy of a node is simply its entropy times the fraction of the samples that are in that node. And we will use the symbol WS to represent the weighted entropy. As an example, the entropy—S—of the node here with 54 samples is 0.445. However, the weighted Entropy—WS—of the node with 54 samples is 54 over 150 times 0.445 or 0.16. As another example, the weighted entropy—WS—of this terminal node with three samples, is three over 150 times the entropy 0.918, which is 0.018. That's the weighted entropy. Note that the weighted entropy of the root is just the entropy. In other words, WS and S are the same for the root since the root contains all of the nodes: 1.58 times 150 divided by 150.

So, here we've annotated every node with its weighted entropy, WS. And observe that as we move down the tree, the entropy always decreases. Equivalently, uncertainty about the class for our samples decreases as we ask further questions. Each decision tree rule reduces the weighted entropy by a certain amount. So, for example, the first rule—petal_length less than or equal to 2.45—reduces the weighted entropy from 1.58 at the root, to 0 in the left child, plus 0.67 in the right child. So, the total reduction in entropy is

therefore ΔWS is equal to 1.58 minus 0.67 or 0.91. Now here when I say ΔWS, I'm using the Greek letter Δ to represent the reduction in entropy. And in a decision tree model, ΔWS is always positive. In other words, the entropy always decreases as we move down the tree because we're learning more.

So, as another example, consider the rule petal_width less than or equal to 1.75 from our original decision tree. Since this rule is a child of the root, the entropy of this node before we even asked this new question, is much lower than we started at the root—because it's already another question before it. So, this new petal_width less than or equal to 1.75 question will reduce the entropy even further. Specifically, after we ask that question, is petal_width less than or equal to 1.75? We either end up in a node whose weighted entropy is 0.16, or whose weighted entropy is 0.046. The sum of these two children is 0.206. And so, the total difference in entropy before and after this question is 0.67 minus 0.206, which gives us 0.46. Or more compactly, we simply say the ΔWS is 0.46 for this rule. That captures how much we learn from that rule.

Now intuitively, as we build the decision tree like this from scratch, we want to select the decision trees rules that yield the largest reduction in entropy. In other words, which have the largest WS values, ΔWS values. And that's exactly what sklearn does. Let's use this idea to evaluate the quality of the four possible splits that we discussed earlier. Our first possible split was: Is the petal_width greater than or equal to 1.5? So, since the left child node has 50 samples in class zero, 35 samples in class one, and 1 sample in class two, the resulting entropy of that node is 1.06—using the same formula we saw in a previous video.

Scaling by the number of samples, we get a weighted entropy of WS equal to 0.61 for this possible left child. We then compute the entropy of this possible right child and we get 0.79. And if we use the weighted entropy formula, we get 0.344. So, computing ΔWS for this rule, we get 1.58 at the root minus 0.61 minus 0.34 or 0.63. So, that's the quality of this possible split. In other words, it reduces entropy by 0.63.

Now the next split that we considered is, is the petal_length greater than or equal to four? So, how good is that rule? Well, here the potential left child has entropy 0.68, which yields a weighted entropy of 0.28. And the right child has entropy 0.99, which yields a weighted entropy of 0.59. Now, I'm not going to read off all the rest of the math here to avoid being boring. But the ultimate result is that this rule, it has a ΔWS of 0.71. In other words, the entropy is reduced by 0.71. That's better than the rule before, we're learning more with this question.

The third split that we considered was, is the petal_width greater than or equal to 0.5? So, now the potential left child for this rule has entropy 1.12 with weighted entropy 0.76. And the right child has entropy 0 since all the samples belong to one class. Working out the math for this choice of split, the ΔWS is 0.82, which is even better than our second split choice. So, we learn even more from this question. There's less uncertainty.

The fourth and final split under consideration earlier was, is the petal_width greater than or equal to 0.8? Here, the potential left child has entropy 1 and the right child has entropy 0 since, again, all samples belong to one class. So, the weighted entropy of the left child is 0.67, since it contains 100 out of 150 samples and 100 over 150 times one is two thirds. This results in a

ΔWS value of 1.58 at the root minus 0.68, so, WS is 0.91. This is the highest ΔWS of any split we tried, and it is in fact the highest possible ΔS for a single rule. So, if faced with just these four choices, our greedy decision tree algorithm will select this rule. It's the best—it's the rule where we learn the most.

So, returning to our description of the traditional decision tree generation algorithm, we've now resolved the one remaining mystery. Namely, how do we decide that one split is better than another? And the answer we just saw is that we'll pick the split with the largest ΔWS. And we still have to decide which parameters and which values to consider. For small models, it's…it's actually feasible to iterate over all possibilities. But if there's too many, we can choose randomly. We won't go through the exact details of these various approaches. Though, you'll get a chance to think about them a little bit on this module's homework.

Now, note that it doesn't really matter if we pick rules based on greater than or equal to or less than or equal to. In fact, whereas my algorithm uses greater than or equal, sklearn uses less than or equal. But it's basically the same thing. For your later perusing, I have included here an image of the entire decision tree generated by scikit-learn for the petal_length and petal_width. So, now that we understand how decision trees are generated, let's finally turn our attention to avoiding overfitting.

Video 9: Restricting Decision Tree Complexity

Let's now return to the question of overfitting. As we saw earlier, a decision tree that's allowed to grow to its full extent, it runs the risk of overfitting. So, how do we control this growth? Well, with logistic and linear regression, we

had weights that were parameters: theta one, theta two, theta three. And we used something like L1 or L2 regularization. But with decision trees, we can't do that. There's no weights. However, there are a number of natural ideas that we could apply to keep a decision tree under control.

So, the first category of ideas will be to disallow our decision trees from growing all the way. So, how would I do that? Well, some examples include, for example, don't split a node if it has fewer than 1% of the samples. And in sklearn, there's a hyperparameter to set up this rule called min_sample_split. Another, a natural approach is, let's not let any node be more than seven levels deep in the tree. In other words, we'll never ask more than seven questions before coming to a final decision about our class. This hyperparameter is called max_depth in sklearn. Third approach is, don't create a split if its ΔWS is less than 0.01. In other words, a split needs to have some meaningful impact on entropy to be worth asking. And in sklearn, this can be set using the hyperparameter min_impurity_decrease.

Because there are so many natural ways to control a decision tree's growth, decision tree models in sklearn have a large number of hyperparameters. Now typically, you don't set them all explicitly as there is a default option. For example, the default option for min_sample_split is two. And because there are so many hyperparameters doing the full grid search of all combinations can be very computationally expensive. And you'll have a chance to play around with this some more on this module's homework.

Now incidentally, one of the reasons I explained to you exactly how sklearn builds decision trees, is so that you understand what the hyperparameters mean, instead of them just being some kind of meaningless values. If I

hadn't taught you about the idea of ΔWS, you'd have no idea what scikit-learn's min_impurity_decrease hyperparameter even means. Now, note that I haven't covered all of the hyperparameters in sklearn. For example, there's one called ccp_alpha. See the scikit-learn documentation for more details on such hyperparameters.

Now there's a totally separate category of ideas where we let the decision tree fully grow, but then later we cut off some of the less useful branches of the tree. For example, consider the highlighted branch of the tree shown. There are many rules here, but relatively few samples to differentiate: Only 8 from one class and 32 of another. There are many ways to try to prune out such pieces of trees. One particularly interesting way, is to use a validation set to see if keeping a given branch of the tree is worth it. Specifically, we run our model on the validation set twice: Once keeping the branch and the second time replacing the branch by its most common prediction. In this case, virginica. If there's no or minimal impact on the validation error, then we delete the split. The idea here is that on unseen data, the split was not useful.

Recall that when a node has multiple samples in it from different classes, the predict method simply returns the most common class. There's also a predict_proba method, which returns the fraction of samples that belong to that class. Now there are tremendous number of ways that have been proposed and explored for controlling the growth of decision trees. The bottom line is, you don't need to know about all of them, but knowing about some of them will help you in the future if you're building decision trees.

I should note there's a different, totally distinct approach to avoiding overfitting with decision trees called a random forest. They're very popular, but we're not going to talk about them right now. We'll talk about those in a later module.

Video 10: Conclusion

In this module, we saw a completely new technique for classification called a decision tree. The fundamental idea with this approach is that we simply asked these yes or no questions about the size of various features and come up with an answer. So, in scikit-learn, each of those questions involves only a single feature, though other possibilities exist as we saw early in the module. Now, since all of our predictions are the result of a sequence of simple yes or no questions, decision trees are very easy to interpret and even manually execute with no arithmetic of any kind required. In other words, you could train even a relatively young child to trace through and generate predictions with, say, a paper copy of a decision tree. In some sense, it's the easiest possible classifier to understand. You'll have a chance to explore this idea of model interpretability on the homework.

Now at the same time, decision trees can fit arbitrarily complex data with 100% accuracy. And, in fact, for datasets where there's no overlapping data points from different classes, a decision tree that is allowed to grow to its full size will always have 100% training set accuracy. Now that tendency towards having a 100% accuracy, means we need to exercise caution to avoid overfitting. Keeping complexity of a decision tree under control is generally not, in my opinion, as mathematically elegant as we saw in other domains like linear models where we had the idea of regularization, which was mathematically beautiful. But with decision trees, we did have a

number of approaches that were pretty natural, including hard constraints to prevent tree overgrowth and pruning rules to trim branches of trees after they're grown.

Then in a later module, we'll also discuss a really interesting idea known as a random forest, which is a commonly-used technique in real-world machine learning that involves decision trees. So, that wraps up our discussion today of decision trees, and I will see you next time.

Video 11: Evaluating Predictive Performance Compared to Humans

The models and data science tools you're learning open up numerous ways to explore unseen relationships in data and gain insights from them. A key aspect of many of the models, notably supervised learning, is the focus on prediction. The ability to use data to make a better prediction is at the heart of the current AI revolution, as well as a lot of enthusiasm for where and how to develop these tools that are new products, companies, and solutions.

Today, I want to talk about an example of a key issue facing any developer of a new prediction tool: How to evaluate whether or not the model is better than the status quo? By that, I don't mean evaluation using cross-validation and fit metrics—an important statistical exercise for model selection—but comparing the model to what is happening in the real world today. For example, many models are developed to try to outperform humans at something they do.

Think about robo-advisors trying to help clients invest instead of a stock broker or financial advisor. Consider predictive models to recommend sales leads for salespeople rather than relying on their network or intuition. I want to focus on one of these issues today. Can an algorithm do a better job than a judge at deciding whether to release a prisoner? We'll come back to the broader ethical and policy issues because they cannot be underestimated. However, let's start with thinking about why this problem might be amenable to machine learning and how we can determine if in fact the algorithm offers value.

We're going to focus on a specific setting that was studied by Jon Kleinberg and his co-authors in a paper in *The Quarterly Journal of Economics*. They study a question facing judges: Whether to offer bail to those accused of a crime? The job of a judge in this case, is to decide whether an individual is going to commit a crime while on bail awaiting their court date—including the crime of jumping bail and not showing up. Take a moment and think about whether this particular problem strikes you as amenable to machine learning or not. If so, what kind of data would you want and how would you build a model? If not, why doesn't it fit? Go ahead and pause the video for a moment and consider these issues.

The answer they give is that this is a good problem for machine learning because it's a pure prediction problem. That's an important distinction for many things judges do—where they are required to weigh evidence, and consider the facts, and the context of the law, and ethical principles. Here, though, we simply want judges to jail people who are likely to commit crimes and let those who are unlikely go. That is a prediction problem. The authors gather data that would be available to judges at the time of the

decision to build a prediction model in New York City for a five-year period starting in 2008. These include prior offenses, current charge, the location of the offense, and the age of the defendant.

It's really important to note here that they limit the data to what would have been available at the time of a decision. It's common to have rich data on decisions in prediction problems. But some of it might have been generated after the decision of interest was made. For example, consider a prediction problem of trying to classify risk for emergency room patients. Often in the rush, doctors will come back to update a patient's chart with more detail. If we use those data, the algorithm is, in effect, using information that would not have been available if it were deployed to try to triage patients at the time of arrival.

They then build a prediction model, a boosted tree model to be precise, that predicts crime, probability of a crime based on observables. The results are quite interesting. It is clear the prediction model fits the actual observed crime rate in a held-out sample well. It does not, however, fit the patterns observed in the judge's bail decisions well. This suggests room for improvement. Where the lack of fit occurs offers further insight. The judges are relatively good at making decisions amongst lower-risk defendants. As risk increases, the jailing rate increases among those with a low expected probability. However, the jail rate essentially plateaus for the judges, while the model and the reality of crime risk continues to rise. The opportunity for improvement is among those at the highest risk of committing a crime, notably those with a prior felony.

But how can we actually determine whether we can be more efficient with an algorithm? A big problem facing us here, and in many cases where we want to compare an algorithm to existing human actions or other actions, is that we don't see what would have happened. That is, if a judge denies bail, we don't know whether a person would have committed a crime. Thus, it is hard to say whether an algorithm would perform well. They suggest a really powerful approach to overcoming this issue in evaluating the algorithm's predictive performance and efficiency. They rely on the fact that defendants are randomized to judges, and judges have widely varying levels of jailing. This kind of situation is not uncommon. Think of a call center where callers are randomly allocated to whoever's available. Consider a sales team where leads are simply given at random.

They exploit this randomization to focus on the most lenient judges. For those judges, we can actually know what the algorithm would do because they release a lot of their defendants. We can then compare the crime and incarceration rate for the algorithm to the crime and incarceration rate for less lenient judges. To think about this simply, suppose there was a judge who simply released everyone. We can take the characteristics of all of that judge's defendants and ask what their risk is in the algorithm. We start with the highest-risk defendant and proceed. Since they were all released, we know how much crime was committed that would not have been, had they been denied bail. Their approach doesn't exactly do this, but it's close. Instead, they divide judges into five quintiles of bail rates and consider how the most lenient 20% compare to the other four groups.

The results are striking. First, consider the most lenient versus the next most lenient. The second quintile, judges reduce crime by 10% by jailing 7%

more people. This is the level of efficiency we want to compare the algorithmic approach to. What they show is that if we instead use an algorithm, we could jail far fewer people to achieve the same reduction in crime. To get a 10% reduction in crime using the algorithm to assign bail, you'd only need to jail 2% more people than the lowest quintile group. Alternatively, if we jailed 7% additional people according to the algorithm, instead of the way judges do it, you could reduce crime by 20%. That's twice what the judges achieved.

If you're interested, the paper goes into much more detail on alternative scenarios. However, the punchline is striking. Because judges are being asked to solve a prediction problem machine learning could potentially do, there's an opportunity for efficiency. That is, we can either achieve a lot less crime for the same incarceration rate, or we could keep far fewer people in jail and not increase crime. Either way, there's an important opportunity.

The study also offers a really nice lesson in how to take the models you develop to the next level and actually consider how well they might perform in practice. Finally, I encourage you to think about whether efficiency is the only criteria in deciding whether an algorithm should be used in bail decisions. What other ethical and policy implications are at play? How might you trade these off in making a decision? These kinds of issues also face data science teams and those developing data products and policies today.