COLUMBIA | ENGINEERING
EXECUTIVE EDUCATION

Week 6
Video Transcripts

Video 1 (9:16): Maximum Margin Classifiers

And so, what are we trying to achieve with the maximum margin idea? So, remember with the perceptron, that if we have two classes, that are linearly separable, meaning that, we can find a line or a hyperplane that perfectly separates the two classes, so that all the data points in one class are on one side of the hyperplane, and all the data from the other class are on the other side of the hyperplane, the perceptron will find that hyperplane. But running the algorithm, the perceptron algorithm, it will find, the first hyperplane, that it comes across and then terminate.
So it just finds, a, hyperplane that separates the two classes. But it doesn't really give any sort of weight, to one hyperplane over the other. Any separating hyperplane, is equally good, as far as, the perceptron is concerned. So the idea of maximum margin, is that, not all hyperplanes are equally good. For example, if you look at this hyperplane. Even though, it separates the two classes. They're awfully close to the hyperplane. The closest points are very close to the hyperplane. And so, we could imagine that, if we sampled more data from the red class and from the blue class, we would start to have some misclassified data points, given this hyperplane. So the maximum margin concept is that, if we want to generalize well to new data, if we want to sample more points from the blue and the red class here, and have it still be a good classifier, we would, we should, put the hyperplane, loosely speaking, right in the middle of the data. Mathematically, what that means is that, we would want to maximize the margin to the hyperplane. What the margin is, is quite simply, it's the distance to the hyperplane of the closest point in the data set. So we take, the distance from every point, in the data set to the hyperplane. For example, this distance, we calculate it for each point. And then, we pick the smallest distance, that's the margin. So we want to maximize the margin, meaning that, we want to find the hyperplane that, separates the two classes. But is, as far away as possible from the data... the closest data points, in each of those classes. So when we say, we want to minimalize the generalization error, we are making a, some sort of an assumption on how the data is generated.
For example, we just discussed, with the perceptron that we don't expect this hyperplane to generalize well to new data, because we expect to see some red class points down, here. And some blue points up here, on the wrong side of the hyperplane, if we sampled more data in the future. So with some approaches, like for example, if we assume some distribution on the two classes. What I'm showing here, is a base classifier, that uses a class specific Multivariate Gaussian, with a class specific mean and a class specific covariance. So, we discussed this previously. We saw there, that we would get a decision boundary that is, is curved like this. And this is due to the fact, that the two classes in this case have different covariances. If we forced the two classes to have the same covariance, we would get a line with different covariances, we'd get a curve. And in that case, what the base classifier is trying to do, is find the decision boundary that cuts into the probabilities, in such a way that the probability of making

an error under this distribution assumption is minimized. So, if we have distributions that are we think are reasonable for the data.
We can come up with these base classifiers that will minimize the probability of an error. Max margin classifiers, don't make any sort of distribution assumption on the data. So with the probabilistic assumption, we get these sorts of base classifiers. If we don't want to make any sort of probability assumptions at all. So, for example, we don't want to define distributions on the two classes. Then the idea of max margin is that, simply finding the hyperplane, that maximizes the distance to the nearest points is the best that we can do. So that's the motivation of max margin classifiers. So, in this lecture, I want to build up some of the intuitions and the mathematical ideas behind max margin classifiers. And the first thing to talk about is convex sets. So let's notice in the context of this data, that we've been looking at on the precious two slides, that, when we want to find a separating hyperplane, between these two classes, the placement of that hyperplane only depends on the outer points of the sets. What that means, is that, these points here that are inside the... the, the boundary of the data aren't in any way going to determine, what the hyperplane should look like, that maximizes the margin. Geometrically, and mathematically, what that means is that, we can represent each of the two classes by, the convex hull and what a convex hull is, is the smallest convex set that can be drawn around the data set.
So, in the next slide, I'll be more specific, about what it means. Intuitively, you can kind of imagine that, it's the smallest shrink-wrapped set around our data set, such that, the line of sight is maintained, between any two points within the set. So if this is, the convex hull of the red class. Then any two points within this hull can see each other. There's a line that can be drawn between them. And it will, still remain within the set. Mathematically, what it means is that, the convex hull of a data set is, all points that can be reached by taking weighted averages of the data. So let's, let this data set in this red class be 'x_1' to 'x_n'. These are two...these are 'n', two-dimensional points. Then every point, 'x_0' in this shaded region. So every point for example this point, call that, 'x_0'. So it's within the convex hull. It can be reached, by setting some probability distribution on the data. So let 'Alpha$_1$' to 'Alpha$_n$' be a probability vector. So all of the points are greater than, or equal to zero, in this vector. And, they sum to one. Then 'x_0' can be represented as a weighted combination, of points in the data set.
So every point that can be reached, by taking a weighted combination of all of these data points, form the convex hull, this shaded region. Any point outside of this convex hull, cannot be reached by taking a weighted average of all the points in the data set. So that's how we can define mathematically, the convex hull of a data set. It's every single point that we can get by weighted, by taking a weighted average of the data. All right! So, let's define the margin, we've already set it. Let's define it, one more time. The margin of a classifying hyperplane 'H', so that would be this hyperplane here, is simply the shortest distance between the hyperplane and any point in either data set, or equivalently, and this will be important when we analyze what the SVM is doing later. Equivalently, between the hyperplane and the convex hull of either of the two classes. So what we want to do with this hyperplane, 'H', is put it exactly in the middle of the two convex hulls. But of course, now the difficult part is how do we actually find this? It's easy to say that, we want to do that. And it makes intuitive sense why we would want to do that. But now we need some sort of an algorithm that's going to actually find this hyperplane that cuts right in the middle between these two data sets.

Video 2 (9:54): Support Vector Machines

So, here is the objective function that we're going to work with. This objective corresponds to what we would call a, Support Vector Machine. Intuitively, we'll make more sense of the name later, and we'll also show why this objective function should help us reach our goal of finding the max margin hyperplane. But, for now, let's just look at the optimization program that we're trying to solve. We're going to assume that we have, 'n' linearly separable points, labeled data pairs 'x_i' 'y_i' where, 'x_i' is in R^d and 'y_i' is a label...a binary label, it's either, plus one or minus one. So, we're doing binary classification here. And, what we want to do, is we want to minimize over 'w'—both the regression coefficient vector 'w'. That's our classifier, that defines the angle of the hyperplane. And 'w_0', which defines the shift of the hyperplane of the squared magnitude of 'w' divided by two, so this is just usually done for mathematical convenience.
Subject to, and here's the important part, these constraints. Subject to the product of 'y_i' with 'x_i', transposed 'w' plus 'w_0' is greater than, or equal to one. So, what does this mean? Remember, with a linear classifier, that we take the sign of this function, to be the predicted label of observation 'x_i'. And so, if the sign of this thing is positive, then we predict that it's plus one, if it's negative, we predict that, the label is minus one. And we want to get the label right. 'y_i', is the ground truth. So, this product of the label, the sign, either minus one or plus one, times the sign of this function, will always be positive, if we predict correctly. So, if it's a negative labelled point, and we predict it to be negative, by this thing having a negative sign then, their product will be positive. And, similarly, if the label is positive, plus one, and we predict it, to be plus one because this is positive, then, again, it's, the product is positive. So, the sign of this thing is going to be positive, if our linear classifier makes a correct prediction. And for convenience we're going to, for mathematical reasons, we're just going to say, that this is greater than, or equal to one, instead of zero.
Notice that, if we have a hyperplane that exactly separates two classes with no points on the boundary, we can simply scale 'w', so that, this is always true. So, it isn't clear from this objective function, why finding the minimum 'L_2' vector 'w', such that we correctly classify all of our data points, returns the max margin hyperplane. Obviously, if we find a vector 'w', that satisfies this, we will find a separable—hyperplane that's separates our two classes because of this constraint, satisfying, this constraint means that we've found a classifier, that can classify perfectly our data set. But why finding a vector 'w', that has the minimum 'L_2', norm subject to this constraint. Why does that one perfectly...why does that one correspond to the max margin classifier? So, that's what we're going to try to develop the intuition on, in the subsequent part of this lecture. So, why don't we first skip to the end, develop some intuition, and then come back, and look at more detail...in more detail, what the objective function, that we discussed on the previous slide is doing? So, can we intuitively say, what the max margin hyperplane should look like? And so right now this is mostly intuition. We're not going to give a rigorous proof, that the max margin hyperplane should satisfy this requirement. But to facilitate the intuitions, what should the max

margin hyperplane look like? So, intuitively, what we can think of as going on, is, we can take the data in both of the classes and construct their respective convex hull.
So, all of the data in class one will define a convex hull like this. So, this is the same data set that we've been looking at on the previous slides. And all of the data in class minus one, would define this convex hull. Then, what we would want to search for is, we would want to find the closest line that connects the two convex hulls. So, think of the two convex hulls being connected by, a rope. For example, we could connect this point on the convex hull, with this point on the convex hull, like this. And that rope has a certain distance to it. If we pull it tightly, there's a certain length of this rope that's connecting these two convex hulls. And now we want to reduce the length of that rope, until we can't reduce it anymore. So, we want to keep pulling these two points, this rope together, until we get to this point here, and this point here. At which point, we can't connect these two convex hulls with a shorter length rope. So, that's what it means to say the two points within the respective convex hulls that are closest together. Then, we would want to define the hyperplane that is perpendicular, to this rope and is exactly in the middle of the rope. So, the rope is defining, the closest points between the convex hulls, and then, that lets us define the max margin hyperplane, to be the hyperplane that's perpendicular to the rope, and exactly in the middle.
Okay, so if we take for granted that, that's satisfying that constraint, defines the max margin hyperplane for us. Then, mathematically, what we can say, is, first, that all we need to know is the left and the right point, the two points on the convex hulls. Given those two points, we have everything that we need to be able to define the max margin hyperplane. So, how can we find these two points? Mathematically, what we're saying is if we let the set, 'S_1', contain all of the data in class plus one and the set, 'S_1' contain all of the data in class minus one, what we're looking for, is two probability vectors, 'Alpha$_1$' that is the length, equal to the number of points in 'S_1'; and 'Alpha$_0$', that's the length equal to the number of points in 'S_0'. Such that we minimize the distance, between these two vectors. Notice that, this vector, by the definition of the convex hull, is a point in the convex hull of 'S_1'. We're taking all of the data in the plus one class, and then averaging it, in some way to get a point in the convex hull of the first class. And this is a point in the convex hull of the minus one class. We take all of the data in the minus one class, we average it, according to the vector 'Alpha$_0$' that gives us some point in this convex hull.
This point, some point in here, corresponds to this function. Some point in here, or along the boundary corresponds to this function, and, therefore, the distance, the 'l_2' distance, between those two points, is the distance between some point in the convex hull of class one and some point in the convex hull of class minus one. By now, trying to minimize this thing, we're trying to find the points in the respective convex hulls that are the closest together. So, in a sense, in order to find the max margin hyperplane, all we need to do is find a vector 'Alpha$_1$' and 'Alpha$_0$' that minimizes this function. Okay, so that's kind of, I went to the end first because that's kind of what, we're looking out for. When we manipulate this SVM objective function, and keep, you know, working with it, we want to ultimately end up at a place where we realize that what we're ultimately trying to find, what we're ultimately trying to do, by optimizing that function is find a vector, 'Alpha$_1$' and 'Alpha$_0$' that minimizes this.

Video 3 (10:54): Primal and Dual Problems I

Okay. So we've set up now an objective function, that incorporates our original constraints, by introducing Lagrange Multipliers. We want to minimize, that function 'L', over 'w', and maximize it over Alpha, and you notice, that we've written the original objective function now in this new way, by multiplying through Alpha. So what we do, and this is following exactly the same techniques the last time we saw it, it's just with a more complex optimization problem, is we first minimize over 'w' for a particular setting of these 'Alpha$_i$''s, for a particular 'Alpha$_i$''s, minimize over 'w' and minimize over 'w$_0$', and then, plug those solutions back into the original objective. So if we want to minimize this thing over 'w', the vector 'w', we take its gradients with respect to the vector 'w', we get that the gradient of this thing is equal to, 'w' minus this term here. And now we set that to zero, and solve for 'w'. So we find that, for a particular setting of the Lagrange Multipliers, 'Alpha$_1$' through 'Alpha$_n$', the vector 'w', that minimizes this thing for that particular set of Lagrange Multipliers, is equal to this.
Similarly, we take the derivative with respect to 'w$_0$'. Notice that, actually 'w$_0$', is now eliminated, but we set that to zero, and we get this additional constraint, that the sum of 'Alpha$_i$' times 'y$_i$', has got to equal zero. Now, if we look at what this is saying, 'y*i*' is plus one or minus one depending on which class the data point, the 'i' data point is in. So what we do, and these 'Alpha$_i$''s are all positive, so what this is saying, is sum up all of the Alpha in class one and, then separately sum up all of the Alphas corresponding to the data in class minus one, and then take the difference, of the sum of Alpha, over class one, minus the sum of the Alphas in class minus one. That has to be equal to zero. So the sum of the Alphas in class one, has to equal the sum of the Alphas corresponding to data in class zero. That's exactly what this is saying. Similarly, if you recall the perceptron, and you recall logistic regression you notice that, again, we're now able to express our vector, our hyperplane, defined by the vector 'w' as a linear, as a sum of the label, times the data point, times some weight. Remember for the perceptron, 'w' was equal to a sum over 'i' in some misclassified set of 'y$_i$', 'x$_i$'. We showed how with the soft 'k' and 'n' classifier, we introduced a kernel, that weights the data points. With logistic regression, there was a term out in front here, that corresponded to the probability of getting it wrong. Here now these Alphas serve as the weights for the SVM, but in a sense, we're still constructing our hyperplane, as just the sum of the data points premultiplied, by which class they are in, plus one or minus one. The only difference really is going to be how we learn these values for 'Alpha$_i$'. Okay, so now we've taken our primal problem, which is minimizing, over some parameters, we've constructed the Lagrangian, which takes into consideration the constraints, of the primal problem, adds Lagrange Multipliers like this. And then, we minimized the Lagrangian, over the original parameters to get the dual problems.
So when we minimize over 'w' and plug back in, we obtain the dual problem, which is equal to this. So, you can verify for yourself, that if we take the value, the vector 'w' and represent it this way, and we plug it back in here for 'w' and here as well, and then if, just as important, if we realize—if we multiply this through against 'w$_0$', we have 'w$_0$' times the sum over 'i', of 'Alpha$_i$', 'y$_i$', and the constraint that, that sum has to equal zero means, that we have 'w$_0$', whatever the value for 'w$_0$' is, times zero. So that's where, 'w$_0$', goes in this way. So 'w' and 'w$_0$' are both eliminated. Now we obtain this function. So I plug back in, for 'w'. I realized that, 'w$_0$' is multiplied by zero and I reshuffled some terms around and now, I

have a function of the Lagrange Multipliers only Alpha. So here, this term goes like this, and plugging in the value for 'w' here, and kind of simplifying this, gives me, this term here. Subject to the additional constraints, now that the sum of 'Alpha$_i$', 'y$_i$', has to equal zero, plus the constraint, that, these 'Alpha$_i$''s all have to be positive or equal to zero. So now I have this dual problem, something that I can maximize over Alpha. Maximizing that thing over Alpha is still a non-trivial problem. There are many software packages out there, that will do it for you. We're not going to discuss the algorithm for maximizing this thing over Alpha. In fact, it's not even obvious, that this is any easier, or more desirable than minimizing, over the primal problem, but we're going to take it at face value with the additional thing to point out here, which will become relevant later, that all of this, all that we are required to use, in order to solve this dual problem, are the dot products, between our data points.
And so when we see these dot products here we immediately think that we can now replace that with a kernel, but that's looking ahead a few slides. Okay. So we've run our algorithm and we've found 'Alpha$_1$' through 'Alpha$_n$', how do we now predict for new data? So remember, what we want is we have a new 'x$_0$', we want to predict, 'y$_0$' to be equal to the sine of this function, our linear classifier. That means that we need to find, 'w' and 'w$_0$', but what we have are, 'Alpha$_1$' through 'Alpha$_n$'. So we've actually got everything that we need. Remember when we solved, when we minimized the Lagrangian over 'w', we found that the optimal 'w' was equal to this function, of the data and the Lagrange Multiplers Alpha. So we've learned Alpha, or we've assumed we learned it with some algorithm—either our own implementation or one of the many good implementations of it— so we–what we do then, is we just simply plug in these Alphas, and take this sum and that's how we can get our linear classifier. Or that's how we can find 'w'. For 'w$_0$', what we need to do, is we need to go back to the Lagrange representation Lagrange representation of our problem. And notice what does the optimal solution look like? What is going to happen when we find the optimal 'Alpha$_1$' through 'Alpha$_n$'? So the thing to notice here, is that assuming that we've found a 'w' that perfectly classifies our data, that satisfies this, specifically this requirement, that this function is greater than or equal to one—remember that was our original constraint that we considered? Assuming that we've found that vector 'w', we know that this term here can either equal zero or it can be greater than zero.
It can't be a negative number. Similarly, this Alpha here can be equal to zero or it has to be greater than zero. It can't be a negative number. So we've found—notice that– if this is greater than zero, then any non-zero value for Alpha that's positive will make this objective lower. Remember, we want to maximize this thing over Alpha and so, if this term here for a particular, 'i' is non-zero, or if it's positive at the solution, then this 'Alpha$_i$' has to equal zero. Right! Similarly, if this 'Alpha$_i$' is not equal to zero, if it's positive, that means, that this term has to be equal to zero. So these two terms are kind of conflicting with each other. They can't both be positive. If one of them is positive, the other is zero. And that's simply by looking at this objective function and realizing that we're trying to minimize over 'w', and maximize over Alpha. That will tell us that, at the solution, they can't both be positive. So what that means is that we simply, in order to find the value for 'w$_0$', we find a particular 'i' for which Alpha, is strictly greater than zero. So we've run our algorithm, gotten 'Alpha$_1$' through 'Alpha$_n$'. Some of them—a lot of them will be zero. Some of them will be positive numbers, greater than zero. We pick one of the positive numbers, and then knowing the fact that this product has to equal zero, we can now solve for 'w$_0$', because we have the vector 'w', we've picked an 'Alpha$_i$' that's greater than zero.

We can solve for 'w_0', such that, this entire term is equal to zero. And what we can show, is that, no matter what I we pick, such that, 'Alpha$_i$', is greater than zero, for all 'i', where 'Alpha$_i$' is greater than zero, the solution is going to always be the same for 'w_0'.

Video 4 (9:06): Primal and Dual Problems II

So let's look at more– in more detail, at how to understand what exactly it is that the dual problem is trying to do? So we've defined the original primal problem, we then switched over to the dual, through the Lagrangian representation, and said for some reason that, working with this dual–maximizing this dual is preferable. Here is where we can see, that maximizing this dual is doing something, that we originally said, that we wanted to do, which is, finding two points in the respective convex holes, of the two classes, that have the minimum distance to each other. So minimizing the distance between the convex holes. All right! So, let's look at the dual problem again. This is what I had written previously. We want to maximize this objective function, over 'Alpha$_1$' to 'Alpha$_n$', such that, with the constraints, that each Alpha has to be greater than, or equal to zero. And again, the sum of the Alphas in class one, minus the sum of the Alphas for data corresponding to class minus one, are, is equal to zero.
So remember again, a 'y', is plus one or minus one. We sum up all the 'Alpha$_i$' s that correspond to data, that correspond to an 'i' for class plus one, that has to equal, the sum of all the 'Alpha$_i$''s corresponding to data in class zero. So that's what I have here. I...I—This constraint simply means that the sum over all data points in class one of Alpha is equal to the sum over all data in class minus one of the corresponding Alpha. Now let's call that, 'C'. So, 'C' is equal to these two sums. We know that it's equal to one number because they have to equal each other. So now if we look at this term here. This is the second term in the objective. We can rewrite this in this way. Notice that, what this actually is, this second term, is equal to, the magnitude of 'w' squared. Remember we, define this thing to be equal to our vector 'w', that we want to minimize the 'l_2' norm of. So the magnitude of 'w' squared. Equivalently, we can write this, in this way, where we sum over all the data in class one. 'i' times 'x_i' divided by, 'C', minus the sum over all the data, in class minus one of 'Alpha$_j$', 'x_j' divided by 'C'. So the plus here, corresponds to the fact that, 'y' is going to be plus one for these data points. The minus term here corresponds to the fact that, 'y' is going to meet minus one, for these data points. And then I do a simple trick of multiplying and dividing by the same thing. So I have divided by 'C', here. In both cases, the square terms means that, I can, I need to multiply by 'C' squared in order to have these two terms cancel out with each other. So this is another way of writing this second term.
So I plug that right back in. Notice that the first term here, the sum over all these Alphas is simply equal to two times 'C', where, I define 'C', to be the sum of the Alphas corresponding to one of the classes, because it doesn't matter, which class I pick, they're equal to each other, that's two 'C'. And then minus one-half times this, this term, that we just discussed on the previous slide. subject to the constraint that, the sum of these two things has to equal constant 'C'. And also that they have to be positive or equal to zero. Okay, so again, this is not necessarily a fully rigorous proof. Okay, I'll start that part over. Again, so this is not necessarily a fully rigorous proof. But if we look at, what we're trying to do here, when we

maximize over 'Alpha$_1$' to 'Alpha$_n$'. Equivalently, we're trying to minimize this second term. So we can show that, by maximizing this thing over the vector, 'Alpha$_1$' through 'Alpha$_n$', that maximum is also going to minimize this term right here, right? Because this term has to be positive, or equal to zero. It can't be negative because of the square right here, and we're subtracting it. And so if we want to maximize something, we can equivalently minimize— or what we equivalently are minimizing the negative of something. So, in a sense, what we're trying to do, not even in a sense—so okay. To conclude. So when we think through what the primal is trying to do by maximizing over the Lagrange Multipliers, 'Alpha$_1$' through 'Alpha$_n$'. We see, or we can think of it as trying to minimize the, finding the weights, 'Alpha$_1$' through 'Alpha$_n$', such that, if we construct a probability distribution using those points. We take all the Alphas for class one and construct a probability distribution by normalizing them. And then, we take all of the Alphas from class minus one and construct a probability distribution by normalizing them, which is what 'C' is doing. Then we're trying to minimize the distance between two points within the respective convex holes, of each of the two classes. Okay, so let's return to the picture now and see if we could find two points, one point in each of the convex holes, such that, we minimize the distance between the convex holes. Then what does the direction, of the classifier, defined by this hyperplane, look like? So we've already discussed, the geometry of the problem. If we want to find a hyperplane that has this decision boundary, then we know that the classifier, the vector 'w' has got a point perpendicular. So any vector 'w', say, if this is my 'w', then this defines a separating hyperplane that's perpendicular to that vector 'w'. Similarly, the vector 'w', that we construct that's going to be perpendicular to this, this line, connecting the two convex holes, has got to be pointing in the direction of that line. In other words, if we have a point, 'v' in class zero, minus one, say, that's 'v', and this is 'u', in class plus one, then, the vector that defines this to be the positive region and this to be the negative region, that has this as its hyperplane, the separating hyperplane, has got to be pointing in this direction from 'v' to 'u'. Remember, the arrow says, which direction, which side is plus one.
And then the other direction from the way the arrow's pointing is the side that says minus one. And the hyperplane is perpendicular. Therefore, we simply take the, we can construct 'w' by finding the points in the minus one class on the hyperplane. Therefore, we can construct the vector 'w', that defines this hyperplane by taking the point on the convex hole to the minus one class, and the point on the convex hole of the plus one class, And taking their difference. And so that's exactly, again, if we look at what the vector, 'w' is, that's exactly what it's doing. We construct 'w' by taking Alpha times 'y' times 'x' and summing over all of the data points. If we just rewrite this in another way, we use our definition of 'C', to be the sum of all of the Alphas in one of the classes. Then equivalently, we can write this sum as being, the difference between, the convex hole and the minus—the point on the convex hole and negative one class, and the point on the convex hole of the plus one class. So this point here corresponds to that point. This vector here corresponds to that point. Their difference corresponds to a vector, pointing in the direction from one point to the other point. And then we're simply scaling it, by some number 'C'. So that's simply stretching it out, or shrinking it.

Video 5 (15:22): Soft-Margin SVM

Thus far we've been discussing the support vector machine, under the assumption, that the two classes are linearly separable. So what happens if, they aren't, and this is going to be more likely, the case. How can we modify the SVM in order to take into account the fact that some data points might lie, might of necessity fall, on the wrong side, of the separating hyperplane? So to handle this, we discuss something called the soft margin SVM. And the difference between, the SVM, as we've been discussing it thus far, and the soft margin SVM, is the introduction of this thing called the slack variable. So for every single data point, where we have this original constraint that this had to be greater than or equal to one, we now introduce a slack variable, 'Xi$_i$', that has to be positive, or equal to zero. And then we allow some slack here. So we, for some data points allow 'Xi$_i$' to be greater than one, in which case we allow for that particular point 'x_i', we allow for it to be misclassified. So, here's a picture of that—to give some intuition. So here we have, the separating hyperplane, and then for a particular point, for example, this one that falls on the wrong side of the hyperplane, we allow the slack variable to be greater than one. So that accounts for the fact, that we don't actually perfectly classify that point, that the sign of 'y', and our linear classifier disagree. Whereas for a point, for example, here that falls on the correct side has a 'Xi', that's less than one and for some of these other points, 'Xi' would be equal to zero. So this is what we're looking to add to our SVM. So we therefore, need to formalize this by building it into our objective function. So the way that this is done, is that we take the original objective function— so the minimum—we wanted to originally minimize this term, and now we add to it, the sum of these slack variables. And because they have to be positive or equal to zero, that's a constraint that we add, this sum is going to be a positive number. And we pre-multiply that by some regularization parameter, Lambda, that says how strictly we're going to enforce linear separability. And now, and we also modify our constraint by subtracting 'Xi$_i$', from the ith constraint in order to, potentially allow the ith data point to be misclassified. And now we want to minimize the sum of these two terms, so we want to minimize the 'l_2', norm squared of the regression hyperplane, defined by 'w', and we want to minimize the sum of these slack variables.
So we don't want to set too many greater than zero. We can quickly see the role of Lambda, in this setting. So for example, if Lambda is extremely small, say in the limit as it goes to zero, then we're essentially paying no penalty for letting 'Xi' be greater than zero. And so, we're really allowing anything to be misclassified. We can't really learn that in that way. When Lambda gets bigger and bigger, as it goes for example to infinity, then we're paying an infinite, an infinitely large penalty by letting any of these Xi's be greater than zero. And then, we get back to the original SVM where, we're trying to enforce perfect separability. So Lambda is this additional parameter now that we have to set its number greater than zero and it can be set in a few ways. The most common way would be to use something like a cross validation type technique. The important thing about Lambda is that it's going to change the hyperplane that we learn. For example, here's a case where, Lambda is very large and so we learn this hyperplane. And these are the points that, have their corresponding Alphas greater than zero. And then for Lambda, very small we would learn, for example, this hyperplane. So the take away from these slides is, that the hyperplane is going to adjust itself based on what we set Lambda to be. So let's look at how we can learn the soft margin SVM. Again, we have our primal problem, where, we have the sum of the

two terms which is a tradeoff of the original thing that we wanted to minimize plus a penalty on these slack terms that we're including, subject to these two constraints. So the original constraint, including the slack that we're now going to allow, as well as the constraint that the slack variables have to be non-negative.

We again construct the dual, by adding Lagrange Multipliers. This time, we're going to add a Lagrange Multiplier 'Alpha$_i$', to enforce this constraint, and another Lagrange Multiplier, so this is new, 'Mu$_i$' to enforce this constraint. And we have this, dual problem that we now want to minimize, over 'w' and 'w0' and these slack variables, and maximize over the Lagrange multipliers. In this term, we have the original objective, that we want to minimize. We add our Lagrange Multipliers, so this is from before, with the additional slack variable here. And then also minus the Lagrange Multiplier time— the new Lagrange Multiplier, that we include to enforce and penalize the slack variable 'Xi$_i$'. And this now we want to maximize over, 'Alpha$_i$' and 'Mu$_i$' subject to the Lagrange Multipliers being non-negative. And also subject to this term now being non-negative, where we've introduced the slack which allows us to violate the original non-negativity constraint. Okay so now let's look at solving for the dual problem. We minimize, 'L' over 'w' 'w$_0$' and each 'Xi$_i$', by taking the derivative and setting to zero. We get these two original constraints again, by minimizing over 'w' and 'w$_0$'. So this is exactly as before, plus this additional constraint where, which is— comes from taking the derivative with respect to 'Xi$_i$'. So from this, from minimizing over 'Xi$_i$', 'Xi$_i$', actually disappears and we're left with this constraint, at Lambda minus 'Alpha$_i$' minus 'Mu$_i$' has to equal zero.

So now, if we plug 'w', back in, and I have a typo here— this term, if we solve for Mu, gives us Lambda minus 'Alpha$_i$'. If we plug 'w', back in, and we plug for 'Mu$_i$' back into 'L', Lambda minus 'Alphai', we'll find that we have the dual problem, which is equal to this. So actually, we have exactly the same maximization problem, over Alpha subject to this constraint, which is— comes from this requirement here, plus an additional constraint that 'Alpha$_i$' has to be, greater than zero, which was as before, but now less than Lambda, which is the new thing. And so, this is all that's new with the SVM including slack variables, this Lambda term here. That's the only difference. Now, 'Alpha$_i$' has to be less than Lambda as well. So where do we get this? Well, 'Alpha$_i$' and 'Mu$_i$', are both, positive or non-negative. And Lambda is something we set, and we have this requirement, that 'Alpha$_i$' plus 'Mu*i*' has to equal Lambda. So of course that means, that if, 'Alpha$_i$' is greater than Lambda, the only setting for 'Mu$_i$', that we can have to make it equal to, Lambda would be a negative number, which violates the constraint on 'Mu$_i$'. So, by the fact that, 'Mu$_i$' has to be positive, that means that 'Alpha$_i$', has to be less than Lambda, because 'Alpha$_i$' plus 'Mu$_i$' has to equal Lambda, according to this constraint which we get by minimizing 'L' over 'Xi'. So this is the SVM, where, we've now introduced slack variables. We have this, as the only difference. But we also now notice, what if we kernalize the SVMs? So we instead of working with 'x', we map 'x', to a higher dimensional space, then we end up with these dot products between these higher dimensional mappings, which we've already discussed can be replaced with any valid kernel, that we want to define on 'x' and 'x$_i$', and 'x$_j$'. So we can simply replace this dot product, with a kernel function, between the two data points, 'x$_i$' and 'x$_j$'. By swapping this out, and swapping a kernel function in, we kernalize the SVM. So now, we have a kernalized SMV with slack, that's the final SVM, that is the most popular, and the most versatile, SVM.

So how do we do classification? Exactly as before, we have that 'w', now is equal to, 'Alpha$_i$', 'y$_i$', times the mapping of 'xi', if we want to do a higher dimensional mapping. And now sum over each 'i', so we've taken 'x$_i$', out and replaced it, with its higher dimensional mapping. We get that, 'w', equals this and just as we've discussed in previous lectures about kernels, when we make a prediction for 'y$_0$', we're simply predicting the sign of the dot product of the mapping for the corresponding 'x$_0$', with 'w' plus 'w$_0$'. And with this, from this mapping, we get that, we can replace the dot product with the kernel, and that gives us the prediction. So we've...we now predict based on the sign of this function here plus 'w$_0$'. So here's what we end up with, on this data set that we've seen a few times before. The black line here corresponds to the decision boundary, which arises from this classification rule. So we've done the kernelized SVM, where, we use the RBF kernel, Gaussian kernel. We make our predictions based on a sum over 'Alphai' times 'y$_i$', times the kernel between our new input point, 'x$_0$' and all of the data points in our data set plus 'w$_0$'. That'll give us, something like this, and from the SVM algorithm, many of these 'Alpha$_i$''s will be equal to zero. And for any 'Alpha$_i$' that's equal to zero, we can interpret this, as saying that we don't consider the ith data point in making our classification.
So any data point for which, 'Alpha$_i$' is positive is going to be a data point that we use to classify. Those are called the support vectors. So the data points 'x$_i$', for which 'Alpha$_i$' is positive, because we view these data points as vectors, they're called the support vectors. So for something like this, each of these black dots is placed over a data point that has 'Alpha$_i$' non-zero. So for example, the 'Alpha$_i$' corresponding to that data point is equal to zero, whereas the 'Alpha$_i$' for that data point is greater than zero. So these are the data points that are actually going to be used—these black dots, in making our classification. After running the SVM, of course, all of these data points were used to learn the SVM, but after learning it, we could literally throw away all, any data that doesn't have a black dot over it. We only need to keep the data that has the black dot. And we only calculate the kernel between the new point and the points that have an Alpha greater than zero. So that, wraps up support vector machines. To summarize, with the basic SVM, we had a linear classifier, assuming that the data was linearly separable, the position of the Affine hyperplane was determined, so that, we could maximize the margin between the two classes.
So that's what the SVM was doing for us. It was maximizing the margin between the two classes. The dual problem we set up to now maximize the dual, we didn't go into detail about this but we can show that it's a convex problem or it's a concave problem. We want to maximize it, so we have a convex optimization problem, meaning that, whatever algorithm we have, when it converges, we get the solution to the problem. We find the values of Alpha that maximize that objective function. The full-fledged SVM, which we discussed in the second—last part of this lecture includes maximum margins, so it's still a maximum margin approach, but it includes slack variables, which allows for overlapping classes. It also includes a kernel, so we implicitly map the data to a higher dimension and then perform the maximum margin classifier in that higher dimension. But when we solve the dual we see that we only need the kernels, we don't need the mapping at all. So it includes, the dual includes kernels, so that we can learn non-linear decision boundaries in the original space. We're still learning a linear classifier, in the higher dimensional map space, but when we look at what the decision is in the original space, we get non-linear decision boundaries. And so that's what, makes the full fledged, quote unquote, full-fledged SVM so much more versatile and useful. And that's where kernels become very useful.

In practice, there are many software packages, so we aren't going to discuss the actual algorithm for maximizing the dual over Alpha, but there are many software packages that are available. In practice, we also need to choose a kernel function which, if we choose, the RBV means, we have to choose a kernel width which says, it gives a definition of proximity in the original space, so that's an important parameter. And the way that we can do that is using cross validation. So we can cross validate for Lambda, which is the penalty on the slack and also cross validate for the kernel width, if we choose to use the RBF kernel, which is the most popular kernel.

Video 6 (11:42): Decision Trees

Next, we're going to look at a technique for doing classification or regression. It's very different from what we discussed, thus far. It's not a linear classifier or a linear regression technique. It's something called, a decision tree. So a decision tree, maps an input 'x' in R^d to an output 'y', which could be either a real value or a class, using a sequence of binary decision rules. So we have a tree, and each node in the tree has a splitting rule. Or, if it's a leaf node, it has a decision rule. And each splitting rule is going to be of the form–look at the j^{th} dimension of 'x'. So in this case, 'j' is indexing the dimension of 'x' for the this slide. And if it's greater than 't', go in one direction down the tree. If it's less than 't', go a different direction. So, for example, look at the first dimension of 'x', if it's greater than one point seven, go the left. If it's less than one to seven, go to the right. If we go to the left, we hit a leaf node, and that leaf node is going to tell us, "Okay, you've terminated. Declare class one." If we go to the right, for the specific example, we would look at the second dimension of 'x'. If it's greater than two point eight, we go to the left, and because we have the leaf, we declare two.
If the second dimension is less than two point eight, we go to the right. Again, we hit a leaf. We declare class three. So this would be a three class, classification problem, where we need to look at these two dimensions of 'x' according to this sequential splitting rule, in order to make a decision. The motivation behind regression trees and classification trees, decision trees in general, is that we want to take the input space. So this is the space that we've been calling–that 'X' lives in, the covariate space or the feature space. And we want to partition it into regions, so that all data that falls within the same region has exactly the same prediction. So for regression or classification, if a data point falls within a particular region, any part of that region, it's going to have the same exact prediction. So for example, if we wanted to partition a two-dimensional input space in this way, where all the data, for example, in this region would have the same prediction, we would see that we couldn't use a decision–a binary decision tree to define something like this. So, this would be an example of what we're going for, but we also want a simple way of deciding how to split this region. And, we could not come up with a binary decision tree that would give us this type of a splitting. So, something like this is what we can obtain with a binary splitting– decision tree, where we have partitions of subspaces. Just fading out.
start over! So the goal, of a decision tree is to take the input space and partition it into regions. If we wanted to partition the input space into regions in general, we might say this would be a partition that we want. However, we could quickly convince ourselves that this type of a region is not something we

can get with a binary decision tree. So the partitions that we're interested in, for the purpose of this lecture, are those that can be defined by binary splitting trees. So not all partitions of the input space are going to be possible with the technique we're going to discuss. And the reason for that is because a binary decision tree is something that's very easy to learn, and it's a very simple rule, something that's very easy to do. So these are the types of partitions that we're going to be able to get, with a binary decision tree. If we want to represent a split, like this, in terms of a decision tree, we can visualize it like this, where for example, we would first look at the first dimension of 'X' and see if it's less than or greater than 't_1', if it's less than 't_1', we go to the left in which case, we now have these two regions that are possible–regions for 'X', for the input 'X'. But then, we have to look at the second dimension. If it's less than 't_2', then we're in region one. If it's greater than 't_2', we know we're in region two. So all of–all five of these regions can be easily found by following this–the rules of this decision tree. If we have a new input 'X' and we want to say which of the five regions is this 'X'...does this 'X' fall in, we can quickly find it according to this decision tree. Whereas, if we just had this sort of a representation, if we had some other way of representing this, it might be much more complicated to actually find out, especially if the data is in much higher dimensions, which of the regions does it fall in? The decision tree can give us that extremely fast.
So that's why, we allow ourselves to be constrained to the regions that can be defined by a binary decision tree. And then, if we're doing regression, we have a partition, for example, like this. So here, this plane is the bottom of this cube, and then, for particular region, for example, 'R_1', every data point in that region will be predicted to have this value. So 'y' is now this third dimension is now the output 'y' that we're going to predict. Any data that falls in this region, 'R_2', will predict 'y' to be zero. Any data that falls in 'R_1' will have this prediction. And so we see that with these regression trees, we are taking the space and learning these sort of step functions to try to model the data. For classification, it's essentially the same. For example, let's look at this classification problem, where we have irises, and the irises that we want to predict can fall into one of three classes. So there are three different types of irises. The way that we make this prediction is by taking an iris, and measuring two features. So we extract two features from a particular iris. We extract the sepal length divided by the sepal width. So one of these is a petal. One of these is a sepal. I don't remember exactly which is which. But we have petals and we have sepals, in these irises. And so we take a petal, and we calculate the length of the petal divided by the width, and average that over all the petals. That gives us one feature for that particular iris. And then, we do the same with the sepals. We take the length divided by the width, and average. That gives us a second feature. And then, we plot these for many irises, for 100 or so irises. And then, we color code it by the class that they fall in, and we get something like this. So for example, the red class of irises, what we're calling red 'x's', all have the two- dimensional features that fall in this region. Whereas, the blue class and the green class are distributed this way.
So this is actually, in addition to our conversation about decision trees, is a good place to discuss the ideas of feature extraction. So we have a...a bunch of irises, and we have labels for those irises. Somebody–some expert has told us what each iris is. But now, we want to have a quick way of taking a new iris and, you know, predicting a– what class it belongs to. So we have to extract features from each iris, in order to do something like this. If we just had a picture and wanted to analyze the picture somehow, it would be much, much more complicated. So we've taken a complex object, and we've

defined a rule for getting two numbers that can...that can represent that object, the ratio of the sepal, length versus width, and the ratio of the petal length versus width. And now, we've taken a particular iris, and we've reduced it to those two numbers. So this iris might be reduced to that number– that point right there. Let's look now, just as an example, a walkthrough how would we–what would a decision tree on this type of a data set look like for classification? So first, imagine that we have the trivial tree, where there's just the one node. So there's not even a decision that needs to be made. We simply want to predict which class each point belongs to. In that case, what we would do is we would predict, again, the most probable class. So, we basically say that there's one region and all data fall into that same exact region. And, because class two is the most probable class, we declare every point in that region to be class two. Now, if we want to have a...a decision tree that only has one decision in it, we might, for example, say, okay, look at the first dimension. And now, if a data point is greater than one point seven, so that's this line here, if the first dimension is greater than one point seven, it's in one region.
If it's less than one point seven, it's in a different region. So now, we've divided the space into two regions. And for this particular region, for this region here, we're now going to come up with a prediction of all the data in this region. We simply make the prediction to be the most prevalent class. In this case, 'y_1' is the most prevalent class in this region. Whereas in the second region, the most prevalent class could be 'y_3'. Now, we want to make another split. And so in this case, we aren't going to–we wouldn't want to split this region because it's perfectly classified. But, if we split this region. For example, if we split 'R_2' along the second dimension, where we look at whether a point has its second dimension value greater than two point eight so that's this region, or, less than two point eight so that's this region, then we could make another split. So, we haven't split the entire space along this point. We're just looking at the second region and making a split along this dimension. The first region would be reached, if we went in this direction, in which case, we don't ever make another split. So that's why, this split is only taking this entire region and partitioning it into two parts. And now, we would look, is it greater than two point eight? If it is, we go to the left and we have class two, that's the most prevalent class. And if it's less, we go to the right. We have class three. That's the most prevalent class.

Video 7 (8:01): Basic Decision Tree Learning Algorithm I

Now we need to actually come up with a way to learn this decision tree from data. And the most common way is to use, something called a greedy algorithm, and that's what we'll discuss today. So this is an algorithm where we basically follow the same sequence of events from the previous slides. We start with one node, and then we decide to split that one node into two children. And, we have to make a decision how to split it, and then given that split, we have to now decide to take a leaf node and split it into two more children. So for example, we would evaluate both leaves, see which one would be the best to split, and we would make this split. So we grow the tree from nothing to something. First, let's briefly consider learning a regression tree. So imagine that we have five regions. What is our regression function look like? We take an input 'x'. We evaluate the function of 'x', which is now going to be equal to this sum, where we take the prediction for particular region 'm', multiply it by an indicator whether 'x'

falls in region 'm', and then sum over all the regions. So this indicator will be true only for the region in which 'x' falls, which will then pick out the correct prediction 'c_m'.
Imagine that we have this partition of these regions, and we wanted to find the optimal values of each 'c_m'. How do we do this? So the way that we–the first thing we have to do is set up some sort of an objective function. And for regression trees, the default objective function is going to be the sum of squared errors. So we take, for each data point, the true output, which is a real value number minus the predicted output, square it and sum it. So this regression function, conditioned on the partition, is simply going to break down into 'M' different problems based on which region we're considering. So if we want to learn 'c_m' for corresponding region 'R_m', we're summing the sum of the squared errors for all of the data restricted to their falling in that region 'R_m'. And so, it's straightforward to show that to find the optimal 'c_m', we simply average the corresponding outputs for all of the data that are in...in that same region. So we look at all the data. We say, which of those data points fall in region 'R_m'. We then, take their corresponding outputs 'y' in the training set, and then average over those outputs. That's how we can minimize the sum of squared errors. So more complicated is finding these regions, to begin with. Given the regions, we can simply take all of the outputs corresponding to the data in a particular region and average them. That minimizes the sum of squared errors. But how do we actually find these regions? So let's consider that we have a particular partition, like this. What we would do is simply look at each region. So it's really just a brute force method. We look individually, at each region. We look at each dimension for all of the data in a particular region, and decide what's the benefit we can get by splitting along that point.
So for example, if we look at all the data in region 'R1', we can then split it along either the first dimension or the second dimension. So we literally, take 'R_1' and evaluate all possible points at which we can split it along each dimension. And say what would the result be. How much would I reduce my objective function sum of squared errors, if I did that? So let's consider splitting a particular region 'R' at the dimension 'j'–along the dimension 'j', at a particular value 's'. So for example, if 'R' is this region here, we look at the first or second dimensions, so 'j' equals one or two, and then split it at a particular value along those two dimensions. We then, have a proposed dimension 'j' and a proposed value 's' for a particular region 'R'. We then say, how would this partition the data in region 'R'. So we construct, a set R^-, which is all of the data that's in region 'R', such that the jth dimension is less than 's'. So we say, the resulting partition would put all of the data in region 'R' into either the negative group or the positive group, depending on the value in the jth dimension, and how it compares with 's'. We then, update the proposed prediction– the proposed value for 'c', if we made that split, and we look at how much would we reduce the resulting objective, minimize the sum of squared errors.
So we do this for every single region. We do it for every single dimension in each region. And we do it for all of the possible splitting points 's' that would lead to unique subsets, a unique partition. And we can, for each of those proposals, get a new possible regression function 'f' and we evaluate each of them. So it literally is a brute force method. The computer can do it very fast. But, it's literally just checking every single possible split we can make and picking the best one, the one that minimizes the sum of squared errors the most. For classification, it's a bit more complex because we need to somehow measure how good a region is performing. With regression, we can look at the sum of the squared errors for all the predictions that are made for all data in a particular region. For classification, we don't have a natural

measure of performance. So there are three measures that we can possibly use. One of them–so what we will do is for the K-class problem, we have 'K' classes. We're going to look at all data in a particular region, and call it 'R_m', and let 'p_k' be the empirical fraction of the data that have label 'k' in the region 'R_m'. So we...we look at a particular region 'R_m', and we construct an empirical distribution on the labels for only the data that fall in that region. Then to measure the quality of our prediction– because remember our prediction for a particular region is just the most prevalent label, we predict the most probable label. The measure of the quality then, for that prediction could be either the classification error– so one minus the maximum of 'p_k'. So if 'p_k'– for a particular 'k' the maximum of 'p_k' is the most prevalent label in that region, so we won't get that one wrong.
But we'll get one minus 'p_k', we'll get that fraction wrong, so that's a classification error. We can also use the Gini index, which is one minus the sum of squares of 'p'–'p_k', or the entropy, which is this term. So each of these is going to be maximized or large, when 'p_k' is uniform, when we are completely uncertain about what the label could be for that region. We have equal number of all the labels in a particular region, we maximize these things. Of course we're trying to minimize them. So when they're minimized, all of the data in a particular region just falls into one class. That's when all three of these are minimized, in those cases, and they're all equal to zero.

Video 8 (9:04): Basic Decision Tree Learning Algorithm II

So let's look at an example for growing a classification tree. Imagine that we have this current setting. We have region one and region two. And we now...we now want to decide which of these regions to split and along which dimension to split it, and where to split it. Our current tree looks like this. Our current tree says take the first dimension, and if it's less than '1.7', go to the left and declare class one. If the first–the value in the first dimension is greater than '1.7', go to the right of the decision tree and declare a class three, which is–the class three is the most prominent class in this region. Class one is the most prominent in this region. Now we have two possible regions that we could split. We could split this region, along either the first dimension– so we could either put another line here– or along the second dimension and put a line here. Or, we could split this region along this dimension, meaning put a line here or along the second dimension, put a line somewhere along here. We have to decide which one of these regions to split, where to split it, and along what dimension. So first notice that this first region classifies class one perfectly. So we're not going to improve the objective function, whatever we choose the objective function to be. We're not going to improve it by splitting this at all. So for the objective, let's look at the Gini coefficient, just to pick one. And let's look at splitting this second region. So, currently this region has a Gini coefficient, so that's what this is indicating, 'u' of 'R_2' has a Gini coefficient equal to one, minus the sum of the probabilities of each class in this region.
So there's one over '101'. So the entire data set in this region is '101' points. There's one class one, so we have one over one o one squared, minus '50' over one o one, minus '50' over one o one, both squared. So there's '50' of the blue points and '50' of the green points, and there's a total of '101' points in this second region. So we have a Gini coefficient of point five o nine eight. We want to now, pick a

dimension to split and a point at which to split to minimize the resulting Gini coefficient of the two regions. So what is that mean? What is the improvement that we get from a split? Well, we have our first Gini coefficient from the original region. We then, choose to split this region into two different regions. So call one of them 'R_M' minus and one of them 'R_M' plus. For example, if this is where we split, then this region would be 'R_M' minus and this region would be 'R_M' plus. We calculate the Gini coefficient for the resulting data in those two regions, like this. And we multiply it by the fraction of the data that fall in those two regions. And so, this is the result that we would get. This is the Gini coefficient that we would get for a particular split. We look at the difference, and now we want to minimize this thing. So let's actually check both dimensions at all of the possible points for this particular region.
So what this plot is showing is for the first dimension, so that's this dimension, split it at a particular value for 't'. So for example, if you split it here at two, you would get this region. So if you split it here at two, you would get this partition. You have all of the data following in this region in 'R_2' minus, all the data in this region in 'R_2' plus. And then, let's check the reduction of the Gini coefficient, for that point, so we get this value. So we see that the reduction as a function of 't'. So 't' here can be roughly '1.7' up to roughly three, so that's what this– these possible values correspond to these values here. At a particular split, we get their reduction value, and we look at the–so this is absolute value of the reduction– we look at the largest value, so for example, right here, at about '2.25', so that's it right here, corresponds to the best split, if we were to split along this first dimension. So the best split for this first dimension is right about here. And we get that magnitude of a reduction. Now we look at the next dimension. We look at the second dimension, and split it–values between two and say roughly '4.5', which is where the data–there's no data above '4.5', so we don't have to check up here.
So we check at that interval between about two and '4.5', and we propose splitting at each of those points and see how do we reduce the objective function or the Gini coefficient, as a result of a particular split. And we suddenly see that at this point here, which corresponds to roughly this point here, we get a reduction of '0.25'. And if we look at the maximum for the previous dimension, it was less than '0.02'. So the reduction we're getting by looking in this second dimension is massive, compared to the reduction from the previous one. So the previous one was maximized at about this point. We're getting a huge reduction in the resulting Gini coefficient, and that's intuitively–just says that this split makes us much more confident about the two classes in each of the resulting regions. So that's where we then decide to take the second dimension, and split at this particular point for this particular region. So that's–it's very brute force, it's not elegant by any stretch of the imagination, but that's what the algorithm does. And of course, computers can do this extremely fast. So one thing that we sidestepped is, when do we stop growing this tree, when do we stop splitting regions.
And the amount of uncertainty that we reduce–we get–the reduction of the uncertainty is not a good way to decide this. So for example, look at this setting, where we have these points are in class one and these points are in class two. Any particular partition that we make is not going to give us any reduction of the uncertainty. No matter where we split to make this first one region into two different regions, we're not reducing our uncertainty. In all splits that we can make, we have the same amount of class one and class two in both splits. However if we could make our region split this way, suddenly we perfectly predict. So uncertainty reduction is not the way to do it. In practice, what is done is that the trees are overgrown, so we make many more splits than are possibly necessary to get many, many

regions. And then, we prune the trees back using an algorithm. So this algorithm is not something that we're going to discuss in this course. It's not a trivial algorithm, but that's what's done in practice. We grow out the tree much bigger than we need it to be, and then we run an algorithm to prune back some of the branches, in order to get a good tree. Another problem with these regression and classification trees is the idea that–of overfitting. So if we grow a tree out too big, we can eventually partition the input space into enough regions, where we perfectly classify or perfectly predict everything. However we get this type of a situation, where the number of nodes in the trees, if we increase it reduces the training error continually, but the testing error on a held-out set at some point starts to get worse and worse. So we always at a–in a certain regime, we're improving our testing error, but then at a certain point our trees become too complex and we start to overfit to the training data, and we get worse and worse testing predictions because it doesn't generalize the new data as well.

Video 9 (6:58): The Bootstrap

For the next part of the lecture, I want to discuss, first, a general technique that's used for much more than just trees...decision trees, and then this technique, which is called the bootstrap, I'm going to apply to the decision-tree problem. First let's discuss the bootstrap, in the abstract. So this is a very, important statistical technique. It's very general. And we're going to see how it can be used to learn something, called ensemble classifiers. The bootstrap is essentially, where we resample from our data. We use these resampled data sets to improve some sort of an estimator. And what we're going to see when we construct these ensemble classifiers, is we are going to generate many mediocre classifiers. So we're going to get models that are-each individual one is going to be not so great. However, when we put them together using a technique called bagging, their kind of...their combination suddenly makes the overall classification much better. So first, let's go through the bootstrap in general. What we have as input is a set of data 'x_1' through 'x_n', and also an estimation rule 'S hat' of a particular statistic 'S' that we're interested in. So for example, our data set could be 'x_1' through 'x_n', and we might be interested in learning the median of the true underlying distribution that generated this data. So we don't care about the median of the actual data that we have. We want the median of whatever distribution generated this data, that we don't get to see. Of course because we don't have the entire–we don't an infinite number of samples from the underlying distribution, we can't find out what exactly is the median.
So what we do is, we have an estimator 'S hat', which estimates the true underlying median to be the median of our data set. All right, so this is the setting that we're in. This motivates something called the bootstrap algorithm, and so I'm first going to discuss it, in the abstract. First, what we do is generate a set of bootstrap samples. So, we're going to call these 'B_1' through 'B' capital 'B'. So we have capital 'B' different bootstrap samples. For each individual sample, we construct this set by picking points from our data set, randomly, 'n' times with replacement. So, we have 'n' points in our data set. For each sample little 'b', we construct a brand new data set, also of size 'n', which we construct by sampling 'n' times from our original data set, uniformly at random, with replacement. So, each particular point 'x_i' can appear in this set, multiple times. So in this sense, each one of these sets is only going to contain data

from our original data set. However, each set is going to have each individual observation, duplicated a certain number of times or not contain particular observation. So a particular 'x_i' might not appear, in a particular sampled data set. So each of these bootstrap data sets, is like a view of our underlying-of the original data set. And then, when we have–when we want to calculate our statistic, we simply calculate it using the bootstrapped set. So for example, for the median, we calculate the median of the bootstrapped data set and then, we estimate the mean and the variance of our statistic, by averaging over our bootstrapped calculation of this statistic, and then calculating the empirical variance, which gives us some uncertainty. So the power of this idea, can be seen, in the example of estimating the median of the data set.
So imagine that we have 'n' observations, where each is in 'R'–so 'n' one-dimensional values. And then, we want to find the median–estimate the median of the distribution that generated this data set. The way that we do this is we, simply sort these 'x_1'–these values in ascending order and then we pick the middle one. Or, if there's an even number of data points, we pick the average between the two middle ones. And then, that's our estimate of the median-of the underlying distribution that generated this data set. So the issue is how confident can we be in that estimation? Well, if we want to know how confident we should be in our prediction of the median, we would have to find the variance of that median. But how can we do this? We only get for a particular data set– it seems like we only get one possible value for the prediction of the median. How can we get multiple values or, how can we somehow evaluate our uncertainty about that prediction? So that's where bootstrapping comes in. What we do is we generate, capital 'B' different bootstrap data sets. For example, for 'B_1', we simply pick 'n' times from a distribution uniformly on our data. So again, some of these are going to repeat, and some of them are not even going to appear in 'B_1'. And we do it randomly. We then, calculate the median of each individual bootstrap data set. So we could get capital 'B', different estimates of the median. We then say that our expected median– the median that we're going to predict has an expectation of the average of the median of each individual bootstrap data set. And our variance, or uncertainty about that prediction is the empirical variance of the medians that we get from each of these individual data sets. So, that's our way to gauge our uncertainty about our prediction of the true underlying median. It's remarkably simple, but there's quite a bit of theory, underlying this technique that relates this simple technique to something that's actually theoretically correct.

Video 10 (7:38): Bagging and Random Forest

So the term 'bagging' comes from the full name, which is bootstrap aggregation. So we bootstrap the data set, we do some– we learn some model on each bootstrapped set, and then we aggregate the results to get one final prediction, that's called, bagging. All right so our algorithm is as follows, for little 'b' equals one to capital 'B', where we decide what this capital 'B' is, we first generate a bootstrap data set of the same size as our training data. So this data set is going to have repeats of certain data points, and some other data points are not going to be in that data set. We then learn, a classifier, or a regressional model 'f*b*' on the bootstrap data set. So, we have capital 'B' of these classifiers, or

regression models, where each one is learned on a particular bootstrap data set. Then, for a new point 'x_0', we want to come up with a prediction, and that prediction is simply the prediction that we would make or the average of the prediction we would make, for each individual bootstrap data set. So we evaluate the new point on the classifier or regression function learned for the bth bootstrap data set, we sum over all of them, and then we average.
So for regression, this average would be the prediction. For classification, we can view–each classifier is like giving its vote for what the class should be, and then we pick the majority. So when we want to bag trees, here's something–here's what we get. Imagine that we're doing binary classification. Our data is in R^5. This would be the original tree that we would learn, if we looked at the entire data set and learned it on the original data set. But if we first, give–resample from our data set 'n' times and learn a tree. This would be, for example, the first tree we learned, the second tree, the third tree, up to the 11th tree. And so the reason that each tree is different, is because they're learned on different data sets. Each individual data set contains our original data in it, but some are duplicated and some don't appear at all. So these are 11 different views of our data set, which would give us 11 different trees. Each one of these trees doesn't have to be great, but their combination is suddenly going to improve the overall prediction. So we can see that when we go from one tree up to, say 25 trees, using these bootstrap samples, we get a clear improvement but then at a certain point, the improvement stops. So that says that at a certain point, bagging more trees is not going to improve our classifier, at all. However, as described, there's a certain limit to the performance, and in general the reason is because each of these trees that we learn is highly correlated with each previous tree. And so that's why, we suddenly stop improving. So this leads to the idea of a random forest. Random forests are a very simple modification of the idea of bagging trees. And the reason that we do these random forests is to reduce the correlation between any two different trees. So the way that we can reduce correlation is by making a simple modification.
We still bootstrap our data set multiple times. We still learn a tree on each bootstrapped data set. Except, when we decide whether we want to split a region, rather than looking at the entire bootstrap data set, looking at all of the dimensions, we're going to randomly subsample a set of dimensions, and only make each partition along one of those dimensions. So this leads to the final algorithm called random forests. As an input, we have something called 'm', which is a positive integer. This is the number of dimensions that we're going to consider, on each split. In general, 'm' is going to be less than 'd'. Often, we set 'm' to be about equal to the square root of 'd'. Then, for each value of 'b', we first generate a bootstrap sample from our data set of the same size as the data set. So we sample our data set, to construct this bootstrap sample. We then train a tree classifier, on that bootstrap data set. But now, we compute our splits as follows. First, when we decide that we want to make another split, we randomly sub–select 'm' dimensions from the original 'd' dimensions, totally at random. And we do this for each 'b'. We then, make the best split restricted to only that subset of dimensions.
So each tree is only going to...only going to consider a small subset of all of the dimensions in our data set. And then, that's the final classifier. So the difference between bagging of these trees and random forests is that when we bag trees, each tree that we learn on the bootstrap data set uses the original algorithm we discussed, whereas with the random forests, we also bag trees but each tree is now learned on only a random subset of the original 'd' dimensions. And what this is going to do is, it's going

to break down the correlations between all of our trees so that we, have more power. So what this is going to do is break down the correlations between the trees that we learn. Here's the final result of what we get with learning a random forest on this binary classification problem. What we did was– we've learned about a few hundred trees from a few hundred bootstrapped data sets. And then, each tree is going to make its prediction of either an orange or a blue class. We let each tree make its prediction, and then we take the majority vote of these few hundred trees to make the final prediction. And we could get these decision boundaries, so for example, all of the data in this region will have the majority vote of all of these trees, say orange, and all of the data in the blue shaded region will have the majority vote, say blue. But notice that we can learn, some quite complicated decision boundaries. However because our trees are splitting along the two dimensions, they tend to align. The decision boundaries tend to naturally align with the two dimensions.