Week 7
Video Transcripts

Video 1 (12:01): Boosting

Okay. So, let's refresh our memory what we discussed at the end of last lecture, the idea of bagging a classifier. And, as we're going to see, boosting is similar to this idea. So, we assume that we have labeled data where we have 'x_1' 'y_1' through 'x_n' 'y_n'. Each 'x' is in some space script 'x', and each 'y' is a corresponding label plus one or minus one depending on the class of 'x'. Then, to generate these bagged classifiers, what we do is we first sample a bootstrap data set. And, we do this capital 'B' times each bootstrapped data set is, has 'n' data points in it, so the same size as the original data set. However, to construct this data set, we sample uniformly with replacement. So for a particular sample, we'll pick 'x_i' 'y_i' with probability one over 'n', and then the next time, we'll pick it again with the same probability. So, many of these labelled data pairs are going to repeat in our bootstrapped data set and some won't appear at all in a particular bootstrapped data set. Then, for each of these script 'b's', what we do is we learn a classifier, call it 'f_b'.
And so, we now have capital 'B' classifiers where each one is learned on a specific bootstrapping of the data set. Then, we define our classification rule, so call that 'f_{bag}', the bagging of the classifiers, to be simply the sign of the sum of the individual classifiers that we learn. So, we learn capital 'B', different classifiers, each on its own bootstrapped view of the data set. We, then, evaluate for a point 'x_0', some new point, the classification on each of those, using each of those classifiers, and then we simply take a majority vote to say what the bagged classifier would say. So, in this sense, these classifiers form a committee, and they're each voting on the label. And, the majority vote is the label that the data point is going to end up being assigned. Boosting is a development of this idea. So, the motivation is that how can we take a mediocre classifier, a really mediocre classifier, so a classifier that doesn't do much better than random guessing, even, and, how can we take many of these weak classifiers. So, weak, again, means bad accuracy, but a little bit better than random guessing. And boost them so that the sum of these individual classifiers gives a very accurate and very good prediction of the label. So, here is a schematic, a high level picture. The next two slides have high level pictures of what boosting is doing. And then, we're going to go into detail to really discuss what this method is.
But, at a high level, think of bagging in this way. We're at, we have our training data set here, our set of training data. And then, for each of the classifiers that we learn, we bootstrap a new data set from the original training sample. So, we bootstrap the training data to get another data set of the same size where we have some repeats of the data and some data is not appearing in a bootstrap sample. And, we learn our classifier. And, we do this many, many times, and then we sum the results of these classifiers. So, boosting at a very high level is almost the same, but different in a very crucial way. So, what boosting does, is it first creates a weighted sample of our data set to learn a classifier. And then, given this weighted sample, it generates a new weighted sample. So, it's not pulling from the original training sample. It's pulling from the weighted sample of the previous round of classification to generate a new

data set for which we get a new classifier. And so, if we want to pull this back a little more to see what the underlying idea of boosting is, and again, we'll discuss this in more detail next. But, what we have is our training sample. We, then, generate a weighted sample from that training set. So, think of this kind of like bootstrapping. We, then, classify that weighted sample and get, now, two outputs. We get a classifier. So, this is the same as with bagging. But, we also get some number Alpha that is associated with this classifier. And, we'll see what that is in a minute. Then, based on how we do with this classifier, we get some sort of an error weight, Epsilon, and now, we feed that into creating a second subsample of, sample of our data set.

So, we, we're basically taking how we classify at a given round and then combining it to get a new weighted data set. But, the key thing is that the output is a sequence of classifiers plus an additional number Alpha. This is a real number Alpha, one, two, and three, and now, our boosted classifier is going to have almost the same form as the bagging classifier where we make our decision at round 't', now, and we weight that decision by this Alpha term that's associated with the 't' classifier. And then, we sum those up and make our prediction based on the sign of the sum. Okay. So, let's look at the actual algorithm in some detail now, and then we'll analyze it in the second half of this lecture. So, here's the AdaBoost algorithm. This is the most famous version of boosting. There are many developments of the idea. This is the basic one, and what we're going to discuss is what's called a sampling version of this AdaBoost algorithm. So, here, again, we have 'n' labeled data points in our training set, 'x_1' 'y_1' through 'x_n' 'y_n' where each 'x' is, again, in some arbitrary space and each 'y' is a binary label plus one or minus one. Then, what we're going to have is an n dimensional weight vector that's going to change with each iteration, or sometimes, they're called rounds of boosting. So, to start out, we're going to say the weight vector at the first iteration, so this subscript is saying what iteration we're at. The weight vector on the i^{th} data point is one over n.

So, this is like a probability distribution on my data, and I initialize to to be uniform, the same exact distribution that we use in bagging. But, now, for each iteration, this distribution's going to change. And so, we'll see how that happens. So, for iteration 't', a particular iteration, what we do is we sample a bootstrapped data set of size 'n' according to the distribution 'w_t'. So, on the first iteration, we sample a bootstrapped data set of size 'n' using a uniform distribution on our data. So, that's the idea from bagging. However, at iteration 't', this distribution on the data might be different. And so, whatever the distribution on our data set is at iteration 't', we sample 'n' times from that to construct a bootstrapped sample 'b*t*'. Then, we learn a classifier at iteration 't' using the data in 'b_t'.

So, this is exactly the same thing as was done in bagging. The only difference is that the way that we constructed this data set was different because it's not necessarily with the uniform distribution. So, that's the key difference. Another key difference is that after we construct this classifier, before we can do to the next iteration, we need to, first, decide how do we update this weight probability distribution on our data. And, also, we have to get these Alphas that I discussed previously that are used in the classification. So, the way we do that is we have a classifier that we learn at iteration 't' using the data in 'b_t'. We, then, calculate the probability of error at iteration t according to the distribution on the data at iteration 't'. So, what that's saying is that the error at iteration 't' is going to be equal to the sum of all of the probability weights on the data that are misclassified according to the classifier that we learned only at iteration 't'. So, we learn 'f_t' from the data in 'bt'. We, then, evaluate that classifier at all end data

points in our original data set. And, if our classifier gets it wrong, then we include that probability of that observation and sum up those misclassified weights. So, notice that we're summing the weights only on the misclassified data according to the classifier that we learn at iteration 't'. Then, we construct or we calculate this term 'Alpha$_t$' to be half times the log of one minus the error divided by the error. So, why this is the case will become clear later. For now, just, it's important to realize that we take this Epsilon that we calculated, and we, then, perform this function to get Alpha for the tth iteration. Next, what we do is we update the weights for iteration 't' plus one on each of the data points, on each of the end data points.
And so, to do that, we first take the original weight at iteration 't' on the ith data point which is this term here. We, then, multiply it by a term e to the minus 'Alpha$_t$' times 'y$_i$' times the classification of the ith data point using the tth classifier. So, this is 'f$_t$'. This is a classifier we learned in this current iteration. We classify the 't', the ith data point. That gives a zero...us plus or minus one. We multiply that by the class, by 'Alpha$_t$' which we've calculated this way. And then, update 'w$_t$' minus the value of the weight at t plus one for the ith data point like this. And then, because we want to turn that into a probability weight, we simply normalize. So, we do this many times, capital 'T' times, where 'T' is arbitrary but large. And then, we get out of this 't' pairs, capital 'T' pairs of classifier and its associated weight Alpha. And then, for a new data point, the boosted classifier is equal to the sin of each individual classifier's prediction weighted by Alpha for that particular classifier. So, we sum that up, and then, again, it's like a majority vote. Except it's a majority weighted vote in this case. And then, we take the sign. That's our final classification.

Video 2 (9:31): Boosting a Decision Stump I

Okay. So let's step through this, in a toy example. So first what we have to do, is we have to decide what's the classifier that we're going to use, and we can pick a classifier that's not going to be, too good, because again we're going to take that weak classifier and boost it to make the sum of those classifiers much better. So, the classifier that we're going to use in this example, is a decision stump. And so, last time we discussed decision trees. Well, the case of a binary decision tree that only has a root node and two children is called a decision stump. So, because we're working with binary classification here, we're thinking of stumps. Okay. So we'll start out with data like this. We have 'R$_x$' in 'R$_2$'. So each of these points corresponds to the covariate of vector 'x', and then a minus sign indicates this–the class is minus one, and a plus sign indicates that, for this data point, the class is plus one. So that's–this is our data that we're working with. We start out with a uniform distribution on this, and we learn a weak classifier. So, we learn a decision stump on this data. And so, what we get when we do that in this example is a–is this classifier. So, we'll start at the root. We decide a dimension that we're going to make a decision rule based on. So, we decide–we end up using this dimension, and we put this splitting point, where we say everything to the right is class minus one, and everything to the left is class plus one. Okay! So, we make three mistakes, and we have 10 data points.
So when we calculate the error, because we start out with the uniform distribution on the data, our error is going to be '0.3', because we have one tenth, plus one tenth, plus one tenth that's '0.3'. Then

when we calculate Alpha, which is again one half times the log of '0.7', divided by '0.3', we get that Alpha is equal to '0.42'. So that's what we get, as the output of the first round of boosting. We get 'Alpha_1' equals '0.42', and we get a classifier that says declare plus one if it's to the left of the–in this dimension, and minus one if it's to the right of this point and–along the first dimension. So, that's the first classifier. Now, we weight the data. And remember that our weighting rule, the way that we weight this, is we say that 'w_1' 'i' times 'e' to the minus 'Alpha_1' 'y_i' using the first classifier 'x_i'. So, 'w_2 hat' 'i' is equal to this, and then we normalize this. So, let's look at what is this doing. The first classifier that we learned, misclassified these three points, and it correctly classified the remaining points. And so, for...for each point that we correctly classified, we have for example, minus one here, the class for this point is minus one, times our classification, which was correct, so it was also minus one, so this is a plus one. In fact, for all of the points that we correctly classified, we're going to have a plus one here, and for the three points that we misclassified, we're going to have a minus one here. And then, 'Alpha_1' because we're assuming that our classifier is always going to be better than random guessing, Alpha is going to always be greater than zero. Because we have log of the probability of getting it right, divided by the probability of getting it wrong.
That's going to be the log of something greater than one, if we assume we're better than random guessing. So, that's going to be a positive number. So, the thing to notice with this update rule, is that for each point 'i' that we classify correctly, we're going to have minus 'Alpha_1', times positive one. So we're going to have 'e' to the minus 'Alpha_1'. Because 'Alpha_1' is greater than zero that's going to be a number that's greater than– that's–that ends up being less than one, because 'e' to the minus 'Alpha_i'– 'Alpha_1' will be less than one. So we're down weighting, all of the points that we got correct. Whatever the weight was at the previous iteration, we multiply it by a number between zero and one. We don't weight it. For all the points that we got wrong, the last time around, we have a minus one here, times minus 'Alpha_i'. So that turns into 'e' to the plus Alpha–'Alpha_1'. So then, we are going to up weight those points. So, it's important to notice that the weights on the misclassified points are getting up weighted the next time around, and the weights on the correctly classified points are being down weighted. So that's what this is showing. The distribution on the data set after the first round of boosting, now has more probability on these three points because they were misclassified last time around, and less probability on these points that were correctly classified.
So what this ends up doing is, when we sample the next bootstrap data set from this, we're going to have more of these points, more of these misclassified points. So when we learn our decision stump, in the second round, we're going to focus in more on getting these points right, because they have higher weights. They appear more and so, we're going to get penalized more for getting these points wrong, which means that the second time around, this is our classifier. Again, we split on the first dimension. All of the points to the left are going to be classified one. So, notice that we get these right. All of the points to the right are classified minus one. So, we get these right. And now, we have these three errors. And so what we do, is we take the weight on these three points from the previous round, the updated weights, and we sum them up, and that weight is going to be equal to '0.21'. We then, calculate 'Alpha_2', which is just a function of this sum of the weighted errors. And we see that it's '0.65'. And so again, we update the weights. We take the points that we got correct, and we down weight them by multiplying by 'e' to the minus 'Alpha_2'. So all of these points are going to be down weighted. And then, these three

points will be up weighted because we multiply their weights by 'e' to the plus 'Alpha_2'. And so when we, then, construct the next distribution on the data set, we get something that looks like this. So we've gotten these four points correct, in the first two classifiers.
So, notice that their weight is very small. And then, we got these points, correct one time and wrong one time, and so they have bigger weights. And so now when we–finally, the third round, I'm going to go through, when we calculate the third round classifier– so this is round three classifier– we get something that looks like this. So now, these points are highly weighted. These points have very little weights. And so we really focus on getting these points correct. And so this time the weak decisions stump is going to classify along the second dimension. If it's greater than this point, classify it to be plus one, less than this point, classify it to be minus one. So we make sure we get these right. And now we have these three errors. And we have this weight–this error. When we sum up these three weights, we get '0.12'. And when we calculate 'Alpha_3' for this classifier, we get '0.92'. Okay so, that's three rounds of boosting. In practice, that goes on for thousands of rounds. But let's stop at three and see what we have. So remember the first classifier, we learned a decision stump that looked like this, the second classifier was a decision stump like this, and the third classifier was this decision stump. The Alphas were equal to '0.42', '0.65', and '0.92'. And so, we weight the output of this by these values. So for example, this region has output plus one, and so we multiply that by '0.42'. This region has minus one, so we multiply that by '0.42'. And do the same. When we actually then, multiply these and add them up, we find that we get this decision boundary. So everything in this region is going to be classified as a plus one, and everything in this region will be classified as a minus one. So by using these three weak classifiers, with appropriate weights, we actually can get a nonlinear decision boundary. And as we add more and more classifiers, we can get more and more complex decision boundaries.

Video 3 (9:07): Boosting a Decision Stump II

And now let's look at an example—a few examples on a few slides before we analyze what it's doing, mathematically. So here's an example, of a testing error on a much more complicated, bigger data set than what we've been looking at, in the toy problem, previously. And so, without going into details on the data set, here's what we get as an output. So, random guessing for this case would be '0.5'. We did a random guessing classifier, then we're going to be '50-50'. Using a single decision stump, so if we just use one classifier, one decision stump to classify, we get a little bit better than random guessing. We get '45.8' percent error. So it's a little better than random guessing. If we learn a complicated tree, decision tree on this classifier—on this data set, we learn a '244' node tree to do binary classification. What we end up getting is '24.7' percent error on the testing set. So that's our performance, learning a complex tree. But now if we take these decision stumps and we boost them, like we've been discussing in the last few slides, here's a—as a function of the number of decision trees that we learn, here's our testing performance. So you see that we start out, we're as bad as the single stump, because our first classifier is just a decision stump. But as we add more and more of these decision stumps, we get better and better performance, until we hit a '5.8' percent error. So this is the idea of boosting a weak classifier. This weak classifier that alone, gets '45.8' percent error, when boosted, can achieve much better accuracy than a more complicated non-boosted model. Okay, so that was an example of boosting on one data set.

Let's look at an example on many data sets. Again, without going into details about what the data sets are, each of these points corresponds to one data set. And the—in this plot, what we show is the error rate, using decision stump on that data set versus boosting the decision stumps. So if a point is above this line in this triangle here, that means that boosting improves the results. So for example, in this data set, boosting had roughly '30' percent error, whereas the decision stump without boosting had roughly '90' percent error. So we see that, in all cases, boosting has improved the performance over the weak classifier, significantly. So, that's a decision stump. It's a very weak classifier. Notice that the decision stumps, alone, are getting very bad performance. Here we show boosting of a much more complicated model, called 'C''4.5', which is just the name of a specific decision tree model that we discussed, like what we've discussed last time. And again we see that boosting these more complex models, can improve performance. So for example, without boosting, we're at about '32' percent error, whereas here we're at about '23' percent error with boosting. But notice that the improvement is not as dramatic as it is in the case of a weaker classifier. So in this case, this classifier is already doing decent enough, and so boosting it can improve that but, it's not going to be as dramatic as the improvement we get, boosting a very weak classifier.

So another question would be, how does boosting a weak classifier compare to not boosting a much better classifier? So let's look at boosting a decision stump, which is just a tree with a root node and two children, versus this very complex decision tree with many nodes, many paths, you know, many leaves. So if we have—and again, each of these points represents one data set. And so, for example, in this data set here, boosting decision stumps had a '17' percent error, whereas the better classifier had only a two or three percent error. So boosting did not help in this case, with a bad classifier over the individual good classifier. But overall, notice that most of the data points...most of the points fall above this line, which means that boosting a weak classifier gave better performance, than not boosting a very good classifier. And finally, we could ask, "Okay, what if we boosted a good classifier versus boosting a bad classifier?" The obvious—there's really not an obvious advantage, as you can see, because many of the points, they're right along this line here. Some are below, some are above, but overall, it's not clear that boosting a decision stump is, in general, worse than boosting a very good decision tree. And so the reason why that would be significant, and why that's this is a good result that we are happy to see, is that learning a decision stump is extremely fast.

It's extremely fast, whereas learning this classifier 'C''4.5' is much slower, much more involved, requires a more complicated algorithm. So, if we have to learn a thousand of these. If we're going to boost a thousand rounds and learn a thousand of these classifiers, we want to learn a thousand very fast, you know, classifiers very quickly, versus a thousand classifiers that take a long time to learn for each one. So what makes boosting work so well? This is a very well-studied question. There's lots of theory about boosting. We're only going to be able to talk about a very tiny bit of it. Before we get into the main theorem about boosting though, let's answer the question, in terms of things that we've already learned in this course. So we can show that, boosting is like learning a high-dimensional feature mapping. So, how we see that is by looking at the boosting decision. Remember, if we have an input 'x_0', and we want to predict the class label for that 'x_0', using a boosting classifier, what we do is we evaluate the classifier

for the tth iteration of boosting. So we get a classification for classifier 'f_t'. We weight that classification by 'Alpha$_t$'. So, this is a plus or minus one. We weight that by 'Alpha$_t$'. We then sum it up over all of the capital 'T' classifiers, and then we predict the label of 'x_0' to be whatever the sign is of the result. So now let's, define Phi of 'x'. So we input an 'x'. We have a feature mapping— what we can think of as a feature mapping, phi of 'x' to be the decisions of each of the 'T' classifiers. So the first classifier outputs are plus or minus one, up to the Tth classifier, which also outputs a plus or minus one, where 'T' is going to be a large number, maybe a thousand or even more. So this is a very high-dimensional vector. Similarly, we can think of a weight— a vector Alpha as containing each of these individual Alphas for—that's associated with each individual weak classifier.
So, take these 'Alpha$_1$' through 'Alpha$_T$', put it into a vector, call that Alpha. Then simply using that change of notation, we can say that this classifier is equal to the sign of the dot product between this high-dimensional feature mapping and a weight vector Alpha. So we can see that boosting is trying to, in an online way, in an adaptive and online method, learn a high-dimensional feature mapping along with the decision—the linear decision boundary defined by Alpha, such that we can classify all of the data, in our training set. So intuitively, that's how we can think of boosting as working as a feature mapping method. Now I said something there, that I have to prove, that it's trying to find a linear classifier that perfectly classifies our training set. In a minute, we're going to actually going to prove that that's what it's doing.

Video 4 (4:51): Application: Face Detection

The face detection problem, as you can imagine, is–we have an image or we have video, and we want to find all the faces in the video or the image, by putting some sort of a box around the face. That's going to define the face in the image or the multiple faces that appear in the image. As you can imagine, this is a complex, difficult problem. But, boosting allows us to not have to try to perfectly define a complicated classifier. We can define a classifier that does okay, and then boost it so that we get excellent performance. So, before we can discuss boosting, we're going– we have to discuss what are the features that we extract. And then, I can tell you right away that once we extract these features, we simply run it through a booster classifier to learn–to boost decision stumps, and then that's what we've already discussed. So, what are the features that we're going to extract from images? So what we do with this problem, is we take an image– and video is just a sequence–going to be viewed as a sequence of images. So we take an image, and we divide it into patches of different scales. So for example, we take '24' by '24', '48' by '48', and we do this across all parts of the image. So we first start with smaller patches of '24' by '24', and we just shift it throughout the entire image. Then we increase it, '48' by '48', and shift that. And then we're hoping that at one of those boxes, that one of those sizes, it falls perfectly on a face, and that we can then say, "I found a face." So, to find the faces in a particular image, what we do is we take boxes of different sizes, we then shift that box across the image, each box picks out a subpart of that image, and then we classify, is there a face in this '24' by '24' block, or is there not a face? And so, in order to do this, we need to extract features from a particular block within the image. And so here is, what features can be used. So for example, we have a bigger image, and then we extract, say, a '48' by

'48' block from a certain region in that image. And it happens to fall on this face, right here. So the previous region that we picked out might have been, this part right here, and the region that we pick out after this one might be this part right here. And for each of these, we want to say, "Is there a face in this image right in the middle of the image or not?" And so for this particular face, what we do is we have many filters. For example, this could be a filter...this could be a filter, and we have many of these filters, '45,000' in total. They're shifted all over the image. We simply mask the image that multiply this filter with this image.
We then, sum all of the pixels in the white region, and subtract that from the sum of all the pixels in the black region. So, we some all of the pixels in the black region, we subtract the sum of all the pixels in the white region. That's one feature. That number is one feature. Similarly, we see sum all of the pixels in these two black regions, we subtract from it the sum of the pixels in this white region. That's another feature. We take each of these little filters, and we shift it all over the image. Similar with this filter, we shift all over the image. And all these other features–filters, we shift them all over. Then, we extract '45,000' of these numbers. And now we're going to allow boosting, to filter through all of this information, and find out which combinations of dimensions, which weak classifier should I combine, in order to be able to do declare that there is, in fact, an image in the data set. And so of course, that means that we have to have a bunch of labeled data that somebody provides us that says, this is an example of a region of an image that has a face, and then, of course, many examples of regions of images that don't have a face.
So I'm not going to discuss, in more detail, the application, but you can see it works well. You can see from your camera that it works well, when you try to take a picture and it automatically locates the images– the faces in the camera. That's what it's doing. It's using boosting to find the faces.

Video 5 (28:18): Analysis of Boosting

So in this lecture, I want to discuss the main theorem of boosting. It's called the training error theorem, and it goes like this. So under the AdaBoost framework, the algorithm that we discussed earlier, if we let 'Epsilon$_t$' be the weighted error of the classifier 'f$_t$', so 'Epsilon$_t$' is just calculated exactly like we discussed in the algorithm. And we define 'f$_{boost}$' to be the boosted classifier after 't' rounds. So we have this classification rule, where we take the tth classifier, and weight it by 'Alpha$_t$', and then sum those up, and declare the sign. Then we can make the following statement about the training error, the error on the training data. So specifically, what is the training error? So the training error is we simply take the boosted classifier, we evaluate it on all points in our data set, and then we sum up the total number of points that we misclassify within our training data set, and divide by the size of the data set. That's the fraction of points in our data set that we misclassify, using a boosted classifier, where we use 'T' rounds...'T' ...'T' weak classifiers to construct this boosted classifier.
So the training error, is going to be less than this term here, where we have 'e' to the minus two. And then this term is the sum of one half minus 'Epsilon$_t$' squared, over each round of boosting. So for every round of boosting, we get an 'Epsilon$_t$'. We say—we take one minus 'Epsilon$_t$'. We square it. We sum it up. We sum that value up for all rounds of boosting. Then our training error, has to be less than or equal

to this number here. Now remember our assumption about the weak classifier. We said it was going to be a little bit better than random guessing. So each of these 'Epsilon$_t$'s', even though it's going to be large, we're going to get a lot wrong, is going to be a little better than random guessing. And so we have a small number here. If we then, add up, many of those small numbers, say thousand or several thousand of these small numbers, we end up with a very big number here. And so this term is going to zero. For example, let's define 'Epsilon$_t$' to be '0.45'. We don't get to define 'Epsilon$_t$', of course. But if— but just to see what type of a bound we're getting here, if we let 'Epsilon$_t$' be '0.45', and we say we're going to do 'T' rounds of boosting, then what that would say is that the training error has to be less than '0.0067'. So that's very small. Most likely for a moderate site data set that means that our training error is equal to zero, already. So this is the theorem that shows how a bunch of these weak classifiers that don't do very well on their own, when combined, pushes the training error down to zero, as the number of these classifiers that we learn increases to infinity. My goal for the rest of this lecture is to prove this theorem. And so the proof of this theorem follows some very simple logic. It can be broken down into three steps. And it's a simple application of the fact that if 'a' is less than 'b' and 'b' is less than 'c', then therefore, 'a' is less than 'c'. So it's a simple logic that let's say— if we wish to prove that 'a' is less than 'c', we can do it in two steps like this.
So the first step is going to be to calculate this term 'b'. So, what we call 'b' in this proof? The second step is going to show that a term 'a' is less than 'b','a' is going to be this term right here. The third step is to show that 'c' is greater than 'b'. And 'c' is going to be this term right here. So if we can show these three things—if can define 'b' and show these two inequalities, we can therefore conclude that 'a' is less than 'c'. So our goal is to find that 'b' to put in between here, such that this term is less than 'b', and this term is greater than 'b'. So let's go through the some of the terms that we're going to need. Let's remember what the AdaBoost algorithm is. Remember at iteration 't', we had a distribution on our data set. So, 'w$_t$' 'i' is the probability weight that we've assigned to the ith training point at iteration 't'. We then learned a weak classifier at iteration 't'. We learned a term Alpha for iteration 't'. And then to get the weight at iteration 't' plus one, we took the old weight and we multiplied it by this term, which remember is going to increase the weight if we got it wrong, so this term will be positive if we get it wrong according to the tth classifier, or decrease the weight if we get it right. So this will be negative, if we get it— if we classify correctly. So there's this term. And to get the probability distribution on the training data at iteration 't' plus one, we simply normalize that term. We took this term and we divided by the sum to normalize it. Okay. So for the proof, I want to define a new term 'Z$_t$'. 'Z$_t$' is going to be equal to the sum of these weights. So whatever we had to divide by to get a new probability distribution, that term is 'Z$_t$'. So our first goal is to expand—write an expanded representation of the weight on the ith point at the capital 'T' first round of boosting, and to show that it's equal to this. First, I want to show that this weight on the ith data point at round 'T' plus one is equal to one over 'n', times 'e' to the sum of the individual classifiers, so remember that this is the boosted classifier after 'T' rounds of boosting, divided by this—the product of these normalizations.
So let's walk through this and show why we can write this weight in this way. So remember that the update rule at iteration 't' plus one for the ith data point was equal to the weight on that data point at iteration 't', times 'e' to minus 'Alpha$_t$' 'y'$_i$', times the classification of 'x$_i$' for the tth classifier. So we've again upweighted or down weighted, depending on whether we got it right or wrong. And then, we

divide by what we've defined to be 'Z_t', which is just the normalizing constant. All this is whatever i need to divide by to make this turn into a probability distribution. So notice that we've written 'w_{t+1}' as a function of 'w_t'. So we can do exactly the same thing, by writing 'w_t' as a function of 'w_{t-1}'. And then we can write 'w_{t-1}' as a function of 'w_{t-2}', and we can peel this back, all the way to the first iteration. And then remember that we define 'w_1' to be the uniform distribution, so our first distribution on the data set was simply uniform, one over 'n'. So in that case we can say that the weight on the ith data point at iteration capital 'T' plus one is simply equal to the weight on the ith data point at the first iteration, which is one over 'n'—that's where this one over 'n' is coming from, times the terms that we multiplied here for each individual round of boosting, divided by the corresponding weight—the corresponding normalizing constants. And so, notice that what we have here is one over 'n', that's this term. In the denominator, we have a product from one to capital 'T' of these normalizing constants 'Z_t'.
We don't know what they are, but whatever they are, we have a product of them. And then in the numerator, notice that we have 'e'— this exponential of a term, multiplied through. And so we know that if we multiply two exponentials, we can just sum the values in the exponent and then have an exponential of that sum. So we can write this product simply as, 'e' to the sum of these terms up here, over 't'. So that's what I've written here. So the product of these terms is equal to this. And then finally, the important thing to notice is that we've simply defined this to be our boosted classifier, after iteration 'T'. So this term is our final classifier, right here. So what we're going to do is we're going to let this product of these normalizing constants be what I call 'b'. The interesting and important thing to know is we're never going to actually be able to say what this is. We don't know what the value of it is. But it has some value, and we're going to call it 'b'. And then all we care about is how 'a' and 'c' relate to that value 'b'. Okay, so that's step one. Now let's move onto step two, where we want to find an 'a' that's less than 'b'. In particular, we want 'a' to be the training error. So our goal is to show that the empirical training error after 'T' steps of boosting using the boosting classifier after 'T' steps is less than or equal to the product of these normalizing constants. From the previous slide, we saw how we could write the weight at round 'T' plus one in this way. And so we can also say that the weight, times the product of these normalizing constants is equal to one over 'n' times 'e' to the minus 'y_i', times the classification that the boosted classifier makes on point 'x_i'.
So this is just an important fact to note. Again, we don't actually know what this value is, but whatever it is, we have this equality. That's the key thing is this equality. Okay. So using this fact, let's go ahead and derive step two. In order to do this, I just want to make it more transparent, what exactly we're doing in this proof. So if we have a value 'z_1' that's negative, is less than one—zero. We have that zero is still greater—is still less than 'e' to the 'z_1'. So 'e' to the minus something is still greater than one. And also, if 'z_2' is greater than zero, then 'e' to that 'z_2' has got to be greater than one. So these are just two basic facts that we're going to use. Now we want to bound this term here, which is our training error. We again sum up the total number of points that our boosted classifier misclassifies, divide by 'n', that's the fraction of our data that we misclassify using the boosted classifier. And now we just want to bound this using, these facts. So the left term is zero or one, that corresponds to whether this could be zero or one, and then the right term is 'e' to the something, which is going to correspond to this. Okay, so what exactly is this doing? So, focus on the ith data point. We're either going to get the ith data point wrong in which case this is equal to one, or we're going to get it right in which case this is equal to zero. If we get

it right, then whatever the label is, it's going to agree with the sign. The sign of the label will agree with the sign of the label that we predict. And so this will be a positive number, and we'll have 'e' to the minus one.
So for any data point 'x_i' that we correctly predict, we have 'e' to the minus one here, and we have zero here. So, zero is of course, less than 'e' to the minus one. So we can upper bound the indicator by this term, in the case that we get it correct. Now let's look at the case, where we get it wrong. If we get the i^{th} data point wrong, according to our boosted classifier— this is going to be a one because we...we get it wrong, and then whatever the label is for the i^{th} data point, the sign of it, is going to be the opposite of the sign that our classifier predicts. And so 'y_i' times 'f_{boost}' of 'x_i' is going to equal negative one, always, if we get it wrong. And so, then we have negative negative one. We have 'e' to the plus one. And so that's greater than one, trivially. And so we can also bound all of the ones that we get correct— incorrect using this same exact function. So this term, no matter what the... what the input evaluates to, is going to be less than this term. And therefore, the training error, the empirical training error is going to be less than one over 'n' times...times the sum of these—this term here. So we have this bound. But now notice, that we can write each individual element here, one over 'n' times each individual element in this way, where we take the product of these normalizing constants, times the weight at iteration 'T' plus one of the i^{th} data point. And so we replace one over 'n' times this thing, with this term here, the weight on that i^{th} data point at iteration 'T' plus one, times the normalizing constants from the first iteration up through the t^{th} iteration. Finally, this is simply a constant as far as 'i' is concerned. So we can bring this up front, and then whatever this distribution is— it doesn't matter what this distribution is, the sum of it over 'i' is got to be—has got to be equal to one. And so we can simply get rid of this sum, and we're left with the product of these normalizing constants. And so again, 'a' is the training error, the empirical training error, and 'b' is the product of these normalizing constants.
Again, we don't actually know what these are. We can't say what this is equal to without actually running it on a particular data set. But whatever it is equal to, we have just proven that it's got to be greater than the empirical training error, up through the t^{th} iteration. So we have a capital 'T' here and a capital 'T' here, for all values of 't', this inequality will hold. So the final part of our proof, is to show that that same product of normalizing constants is less than some number 'c', which is equal to the term of interest on the right hand side of our theorem. So this is a little bit more complex. It's going to require a couple slides. So let's walk through it. So let's first look at a particular 'Z' at iteration 't'. Remember that we define 'Z' at iteration 't' to be equal to the sum of the weight on data point 'i' at iteration 't', times this additional term, which the upweights or down weights that point depending on whether our classifier at iteration 't' correctly or incorrectly predicts the i^{th} data point. So now let's split this into two sums. Let's first split this sum, over all points that we correctly classify. So if we sum over the values of 'i' that we correctly classify at iteration 't', then we know that 'y_i' times 'f' evaluated at 'x_i' has got to be equal to plus one. Because we get the right classification. So we can sum 'e' to the minus 'Alpha_t', times the weight on data point 'i' for all points that we correctly classify. And then, all points that we misclassify, which is all the other points that aren't considered here because we either correctly or incorrectly classify each point, we sum the weight on the miscorrect— classified points, times 'e' to the plus 'Alpha_t'. Because no matter what the signs are if we misclassify it, the product is going to be negative one, and then we have negative one, times negative 'Alpha_t', that's positive 'Alpha_t'.

So we can break this down into the sum of these two terms, where we sum the weights of the correctly classified points, times 'e' to the minus 'Alpha$_t$'. So remember the way that we defined 'Epsilon$_t$', is the sum of the weights of the miscorrected— misclassified points. So we have here 'Epsilon$_t$', times 'e' to the 'Alpha$_t$'. That's because we can simply pull this term out front, and sum the weights of all the misclassified points and call that 'Epsilon$_t$'. That's a definition. And then, also because these weights have to sum to one, we have one minus 'Epsilon$_t$' here, times 'e' to the negative 'Alpha$_t$'. So what we've done is simply give another way of saying what 'Z$_t$' is. It's equal to 'e' to the minus 'Alpha$_t$', times one minus the empirical misclassification, plus 'e' to the 'Alpha$_t$', times the misclassification error that we get at iteration 't'. Okay, so from step two, we've already proved this, right! We've already proved that the training error is less than the product of these 'Z$_t$'s'. And we've given a way of writing 'Z$_t$' as a function of 'Alpha$_t$' and 'Epsilon$_t$'. 'Epsilon*t*' is determined already. It's defined to be the sum of the misclassified weights, sum of the weights on misclassified data. 'Alpha$_t$' at this point has not been defined yet. We can set 'Alpha$_t$' to what we want. And so this shows, why we set 'Alpha$_t$', how we do in the AdaBoost algorithm. Because our goal is to minimize the training error. So our goal should be to minimize the upper bound on the training error. If we want to minimize this thing, we can—we should seek to minimize this thing, here. So let's pick an 'Alpha$_t$' that minimizes this term for the tth iteration. Let's minimize 'Z$_t$' over 'Alpha$_t$'. When we do that, when we actually take derivatives and solve for 'Alpha$_t$' to minimize this thing, we find that the value of 'Alpha$_t$' that minimizes this function is equal to one half, times the log of, one minus 'Epsilon$_t$', divided by 'Epsilon$_t$'. So this is what we defined 'Alpha$_t$' to be in the AdaBoost algorithm, and this is why we define it to be that, because our goal is to minimize this term here, in order to minimize the training error. Okay. So we define 'Alpha$_t$' to be equal to this, and therefore we plug this term back into this function, to get rid of 'Alpha$_t$' and write 'Z$_t$' only as a function of 'Epsilon$_t$'. When we do that, we simplify. We find that 'Z$_t$' is equal to two, times the square root of 'Epsilon$_t$', times one minus 'Epsilon$_t$'. So this is now where we're at, with 'Z$_t$', using this particular value of 'Alpha*t*', which is what we're going to do in the AdaBoost algorithm. Finally now, we want to upper bound 'Z$_t$'. So to do this, let's first be bit clever and rewrite this thing, as this. So you can verify that this is actually equal to this. Looks like we took something simple and made it look more complicated. But that's because this is going to make it easier to upper bound the term for 'Z$_t$'. So we're going to focus on this term, and we're going to use another inequality. We're going to use the fact that one minus 'x' is less than 'e' to the negative 'x'. So this is an inequality that's very useful to know. It's also very simple. If you want a quick verification, you can just look at the plot. It's not a proof of course, but you can see that one minus 'x', which is this function is always less than or equal to 'e' to the minus 'x', which is this function here.

So we have this inequality. We showed that we can write 'Z$_t$' in this way, one minus a term, let this be 'x', raised to the one half. We can upper bound this term, this inner term by this here. So we replace this inner term with 'e' to the minus 'x' where this is 'x', and so we get 'e' to the minus four, times one half minus 'Epsilon$_t$', and then we raise that entire thing to one half. And we remember that the rules for the exponential of something raised to a power, is we simply multiply one half against this term in the exponent, and so four becomes a two. So using this, inequality, we can show that 'Z$_t$' has got to be less than or equal to this term, here. And so stepping back—just the thing to notice is we don't know what 'Epsilon$_t$' is. That's going to be different for every single algorithm— for every single data set, I should

say. Okay, so we can actually now conclude, the right-hand side of this inequality. We can find what is 'c', that is greater than what we've defined to be 'b'. So we've defined—we've shown that 'Z_t' is less than or equal to this term. We know that both sides are positive. We know that 'Z_t' is positive because it's the sum of positive numbers. And we know that 'e' to anything, the exponential of any number, is a positive number. So both of these sides are positive. Therefore, the product of these 'Z_t's can be individually upper bounded. We don't have to worry about a product of positive and negative numbers somehow switching this inequality. We're only dealing with positive numbers. Therefore, we can simply replace each 'Z_t' in this product, which is what, of course, 'b' is equal to with its individual upper bound. So the product of these normalizing constants, is less than or equal to the product of these terms here, which is equal to this. Okay, so now we can combine everything. We showed in step two that we had this term is less than this term. In step three which we've just concluded, we've shown that this term is less than this term. And therefore, we can say that the training error is less than this term. And if you go back, you'll notice that that's exactly what the theorem stated.
So, we've proven the theorem. And so, the important thing to notice about this, aside from the importance of the theorem itself, is that this is completely data set independent. We've defined an algorithm. That algorithm has given us rules for how to learn a classifier 'ft'. It's given us rules for how to classify—calculate these 'Epsilon$_t$'s' based on how the classifier does at iteration 't'. And it's given us rule for how to calculate 'Alpha$_t$'. If we follow these rules on any data set, what this proof shows is that for any of those data sets, this inequality is going to hold. So 'Epsilon$_t$' again, is not something that we know, in advance what to put in here. It's whatever the data set is. But when we do run it on a data set, we will get these 'Epsilon$_t$'s'. Also, important to notice is that this product of 'Z'— of these normalizing constants in the middle, was just a stepping stone. And that, at first we may perhaps were a little bit worried that we didn't actually know what this value is equal to. We...we couldn't actually solve for this term. But in the end, it turns out that it doesn't matter what this term is. We're just using it, to prove that the thing on the left, which is what we care about, is less than the thing on the right, which is what we care about. And whatever this term is, it can be thrown away, because we're only going to end up keeping this inequality, here. We're not going to worry about what's in the middle Okay, so we've proven the fundamental training error theorem for boosting. It's very good to know that boosting very bad, weak classifiers will actually drive the training error down to zero. So we can—proven that the combination of these really bad classifiers is actually going to do a perfect job on our training set. But we should also be worried that that might end up over-fitting our training set. So...so the fact that we're going to learn a thousand or '2,000' of these bad classifiers on a training set, should raise... raise an alert or a flag that maybe we're over-fitting.
We're doing too good on the training set and it's not going to generalize well, to new data in our testing set. So I'm not going to discuss any theory, there. But empirically, the nice thing about boosting is that, in general, this doesn't happen. So for example, here's an—here's a boosted decision stump on a real data set, where we ran a decision tree. We learned a decision tree, very complicated decision tree, it had an error of about '14' percent. On that same decision tree— I'm sorry, on that same data set using boosted decision stumps, we show the error on the training data set. So this is the testing error. Here's the training error. You see it's being driven to zero. But at the same time, the testing error is being reduced, and it doesn't seem to be getting worse. So what you would think would happen if we're over-

fitting, is that as we learn a more complex classifier, so as we go this way that essential— eventually this thing will start going up like this. But in practice, that's not, often not what is observed.

Video 6 (8:09): Unsupervised Learning

So let's step back and think about the framework of supervised learning that we've been focusing on exclusively thus far. So, we had pairs of data, an 'x', and a corresponding 'y'. Where 'x' was an input, and 'y' was an output. And we wanted to learn some sort of a function 'f' such that we can accurately predict the output as a function of the in–of the input. So we, we predict 'y_i' to be approximately equal to the function of 'x_i'. Then the reason why we learned this function is when we get new data, we get a new 'x_0'. We don't know what the corresponding 'y_0' is going to be. We don't know what the response is or what class 'x_0' belongs to. And so we use the function that we learned on the training data to predict the corresponding output 'y_0' for the input 'x_0'. We talked about a few different motivations. One of which is probabilistic motivation. So this is where we think of 'x', and 'y' as random variables with a joint distribution like this. So, this is some true underlying distribution that we don't get to observe that nature is using to generate these labeled pairs of 'x', and 'y'. And then we thought of supervised learning as trying to learn the conditional distribution of the probability of 'y' given 'x'. So there's some joint distribution, but we get this, and we get to see some pairs from this joint distribution.
But then we want to learn the conditional distribution so we can predict 'y' given 'x'. And we saw how this could be done directly or indirectly. Directly was where we directly defined this conditional distribution on the label, or the output given an input. For example, with the logistic regression; or indirectly using the Bayes rule where we could use a Bayes classifier and say that the— that we were going to predict 'y' given 'x' to be the arg max of the class conditional distribution of 'x' times a prior probability on the label. So here is where we use Bayes rule to turn the conditional distribution on 'y' given 'x' into a conditional distribution on 'x' given 'y' times a prior on 'y'. So this is a very general technique. So I mention this to give some motivation, some context for what unsupervised learning is trying to do.
We see that the Bayes classifier factorizes this joint likelihood into the class. The conditional probability of 'x' given 'y' times the probability on 'y'. Equivalently, we can use exactly the same rules of probability to say that the joint likelihood of both 'x' and 'y' is the conditional likelihood of 'y' given 'x' times a prior on 'x'. So in a sense, with supervised learning, we either focused on this term or we focused on learning both of these terms together. Now you can think of what we're going to do is to divide-develop models and define models that allow us just to define some sort of a distribution on 'x'. Or you could even think learning some class conditional distribution which we had to define. We assumed that this was a simple Multivariate Gaussian previously. But maybe there are more complex distributions that are better suited to the data we're modelling that we might want to learn. So what should be distributions on the data be? What should these distributions on 'x' be? So that's the focus of unsupervised learning in a sense. And even though I'm framing it as kind of a special case of supervised learning, where I'm almost implying that there's some underlying supervised task we want to perform. There often isn't one. Often what we really care about is understanding what information is in the data that we have. And that can

boil down to in the probabilistic sense, learning some distribution on our data that captures its underlying qualities. So that leads us to the unsupervised framework. And so this what we're going to be focusing on for most of the second half of this course. So we have now data given to us where we simply have these 'x's'. 'x_1' through 'x_n', where each 'x' is in some space. For example, 'x' could be in R^d. Now we're going to define some model either probabilistic or non-probabilistic. on 'x' only. And the goal that we are now going to have is to learn through this model the structure underlying this data set. So learn what's going on with this data set. Learn what are the statistical properties of this data set etc., etc. And it's going to be less clear what exactly the measure of quality is. With supervised learning, with clear that we were trying to predict something at the end of the day.
And so, we could use as a measure of performance how well we're able to make predictions on new data. Now, we're trying to do other things like learn, you know, the underlying characteristics of the data. Also possibly make predictions but it's going to be less clear what exactly the quality measure is going to be. Here's an overview of the second half of the course. This is going to be most of the second half, not entirely the second half. But we're going to focus on three types of unsupervised learning models. The first thing we're going to focus on in the first few lectures are clustering models. So, the goal with these types of models is to take our data, 'x_1' through 'x_n', and to partition them into groups. And the applications for this include image segmentation, data quantization, also it's often used as preprocessing step for other models. So, it's a very useful technique that has many different applications. Then, we're going to discuss matrix factorization methods. And this is where we're going to try to learn some underlying dot product representation of our data. And these are going to be useful for things like reference modeling, you know movie prediction— movie rating prediction, topic modeling, so taking text documents and learning what are the underlying themes in these documents, and so on. And then finally, the last main set of models we're going to discuss are sequential models. And so these are models where there's some sort of a sequential assumption. Either the data is sequential in nature or we're going to use some sort of a sequential way of representing the data.
So we'll see an example of how to rank objects in pair-wise comparisons. And target tracking would be another application. And as is going to be evident. The division between supervised and unsupervised is not clear cut. What sometimes can be viewed as an unsupervised technique, given a different perspective, just viewing the data from a different–in a different light will suddenly become a supervised technique. So, the division is kind of rough. But, you know the techniques are still very useful for a variety of things, so don't necessarily think that the techniques we're discussing now are going to be completely useless for supervised problems and vice versa.

Video 7 (8:18): Clustering I

Okay, so in this lecture, we're going to discuss clustering, in general, and one specific algorithm for it called K-means. So, the clustering problem is fairly simple. We're given data, 'x_1' through 'x_n', and we want to partition it into clusters, as I mentioned previously, the goal is to be–is going to be to find these clusters only given the data and some modelling assumption. And we want to learn clusters such that observations or data points that are in the same cluster are considered similar. So for example, they're

all close to each other. And data in different clusters or in different groups are considered different. They're somehow far apart. So, for example, here's an image on the right. Now, the color coding corresponds to cluster assignment. So, these are not labels anymore. We just have these two-dimensional vectors. That's all that we have in our data set. And you could look at this and say that the data naturally falls into two clusters. All of the data in this region is one cluster, and all of the data in this region is another cluster. How can we define an algorithm that's going to tell us that, that's going to learn what these clusters are and tell us for each individual data point what cluster it belongs to?
So, to do this, we're going to use a latent variable that I'm going to introduce right here before we move on to the K-means algorithm. So, we're going to introduce this latent variable 'c', and if we assume that there are 'k' clusters underlining our data set, and this is going to be just a parameter that we set. We say that we set 'k' to be some number, and I'll discuss later some ways of deciding this number. Then for the i^{th} data point 'x_i'–if 'x_i' is assigned to the cluster 'k', the k^{th} cluster, then 'c_i' is going to be equal to 'k'. So, this auxiliary variable 'c_i' is going to be an index of what cluster the observation 'x_i' belongs to. So, in this sense, we can almost think of this as, like, a supervised problem except where we don't have the labels. Now, we have clusters. Each cluster has a label. This is the label of which cluster the i^{th} data point belongs to. But we don't know in advance what these labels are, so we want to simultaneously learn the clusters and learn the labels. So, let's discuss the K-means algorithm. This is the fundamental clustering algorithm. It's also, I think, the simplest, and, in many cases, the most useful. Its–I think–probably the most widely used. So, in this problem, we again, have data 'x_1' through 'x_n'. And for now let's assume that each point is a vector in our 'D' just to make it simple. And the output of our algorithm is going to be a vector 'c' also of length 'n'. And a set of vectors Mu K-vectors–K-mean vectors. So, this vector 'c' is exactly from the previous slide where the i^{th} entry in this vector is going to indicate which cluster the i^{th} data point belongs to. So, if 'c_i' and 'c_j' are both equal to 'k', then that says that the point 'x_i' and 'x_j' are clustered together in cluster 'k'. Then, Mu is going to be a set of 'k', capital 'K', 'D' dimensional vectors. Each 'Mu_K' is an R^d. This is the same exact space as 'x', so these vectors live in the same spaces as the data. And each of these 'Mu_K's' are going to be called centroids, and they're going to define the center of a cluster. So, 'Muk' will define the center of the k^{th} cluster, and then the data assigned to 'Mu_K' are somehow going to be near that point–that centroid.
So, our goal now is to learn these two sets, the vector 'c' and the set of centroids Mu. And so in order to do this, we need to define an objective function. We need to have a mathematical way of telling us what values for 'c', and Mu are better than what other values and a way for optimizing and finding a local optimal–locally optimal setting for 'c', and Mu. And we're going to pick one that's going to be easy to optimize, as we're going to see with K-means. Okay, so let's get right into it and look at the objective function. So, this is what we're trying to minimize over the vector 'c' and the set of vector Mu. And it's a sum over all of our data. So, from 'i' equals one to 'n', where, for each data point, for each value of 'i', we sum over each of the clusters a...an indicator that the i^{th} data point is assigned or– whether it's assigned or not to the k^{th} cluster. So, this will be equal to zero for 'k' minus one values, and it's going to equal one for one value. And it's simply going to be an indicator saying, "Does 'c_i' equal 'k'?" And then that indicator is multiplied against the squared Euclidean distance of the i^{th} data point to the k^{th} cluster. So, we calculate the i^{th}–the distance of the i^{th} data point to the k^{th} cluster for all values of 'k'. We square it, but then, this indicator–through this sum, this indicator is only going to pick out the square distance

to the cluster that the i^{th} point is actually assigned to. So, it'll be zero times the distance for 'k' minus one clusters and one times the square distance for the cluster that the i^{th} point is actually assigned to.
So, another way to see this is by taking this objective function that we want to minimize over each 'c_i' and over each 'Mu_K' and writing this way. For particular assignment, vector 'c_i', we sum, for each value 'k', the total sum of squared...squared distances of all of the data assigned to the k^{th} clusters. So, here we're summing over all values 'i' such that the i^{th} data point is assigned to the k^{th} cluster, the squared distance of that i^{th} data point to the k^{th} cluster centroid. And then we sum this for the k^{th} centroid, get this sum, and then sum it over all of the centroid.
So, this is the total sum of squared errors of these signs. This is what we're trying to minimize. We're trying to find the centroids, and find these assignments such that all the data assigned to the same cluster are close to the cluster centroid. And we don't care about the distance to any of the other centroids, only to the one that it's assigned to. So, this objective is not a convex objective function. What this means is that we can say we want to find the optimal values for 'c' and Mu that minimize this, but we can't actually do it. We can only derive an algorithm to minimize this locally, meaning we can't take the derivative of this thing and find the perfect values for Mu, and 'c', but we can derive an algorithm that is going to continually improve this objective function and continually minimize it until it converges to a local minimum.

Video 8 (16:21): Clustering II

Okay, so to minimize the K-means objective function, we need to have some sort of an iterative algorithm, because we can't take the derivative, with respect to Mu, and 'c', and set both of them to zero, and solve. Also notice that, our algorithm can't involve gradient methods, because even though Mu is a real value vector, 'c' is a discrete valued variable, that we want to set and can take one of 'K' values. And so it doesn't make sense to try to take the derivative with respect to 'c'. So we're not...we're also not going to be able to derive a gradient method, to minimize this thing, similar to how we did, with for example, logistic regression. So let's look at this coordinate descent algorithm. It's fairly straightforward, it's not more complex, than what we've already discussed with gradient methods. We're going to discuss a specific example of coordinate descent, for K-means, so let's look at the K-means objective. Again, we want to minimize this thing over all the values of 'c_1' to 'c_n' and over the vectors 'Mu_1' to 'Mu_k'. And so, instead of taking the gradients of this thing with respect to all the variables, which of course we can't because 'c' is discrete. And instead of trying to somehow, minimize it over everything at once, what we're going to do, is we're going to split these model variables into two sets, two different sets. One set is going to be all the vectors Mu, and the other set will be all of the indicators, or the cluster assignment 'c'. So we have these two sets, that we're going to break our variables in to, and they fall naturally into the two key variables, in the model. And even though we can't find the minimum of this thing, with respect to Mu, and 'c' at the same time, we're going to see that, if we, for example, we held Mu fixed we could minimize this thing over 'c' exactly.
So for a particular setting of Mu, we can find the best setting of 'c' exactly. And also, for a particular setting of 'c', we can find the best Mu exactly. So if we, hold 'c' fixed we can minimize this thing, over Mu

very easily. And if we hold Mu fixed, we can minimize this thing, over 'c' very easily. And so we're going to derive an algorithm that's like that, where we hold one fixed, optimize over another set of variables, then holding that other set fixed, optimize over the first set and iterate back and forth. So this is called coordinate descent. These are the two coordinates, and holding one coordinate fixed we descend in the other coordinate. Okay, so here's the outline of coordinate descent. Again, it's very general, but I'm just going to present it now in the context of K-means. So we're going to input 'x_1' to 'x_n' where, each 'x' is in R^d. We're going to randomly initialize our centroids, that's one initialization. There are others that we could choose, but just imagine that, we randomly initialized the cluster centroids. And then, the coordinate descent algorithm for optimizing K-means, iterates between these two following steps. First, in one iteration we first hold Mu fixed, and then for each value of 'c_i' we find the optimal value for each 'c_i'. So we minimize the objective, over each 'c_i' separately, given Mu. Then the second step is to, take those optimized 'c_i's, so hold 'c' fixed and now find the best vectors 'Mu_k' for 'K' equals one to cap 'K' in R^d that's going to minimize our objective, for that particular setting of 'c', and then, just keep iterating back and forth. So there's a circular way, if it's not obvious yet, why we need to iterate these two things. If you think about it, if we hold Mu fixed we can—we're going to be able to say that, we can find the best cluster assignments in the vector 'c' that, minimizes the objective function for that particular Mu. But once we change 'c', we can probably find a better setting of Mu, for that new setting of 'c'. So the Mu is updated, the 'c', is the best 'c', for the first Mu, the initial Mu but now that we've changed 'c' maybe there's a better Mu that we can find. Similarly, if we find the best, most optimal setting for Mu given a particular setting for 'c', we've now changed Mu and so there's possibly a better 'c'. So it's a circular way of thinking, that every time we optimize a variable, in the model it's only optimal for the setting of the other variables. And then, by optimizing it those other variables, perhaps now can be optimized even more. And the whole reason why, we iterate back and forth is, because Mu, and 'c', they depend on each other in the objective function. So they aren't separated from each other, the setting of 'c', is going to impact Mu and the setting of Mu will impact 'c', so when we change one we're going to impact how good the other one is. Okay! Let's look in more detail at the coordinate descent algorithm. Remember, that the first step we update 'c' given Mu, so we can call this, the assignment step. It's where we take each data point in our data set and assign it to one of the capital 'K' clusters. So for this step, we're given a setting for 'Mu_1' through 'Mu_k', and we want to update 'c_1' through 'c_n'. So the way that we update this is, we look at our objective function, that's the thing that we're trying to minimize. And for this step, we're going to write it out in this way to make a little bit more clear. So, what I've written...the way I've written this is I've expanded the sum over each data point to separate it, so here's, the sum over each cluster assignment for the first data point, through the sum over each cluster of the assignment, for the nth data point.
And we have one of these things, in between here for every data point in between. Now our goal is to minimize this thing with respect to the vector 'c'. We want to pick 'c_1' through 'c_n' to minimize it, and so we need to come up with a way to do that, And if we look at this, objective function we see that, whatever we set 'c_1' to be equal to, it has no impact at all, on what we set 'c_n' to be equal to. These are—we can view these as 'n'...and completely separate and independent problems, given the setting for 'Mu_1' through 'Mu_K'. So if we want to minimize 'c_1', we want to minimize the objective over 'c_1', we want to assign the first observation to one of the 'K' clusters. All we need to do is, focus in, on this term here,

and minimize this thing independently of all of the other terms in the objective. So really, the optimization of, for example, 'c_i', is an independent optimization, where we pick out the ith sum in this...in this sequence, and we minimize the squared distance from the ith data point, to the Kth centroid, over the subscript 'K'. So 'c_i', is simply going to be assigned to the cluster, that has the minimum squared distance here. So in plain English we're assigning 'x_i', to the centroid it's closest to. So notice that, because 'c_i' is a discrete variable it can take one of 'K' values, there are no derivatives here, however, because there are only 'K' possible settings, we simply can calculate this distance and square it, to each of the capital 'K' centroids, and then pick the index of the smallest one.
So it's literally for the ith data point, to assign it to a cluster, we literally are just searching all of the clusters and finding the one that, the ith data point is closest to. Okay, so that's the assignment step, that we've updated 'c', we've optimized the objective, over 'c' for a particular Mu, because we have minimized each individual term over each data point point-wise. Now we need to update Mu. So we'll call this the update step. Given the assignment, 'c_1' through 'c_n', we want to update the cluster centroids 'Mu_1' through 'Mu_K'. And so again, we're going to rewrite the exact same objective function in a new way. So this, objective function is identical to, the previous objective function except, we're now splitting the sum, over 'K', instead of, over 'n'. So now we sum, over each data point, in each individual term, and we do it for the first cluster, up through the capital Kth cluster. And then, sum each of those together. So we want to minimize this thing over all values of Mu. And using exactly the same argument as before, notice that, updating this from 'Mu_1' is separate and independent from whatever the other values of Mu are.
Also important to remember is that, 'c_i' can be only equal to, one value between, one and 'K'. So for a particular value of 'c_i'... for a particular 'c_i', this will be zero for 'K' minus one values, and it will be equal to one for only one of these... in only one of these terms corresponding to the cluster, that the ith data point was assigned, to in the previous slide. Okay, so we want to optimize this thing over 'Mu_K'. What we do is, we take the derivative of it with respect to 'Mu_K' and set it to zero, and we find that we can analytically solve. And what we end up getting when we minimize this thing over 'Mu_k' is that, the update for the Kth centroid, simply sums all of the data that was assigned to the Kth centroid, and then, divides by the number of data points, in the Kth centroid. So that's all this is saying, take the average of all the data, that was assigned to the Kth centroid. In other words, we take the mean you could think of it of all the data assigned to the Kth centroid, hence the name, K means cluster. So here's the algorithm all on one slide. We're given data points 'x_1' through 'xn', each point is in R^d. Our goal is to minimize this objective function, this is the K-means objective function. And so what we do is, we're first going to randomly initialize our centroids, in some way, or initialize them in some way. And then, we're going to iterate, updating 'c', and Mu until this objective function stops changing.
So what we first do in one iteration, we update each, 'c_i', by setting 'ci', to be the index of the cluster, that the point 'x_i', is closest to, in the Euclidean sense. And then, when we updated each of these cluster assignments we update, the cluster centroids by taking, the average of all of the data assigned to a particular centroid. So for the Kth centroid, we simply average, all of the data that was assigned to the Kth centroid, according to the most recent update for the vector 'c'. So let's look at a walkthrough of this. Here's our data set, each of these points, is a two-dimensional vector shown here. And now we want to cluster, these data into two clusters, meaning we want to output for each data point an index of

whether, it was in the first cluster or the second cluster. And we also want to output the two centroids, so we want to output two vectors, that are both two-dimensional, where, each vector is the center of its respective cluster. So just to make it clear, let's assign the first cluster centroid to be here, and the second centroid to be here. Now, if we run K-means, we're in the first iteration now, the first step, within the first iteration, is to assign all of the data, to the nearest centroid. So all of the data, to the right side, of this pink line, is closest to this centroid, and so all of these data points are going to be assigned to the red cluster, so the color red. And all of the data, to the left of this pink line, is closest to this centroid, so they're all going to be colored, as a blue cluster point. The next step is now, to take all of these red data points, and average them and update the centroid for the red cluster and we get this update. The average of all of these red points because there are many more points, it's more dense in this region, the average is going to be right here. And similarly, with the blue points we average all of their values, there are many more points in this region, which is much more dense and so these points drag the centroid, drag the average to right here.
So that's one iteration of K-means. And now we just keep repeating this. So we have the new centroids, we again assign each data point to is closest cluster indicated by the color. And then, we update the centroids by averaging all of the data, that was assigned to that cluster. So we average all of the red points to get an update of the red centroid, and average all the blue points to get an update of the blue centroid, and we keep repeating this by assigning, and then averaging. And eventually, we stop, the algorithm is now converged. These are, our two clusters, and what we return are these two points, these are the two centroids, say, 'Mu$_1$' and 'Mu$_2$'. And we also return for each data point, an indicator of which cluster it belonged to. So all of these red points, if this is the ith data point then, 'c$_i$' will be one for this data point. And if this is the jth data point then, 'c$_j$' will be equal to two, for that data point. So we have those two things that are returned. Now, if we looked at the objective function of K-means, if we actually plotted this function, 'L' what we'll see is something that looks like this. So we have the update of 'c' here, the blue dots are after updating 'c', the red dots are after updating Mu.
So after each red dot we've completed one iteration. So here's one, two, three, and four iterations. And we get something that looks like this. Now in practice, if we have a lot of data, maybe it'll take many iterations to converge, what we could also do is simply monitor this thing, and then stop, when it stops changing, very much.

Video 9 (8:35): Convergence of K-Means

So if we want to know why K-means converges, we can intuitively think about it this way. Every time we make an update to either 'c$_i$' or 'Mu$_k$', we're finding a new value for these parameters that decreases 'L' compared to the previous value for those parameters. So, for a particular 'c$_i$', when we update it, we have found a new value that makes 'L' less than it was previously. In that sense, 'L' is monotonically decreasing. Every update we make, by the design of the algorithm, is finding a new value of the parameter such that 'L' is less than it was previously. Also, if we look at the objective, it's bounded below by zero. So it can't be, by just by construction of the objective, if you look at, we're summing squared, squared distances, and so that sum has to be a positive number. It can't be a negative number which

means that, there's a lower bound, and so 'L' has to converge at some point, but probably not to zero. So there's more to it than that, but we won't go into more details about the proof of convergence.
But when 'c', one way we can prove that it has converged is if 'c' stops changing, then Mu is not going to change. And, if Mu is not going to change, 'c' is not going to change. So once these values stop changing, we know that we've converged. We can't improve it any more. However, again, because this function 'L' is not convex in 'c', and Mu, maybe it's convex in one of them but not in the other, but it's not convex in both 'c', and Mu together, what we've converged to is a local optimal solution not the global optimal. So in practice, what non-convexity means, when you hear non-convex, you should think that what that's telling you is that different initializations will converge to different final solutions. If you hear convex, what that means is any initialization will always converge to exactly the same solution and it will also be the best solution. Non-convex means it'll converge to different solutions and they'll only be locally optimal. In practice, what is observed however is that the quality of the solutions will be similar. They won't necessary-you won't necessarily converge to a solution that's much worse than another. That's an empirical observation. But there are no guarantees. And so if we want to feel more confident about the quality of our solution, a computationally intensive way but a way nonetheless, is to simply run the algorithm multiple times from multiple different initializations and then pick the objective with the lowest value. Because if it's the same model but with different initializations, then if you have two local optimal solutions, you can definitively say that one is better than the other if the objective is less for one than it is for the other. Another practical issue with K-means is selecting 'K'. So, we don't know in practice how many clusters we should...we should use. We can't necessarily visualize the data like we did in the previous slides. And so selecting 'K' is a tricky thing. Also important to notice is that we can't compare K-means objective functions across different settings of 'K'. For the same setting of 'K' we can compare different runs of the K-means objective function. But for different settings of 'K', we can't compare them because they're different models. We can't directly compare them. And in particular, as 'K' increases, this thing is going to always decrease. So as 'K' gets bigger and bigger, we're always going to decrease this thing more and more.
For example, if you want a quick way of thinking about it, if we set 'K' to be equal to 'n', so we set the number of clusters to be equal to the number of data points, and then we let the centroid for the K^{th} cluster be equal to the K^{th} data point, then we can make this thing equal zero. That's the optimal, it can't be less than zero. And so we've perfectly clustered all our data point by putting each data point in its own unique individual cluster. However, I don't need to convince you that that's not a desirable thing. We've kind of defeated the purpose of modeling which is to simplify the data and summarize it in some way. So, how can we learn 'K'? There's multiple methods. One thing, so a few of them are a bit hand-wavey, but one thing is we can use our advanced knowledge about the problem. So for example, if you want to split a set of tasks across 'K' different people, then you know that there are going to be 'K' different clusters that you need to set. Another thing we can do is look at the relative decrease of the objective function. And so even though as 'K' increases, 'L' is going to always be decreasing because we're always going to be able to cluster the data in a way that these squared distances are going to be less and less and less. We're always going to get clusters so that the data's closer to the centroid that it's assigned to.

Relatively speaking, however, if we think of K* as being the ideal number of clusters in our data, then-- and this is simply a heuristic argument, but then what we can think of is that as we increase 'K', for a 'K' less than K*, so K*, for example is '10' and 'K' is now increasing from three to four to five, and so on, we should see much, much more improvement in our objective function as we let 'K' increase to K*. So adding that extra cluster should really help cut this thing down quite a bit. But then once we start going past K* and we start overclustering...overclustering our data, we should see that the marginal increase-- marginal decrease in this objective is getting less and less. So, after we increase the number of clusters to be more than what is necessary according to the data, the improvement, you know the, we get diminishing returns in a way. So you could imagine it to look like this. So here's 'L', and here's 'K', and maybe here is K*. So we should see it in this regime really decreasing quickly, and then suddenly maybe it starts flattening out. So that's a heuristic thing that you can try. Another thing, and one of the reasons why 'K' means is so popular, is that it's often used as a preprocessing step as part of a larger application. So, for example, if we want to take our continuous data and we don't want to put a continuous model of it, we don't want to learn a continuous model of it, we maybe want to first discretize the data, so we take each point, 'x_i' and instead of learning a continuous distribution on that, we assign it to a cluster and now each point is represented by an index between one, and 'K'. So we've discretized the data. Then we can learn a discreet model. Often times, that 's something that we'll want to do. In that case, that discreet model, and again this is a bit hand-wavey, is being used for its own application. And so we can use whatever we're trying to use that, our larger application for, that might have its own quality measures and we can then, for multiple discretizations, multiple values of 'K', see how well we perform on some other task we're trying to do. Also there are more advanced, such as beginot parametrics. But we won't discuss that in this class.

Video 10 (6:52): Applications of K-means

So let's look at a simple application of K-means, just to get more of a feel for it. So, for example, if we wanted to do lossy data compression, how would we use K-means to do that? So this isn't necessarily what's done in the most state of the art techniques, but let's take a look at what we would get with this. So here's our original image. It's a '1024' by '1024' grayscale image. So, you can think of this as a matrix that's '1024' by '1024'. And each value in this matrix is one, two...a number between zero and '255'. Now, what we do with this is we chop this up into two by two patches. So we take it, and we chop it up into columns of width two, and rows of width two. And then we treat each two by two patch as a four-dimensional vector. So, each two by two patch now becomes a data point in R^4, and we have a bunch of these points, and that's what we're going to do K-means on. So we learned K-means on these points in our four. For example, we learned '200' clusters. We then take each two by two patch, and we replace it in the image with the centroid that it was assigned to.
So, in the original image we have these four-dimensional vectors that are vectorized versions of two by two patches that can take almost any value. There are many different unique values. Now, if we look at this image, and we look at any two by two patch, there are only '200' unique values that those patches can take corresponding to the '200' centroids that we learned, where we replace the patch with the

centroid that it was assigned to as indicated by its corresponding value of 'c'. So, what we've done here is reduced this from about one megabyte to '239' kilobytes. If we want to compress it more, maybe we take all of these four-dimensional vectors we extract and learn four clusters, and then replace those clusters...those points with their centroid, then we get something like this. So, we've degraded the quality, but we've compressed it from one megabyte to '62' kilobytes. And, again, if we looked at any two by two patch in here, now it can only take one of four unique values, all right! So lastly, I want to very quickly discuss an extension of K-means.

K-means is called K-means, because of the squared Euclidean distance measure that we're trying to minimize in the objective. But, what if we want to use a different distance measure. So that leads to something called K-medoids. And so, let's look at the algorithm for K-medoids here. So here we again input data, 'x_1' to 'x_n'. Now, it does not have to be in R^d. It can be anything, and we define a distance measure between any data point 'x,' and a centroid Mu. Previously, this was the squared Euclidean distance for K-means, now it can be anything. We take out in the objective function for K-means, the square Euclidean distance, and we replace it with this distance measure. And now, how do we optimize this thing? So, we can follow exactly the same steps previously where we iterate the following two steps, where, first, in an iteration we want to update 'c_i'. And so, we set 'c_i' to be equal to the index of the cluster that the point 'x_i' is closest to, according to this distance measure. So that's the key difference. We now use this distance measure to assign each data point to its nearest cluster, where nearness is measured by this measure. After updating each cluster assignment we then update each cluster centroid, where now we want to minimize over Mu over the centroid the sum of these distances over each data point that was assigned the cluster we're updating. So, if we're updating the k-cluster, then we update the sum of the distances using only the data that was assigned to the k-cluster, according to our updated 'c'. So this thing for the square Euclidean distance was analytic. We could take its derivative, set it to zero, and find the optimal setting for Mu using an equation. Now, according to what distance measure we use, perhaps we need to run an iterative algorithm to minimize this thing in Mu.

So in that case we would have nested iterative algorithms. In each iteration of the K-medoids algorithm, we would have to call an iterative algorithm to update each centroid 'Mu_k'. So, for example, what would we use, we saw already that with K-means we simply defined this thing to be the square Euclidean distance. However, if we wanted to make our algorithm more robust outliers, so we've seen in some previous slides how if we have an outlier in our data set, and we calculate the mean, that outlier can drag the centroid out quite a bit. If we want to define an algorithm that's going to be robust to these types of outliers, we might want to use the 'L_1' distance. So, remember, with the 'L_1' regularization, the LASSO for linear regression, we saw how this led to sparsity. So it also can be used here to make this distance measure more robust so that outliers can't drag the centroid out towards them as much. Also, if our data is not in R^d, so this would be another example where we're working in R^d, if our data is something more complex, we could define a more complex distance measure. For example, if the data is a histogram of counts of the number of times a word appears in a document, then maybe we want to define a more appropriate distance measure that would take into account the support of the data set.