Week 4
Video Transcripts

Video 1 (06:29): Classification

Okay, let's move on to the next other type of supervised learning technique we're going to discuss in this course, Classification. Okay, so the classification problem is very similar to, the regression problem. Again, as inputs, we have measurements 'x_1' through 'x_n', these are coming in to us, in some input space, and you can just think of that as being 'R^d'. And then we want to, predict the output that corresponds to the input.

So, for each 'x' coming in to us, we want to predict a corresponding 'y', except, in this case, the 'y', associated with an 'x' is from some discreet output space that assigns, 'x' just one of 'K', possible classes. So, there's no relationship between the numbers if, classes one versus two, that doesn't mean second class is one greater than the first class, there's just an indicator of a group that 'x' belongs to.

So, for example, for something called, Binary Classification, we would want to predict- we would want to assign an incoming 'x' to a class, either, plus or minus one, or plus one or zero, according to which, of the two classes that observation belongs to.

You can think of spam email, as an example. If the input is an email, we would want to categorize or classify that email as being either spam or not spam. And so, the input would be the email, and the corresponding output, the 'y', would be a label of spam or not spam. For multiclass classification, we simply have multiple classes. So, for example, 'K' class- classification problem would, assign the input to one of 'K' possible groups or classes.

So, as with the regression problem, we need to define what a classifier is. So, classification–a classifier is just a function, just like the regression problem, we defined a function. With classification, we'll define a function as well, except this function is going to map the input 'x' to a class. So, we'll take, as an input to the function, the 'x', the email, for example. And the output is going to be, a label, for the index, of which class that input belongs to. So, as with regression, this problem is two-fold.

First, we need to define what the classifier is. We need to define, what is the function that we're going to use to assign inputs to outputs. And also, we need to then, learn the actual classifier, so the classifier that we define is going to have unknown parameters and we need to then learn those parameters using labeled training data. So, we would consider pairs of labeled points in our training set labeled data, in this case.

Okay, so let's look at the simplest classifier that one could come up with, the nearest neighbor classifier. Okay, so here's an example of a nearest neighbour classification problem. So, as input, I'm showing data in 'R^2', so each of these points corresponds to an input 'x' location. And then the corresponding label for this dataset, is shown as the color assigned to the point. So, for example, this is a two- class, classification problem, all of the points that are colored blue belong to one of the two classes, call it class zero.

And all of the points that are colored red belong to the other class, call it class one. And so, now our goal is for a new input point, 'x', we have the new location, we want to assign that 'x', to one of the two classes, meaning we want to know which color should I color this dot as, red or blue. So, the nearest neighbors– the nearest neighbor classifier, does this by simply finding the point within our labeled dataset, among all of these colored points, finding the point that is closest to the new input point that we want to classify, and then assigning the class of this input to be whatever the class is of the nearest neighbor. So, for this example, the new point that we wish to classify, is closest to this point, here, in our training dataset.

This point is labelled class blue, and therefore, we would say that this new point, we will predict is going to be, also class blue. So, it's a straightforward classifier. The first question you might ask, is what distance should we use? So, in the previous slide, I used the Euclidean distance, which is just the straight line distance. Arguably, you could say, that's the default distance that you would use, however, there are many other distances, any distance, essentially, is a potential distance measure that we could use for the nearest neighbor classifier. So, for example, 'l_p' distances, the 'l_p' distance between the vector 'u' and 'v' is simply equal to this.

Maybe, we don't have vectors in 'R^d', maybe we have strings. So, may be our inputs are a string, like a word, for example. So, another distance we could use is the edit distance. Maybe our data is not in 'R^d', maybe it's a string of characters, and so the edit distance would say how many additions and subtractions of characters, substitutions, are necessary to make my input string equal to any given other string in my training set, and then find the string in the training set, that requires the fewest changes, to match my input string, and then, that would be the nearest neighbor.

A correlation distance is another one for signal processing problems, if you want to match signals, you could use a correlation distance between those two signals. So, there are many different distances that can be used, depending on the problem, for the nearest neighbor classifier.

Video 2 (04:30): Example: OCR with NN Classifier

So, let's look at a simple example, of the nearest neighbor classifier, using the Euclidean distance. So, our dataset, are twenty-eight by twenty-eight images, of handwritten digits. So, what we do is, we treat this image as a vector in 'R' seven hundred and eighty-four, because that's how many pixels are in this image. And here are examples, of what the data looks like, that we're dealing with, for each of ten possible digits.

So, this is one input, here's another, and so on. And then, with each of these data– with each of these observations in our training set we'll have a label, so we'll have, an observation that says, this is an instance of a zero, and this is an instance of a seven, and we'll have many of these labeled examples, of all of the digits.

So, for example, in this problem, we have sixty thousand labeled data points, meaning we have sixty thousand different instances of a particular digit and the corresponding label of which digit it is. And we'll also have a testing set of ten thousand handwritten digits, and these are ten thousand new images, that we don't know the label for. So, for the nearest neighbor classifier, first let's think of the training error. It's a bit vacuous, but it's good to think of what that actually means.

So, if we wanted to predict, the label of one of our label data points, so if we have a label zero and we want to predict 'C', according to our nearest neighbor classifier how well would we predict that label? The answer is, we would predict it perfectly because any point in our training set is closest to itself, and therefore, it would pick its own label.

So, the training error, in the nearest neighbor classifier is always zero; we can predict, an observation's own label perfectly, always. However, if we looked at the testing data we can notice that, we don't do perfectly, because for an unlabeled data point we can only pick, one of the label data points in order to assign it a label and so we do have some error. However, we see that it's very small for this problem. So using the 'l_2' distance and the nearest neighbor classifier we see that the error rate, the probability, the fraction of these ten thousand training points that we got wrong, by simply assigning, the label to be the label of, the closest label point, is point zero three; ninety-seven percent of the time, we get it right by just pairing an unlabeled handwritten digit with the closest labeled handwritten digit, and then, assigning the label of the unknown digit to be the label of the closest digit.

So, let's look at some examples. For example, here are three– four different examples of what we get wrong. So, for example, this is an input two. We don't know it's a two, so we find the handwritten digit in our training set that's the closest to it in Euclidean sense and this is what it returns. And this is labeled as an eight, therefore, we would say, this is an eight, so we got it wrong. Similarly, this input three is closest to this, labeled five, therefore we assign the label five, to the input three, and we get it wrong. So, these are some examples of what we end up getting wrong.

So, one observation, and this is, for this particular dataset is that if instead of picking the closest-labeled training point and assigning the label based on the nearest neighbor what if we looked, for example, at the three closest neighbors? So, for this test point of a two, if we look at, the three closets neighbors, the closest is this eight from the original nearest neighbor problem, but the next two closest are twos as well. So, if we looked at the three closest neighbors, and then, assigned the label based on a vote, in that case, we would have gotten this two right, because two of the training sets would have voted to label this a two, and one would have voted to label it an eight.

Video 3 (04:30): k-nearest Neighbors Classifier

So, this leads to the, straightforward extension of the nearest neighbor classifier to the 'k', nearest classifier. So, with this, we're given 'n', label data points. So, we have 'n' instances of an input 'x', and its associated label 'y', So, then, for a new input 'x', what we do is we find the closest labeled data points according to whatever distance measure. We return the 'k', closest ones. And so, let's let those be indexed in this way. So, the index of the closest labeled point is, 'i_1', and the index of the k^{th} closest labeled data point is, 'i_k'. And then, label the new 'x', by taking the majority vote of the corresponding labels of these, labeled examples 'k', nearest neighbors.

And then, we can break ties in either of these steps, arbitrarily. So, here's an example of the performance of 'k' and 'n' on the same exact data set, that I discussed, previously, where, we have 'k' equal to one, three, five, seven, nine. So, this case, we reduce to the nearest neighbor classifier. Three is where we look, at the three nearest neighbors and then do a vote. Five nearest neighbors, do a vote, and so on.

And so, we can see that we get a difference in performance. So, this is an example of, where we might want to do a sweep of values, and then pick the best performing value. So, we can think of the effect of 'k', as sort of a smoothing parameter. As 'k', gets bigger, we smooth out our decisions. And, what we mean by that, is that, for example, look at these two classification problems.

So, what I have here, is a three- class, classification problem. The data is in 'r2', and then, each observation which is denoted by a circle, can be one of three classes. And so, they're color-coded. This blue class, the green class, or the orange class. That's what color, the circle takes. And then, if we look at what's called a decision boundary, for the nearest neighbor classifier, that means we look at every single potential new point, and say, what would I classify this new point, and color code it.

We get a blue region, which says, that any point in this blue region is going to have its nearest neighbor be a blue class observation. And so, every–the shaded blue region corresponds to, what would be defined to be class blue. The shaded green region corresponds to the part of the space that we would assign to the green class. And, similarly, with the orange shaded region that would any point in the orange region would be assigned to the orange class.

So, the decision boundary is the point at which this changes. So, this is a decision boundary. For example, at this point, it's equally close to, the blue class and the green class. If you go, Epsilon to one side, you're in the blue class and Epsilon to the other side, you're in the green class. And so, this line, the decision boundary, is the point at which our prediction changes. And so, we can see that we get a very fragmented decision boundary with the nearest neighbor classifier, because, essentially every point is staking out its own region.

So, for example, this orange point stakes out its own region, and everything within this decision boundary is closest to the orange point. And so, will be labeled in orange class. If we look at 15 nearest neighbors, instead, then we get a much smoother decision boundary. So, for example, this orange point, or if we pick a new point, that's very close to the orange point, maybe the nearest neighbor is an orange class point, observation, but most of them will be blue.

And so, we'll classify anything in this region as blue by looking at the 15 nearest neighbors. And so, it smooths out.

Video 4 (04:21): Statistical Setting

Okay, so that's our first classifier. It's still used quite a bit. It's a, it's not a bad classifier to try out, as your first option. We saw with the digits that, with nearest neighbors, we already were predicting with seventy-nine–with ninety-seven percent accuracy. So it certainly, as simple as it is, it's definitely not something to, just disregard. But now let's look at some other classifiers. And specifically, let's start by thinking statistically.

So how can we measure the quality of a classifier? So for any classifier, we really care about two sides of the same coin. We care about the prediction accuracy, which is the probability that our classifier's going to be right. So what's the probability that, we have a classifier, that is going to predict the correct label for 'x'? Similarly, we could care about the prediction error, so we want to maximize the prediction accuracy, the prediction error, is just the probability that we get it wrong.

And so we want to minimize that. But before we can calculate these probabilities, we're making a statistical assumption, so we're going to assume, that there's some distribution 'P', underlying our data set. And what this means, is that, the labeled examples that are coming to us, under our assumption, are going to be *iid*, independent and identically distributed, according to some underlying ground truth distribution that nature provides, but we don't necessarily get to see.

So for example, when the next e-mail comes to us, or when we get the next e-mail in our data set. We're going to assume that that's just coming totally at random. And the label of, whether, it's in spam or not, is also totally at random, according to some joint distribution, on both, the e-mail and the label combined. So there's a joint distribution now on both the covariates 'x', and the label 'y'. So why do we make this assumption? So, a similar question is to ask, when is there any hope for finding an accurate classifier? Why do we think that we should be able to predict spam e-mails, and distinguish spam e-mails, from non-spam e-mails, with any accuracy? And so, the key assumption is that, by assuming that, our labeled data in our training set are coming to us, *iid*, according, to some ground truth distribution. And also that all future data that's going to come to us, is generated according to that same distribution, with the exception, that with our training set, we get to observe 'y', whereas, with our testing set, we only get to observe 'x'. We can take advantage of the statistical regularities there to assume, that things that perform well on our training set are going to generalize well to our testing set.

So this is the flowchart of that. So, past-labeled examples from 'P' come in *iid*, we pass that to some learning algorithm. Then some new, unlabelled example comes to us according to the same exact distribution, and by then predicting the label, because we have the same distribution assumptions here and here, what we learn on, from this data set will generalize well to this data set. So there's an entire theory that formalizes this, statistical learning theory, or theoretical computer science, those two areas would, formalize this.

We're simply going to make this claim that by making– by assuming that our data is coming to us *iid*, what we learn on, based on the past will generalize well to the future. And even though we didn't discuss this for regression, it's important to say that, actually, regression is also making the same assumptions.

Video 5 (10:21): Optimal Classifiers

Is it possible to talk about what an optimal classifier would look like, using this assumption of our data coming to us *iid*, from nature? The answer is 'yes', and it's going to motivate something called the base classifier. The theory, here, at a high level is only going to motivate why a base classifier might be the right thing to do, but we aren't going to then say–be able to say, that we actually found the optimal classifier.

So this is motivating a direction for how to develop a classifier, called a base classifier. So again, we'll assume that our data has come to us *iid*, according to some distribution, where we're getting labeled pairs simultaneously *iid*, so there's a joint distribution on both the covariates and its label.

So there's a joint distribution on the contents of the e-mail and the label of whether it's a spam or non-spam e-mail. And so we're going to use this distribution, even though, we don't know what it is, we can still reason about it, in the following two ways.

So first, let's discuss what is the expectation of the indicator of an event. So we need to discuss this, before we can proceed. So here are two quantities that we could, in principle, calculate, with this underlying distribution if we had access to it. So we can't actually calculate these things, but we can still reason about them. So first, what is the expectation of the indicator of an event? So what is the expectation under this unknown distribution, that we obtain a 'Y', a label, equal to one.

So we're thinking about, binary classifiers for the moment. So the expectation, under 'P', of our label being equal to one is simply the probability that the label equals one, according to this distribution. So we, integrate out 'X', and we get a probability on the label being either one or zero or some other number. The expectation of the indicator of the event is equal to the probability of the event under the distribution that you're using to calculate the expectation.

So this is a general principle. The second principle is about the tower property of conditional expectation. So conditional expectations can be viewed as random variables. So what does this mean? Let 'A' and 'B', both be random variables. Therefore, the expectation of 'A' given 'B' can be viewed as a random variable as well. So what this expectation is calculating is the expectation of 'A', so we integrate out all uncertainty of 'A', conditioned on a particular, 'B', so for a specific value of 'B', we get the expectation of 'A', and call that 'C'. However, because 'B' is now a random variable, this expectation also becomes a random variable.

So we randomly generate a 'B', plug it in here, get our expectation of 'A'. But because 'B' is random, this expectation, now, is also random. However, the expectation of this random variable gets rid of all the randomness. So the expectation of 'C', where, this is integrating over 'B', is equal to, the expectation of, the conditional expectation of 'A', given 'B'. That's simply equal to the expectation of 'A', where, 'B' now has been integrated out.

So this is a very important property of conditional expectation called the tower property. Expectations of conditional expectations, remove the conditioning, essentially. So let's use these two properties, to calculate what an optimal classifier might look like. So, of course, before we can say something is optimal, we have to say, what we're optimizing. So in this sense, we are going to say, that the optimal classifier is the classifier that minimizes the probability of making a mistake according to the assumption that our data is *iid* from some underlying distribution.

Okay, so what we're saying here is, that we want to minimize the probability that our classifier, whatever it is, does not predict the label of 'X' to be equal to, what its true label is. So remember that 'X' and 'Y' are both generated from some underlying distribution. They're independent from any other labeled pairs. So we're thinking of generating a labeled pair, generate a random 'X' and its associated label 'Y' according to, the underlying distribution.

And then look, does our classifier actually predict the label of 'X' to be equal to what the label turns out to be? Now calculate the probability of that by integrating over the distribution ' P'. So from the previous slide, we can say that the probability that our classifier will make a mistake is equal to, the expectation of the indicator, that our classifier makes a mistake, where, this expectation uses the true underlying distribution 'P', to calculate it. And then the tower property of conditional expectation, let's just say that, we can first calculate the conditional expectation of this indicator given the value of 'X'. So let's first pretend like, we know the value of 'X', and then say, for that specific value, what is the expectation of the indicator that I don't predict the label correctly? And then take the expectation of that.

So this might be a bit confusing. Why is this a random variable? So our classifier is not random. Given 'X', 'f' of 'X' is not a random variable. So our classifier will always predict the same label for a particular 'X'. However, we assume that the underlying distribution does not have a deterministic distribution on 'Y' given the 'X'. So, for example, the same e-mail, might 95% of the time, truly be a spam e-mail, but 5% of the time, that exact same e-mail, might not be spam.

So you could imagine conditions where that were the case. So in this sense, 'Y' is what's random, and that's why, this is a random variable. And now when we take the expectation of that, we get rid of the impact of 'X'. Okay, so let's expand this inner conditional expectation. So for some specific point 'X', so, for example, evaluating this conditional expectation at a specific value 'X', we can expand the expectation this way; so this right-hand side, is literally equal to, this left-hand side, where, we're taking the expectation of the indicator that we get, that our classifier gets it wrong at this particular 'X', using the distribution, so the conditional distribution of the probability of 'Y', given the 'X', that we're evaluating at.

So 'X' is fixed, but now 'Y' is random. So we evaluate this conditional expectation here. Okay, so our goal now is for every point 'X'. So point-wise, we want to minimize this function over each point, 'X'. So in order to minimize it, what that means is that our classifier is going to assign one of 'K', possible labels to every point 'X'. And so, that means that, this is going to be equal to zero, only for one value of 'Y'. So what this is really doing is saying for the label that we assign to 'X', sum up the probabilities of all of the labels, other than that, other than the assignment.

In other words, sum up the probability that we get it wrong. So if we want to minimize this equality, we can do so by assigning the label to 'X' to be the most probable label according to nature. That way, this is equal to zero, for the maximum value here, and then we're summing up the smallest, the 'K' minus one smallest probabilities, according to this distribution. So again, our Bayes classifier, this is called a Bayes classifier, is for a particular input 'X', predict the label, to be the most probable label, conditioned on 'X' according, to the true underlying distribution, given to us from nature.

So what we can show, is that, by making this prediction, that the Bayes classifier will have the smallest prediction error among all possible classifiers. So if we get our labeled data as *iid* from some underlying distribution from nature, and then for every single 'X', we predict the label, to be the most probable label, according to that distribution, then we're going to minimize our probability of making an error.

So, of course, the problem is we don't know what the underlying distribution is from nature. So we don't actually know what this distribution is. So we need to somehow approximate it. And that approximation is going to, therefore, not allow us to say, that we have the optimal classifier, anymore.

Video 6 (02:46): Bayes Classifiers

Okay, so this leads us to the Bayes classifier. So under the assumption that, our labeled data is generated *iid*, from some underlying distribution 'P', the optimal classifier, is that we should predict for a particular 'X' the label that is most probable given that 'X'. So here this is calculated, as the conditional distribution from 'P', which is a joint distribution on both 'X' and 'Y'. We marginalize, we divide by, the marginal to get the conditional distribution of 'Y' given 'X', using this distribution.

That's what we're now trying to maximize. So, for example, if we have a spam e-mail, and for that specific e-mail, nature says that, there's a ninety-five percent chance it's spam, and there's a five percent chance that it's not spam, and that's how it's going to decide whether it's spam or not spam, we should predict the most probable label. So, we should predict it to be spam. The problem is, that we don't have this probability distribution, so we somehow need to approximate it.

And so let's do that using Bayes rule. So from Bayes rule, we equivalently have that the posterior probability of the label of 'X', given the value of 'x' is proportional to the prior probability of that label times the likelihood of 'X', given the label, divided by some normalizing constant that doesn't depend on 'Y', the label, divided by the probability of 'X', where 'Y' is marginalized out. So if our goal, is just to maximize this posterior probability of the label given the, covariant 'X', then equivalently we want to maximize the likelihood of 'X' given 'Y' times the prior on, 'Y' over, 'Y'. So again, this probability distribution on 'Y', is the class prior.

It's the a priori prevalence of any of the classes. This distribution on 'X', given the class, is called the class conditional distribution of 'X'. So given that our vector 'x' comes from class 'Y', we have a class-specific distribution on 'X'. So now here's, where we get to the approximation, in general, we don't know either of these distributions, and so we're going to approximate them somehow.

Video 7 (05.26): Example: Gaussian Class Conditional Densities

So let's look at one example where, we have Gaussian class conditional densities. So in this case we're going to assume, for this specific example, we'll assume that our data, 'X', is in 'R', and that our labels are either zero or one. So we get a value from 'R', randomly, and we now want to label it to be either class zero or one. So let's pretend like we know nature, and see what this would predict. So in this specific example we're going to assume, that our data 'X', is in 'R', and the label is either zero or one, so it's a binary classification problem. And let's assume that nature generates these labeled pairs as follows, So first, it picks a class for the label according to a probability, So it will say that, the probability that, 'Y' is equal to zero is equal to 'pi$_0$', and the probability, that 'Y' is equal to one is 'pi$_1$'. And then, the class conditional density, we're going to assume is a, Univariate Gaussian. So given that our data 'X', is in class one, we generate 'X' from a Gaussian with mean and variance that's specific to class one. And if our data 'X' is class zero, we generate it from a Gaussian, with mean and variance that's specific to class zero. So now the Bayes Classifier, for this problem is going to say, to predict the label of 'X' to be the ‘arg max’ of the likelihood of 'X', given the label times the prior of the label. And so if we do this, and we see what are we going to predict, we'll see that, we'll predict the label to be one, if this value is greater than this value, and zero otherwise. So this is the likelihood term here, for 'Y' equals one.

This 'pi1' is equal to the prior prevalence of class one. And this term here, corresponds to the likelihood of 'X', given the Gaussian associated with class one. If that product is greater than the corresponding product for class zero, then it's more probable a posteriori, that the label is one, and so we'll predict it to be one. So this type of a classifier is called a Generative Model, because we have distributions on both 'X' and 'Y'. Up until now, we've been discussing distributions on 'Y', condition on 'X'. So we've never really defined a probability distribution on 'X'. In that case, we would be– we've been working with Discriminative Models. So a Discriminative Model is one in which, we want to predict some response, 'Y' or some class 'Y', conditioned on an input 'X' where, we don't make any distribution assumptions on 'X'. And a Generative Model, is where, we assume that there is some distribution on both 'X' and 'Y', and we define those distributions, and then we want to predict based on the conditional distribution of 'Y' given 'X', where, we use Bayes rule to switch that. So here's an example, where, we defined the class prevalence to be fifty-fifty, class zero or class one.

We define the Gaussian for class zero to be zero mean with unit variance. And we defined the Gaussian for class one to have mean one and the same variance. So if we look at, the decision boundary using a Bayes Classifier where, this is the true underlying distribution, we'll get something like this which is very intuitive. All it's saying, is that, if our data point that we get 'X' is greater than this point, we're going to assign it to the class one. And if it's less than this point, we'll assign it to class zero. And so, by making this decision boundary, we're simply minimizing the probability of getting it wrong, according to, these two probability distributions. So let's look at another example where, we now change the densities of the Gaussian. So here, again we have the class prevalence to be fifty-fifty, class zero or one.

The class conditional density, for class zero is the same, zero mean, you know, variance Gaussian. But now the class conditional density for class one is a Gaussian with mean one and variance one half. So we have something that, looks like this, and we want to draw a decision boundary such that we declare it to be class one if it's greater, class zero if it's less, and one that minimizes the probability we're going to make a mistake. In that case, we get something like this, but we actually have, a decision region, we don't have one decision boundary, and so we'll declare class one, if we're in this region, because that's the location where, it's more probable, and class zero, if we're in either of these two regions, because this is the most probable region according to the Bayes Classifier.

Video 8 (03.06): Example: Multivariate Gaussians

Okay. So that was an example, where our covariance are in 'R'. Let's, look at what this looks like in 'R^2'. So, again, we have data points that are coming to us in 'R^2'. We have a label zero one, according to which class it comes from. And we're going to assume that the class conditional densities are both Gaussians, with a unique mean and a unique covariance. So, the covariance for class zero is $Sigma_0$ and the covariance for class one is $Sigma_1$. And so now, if we actually look at the case where those covariances are equal, we get a decision boundary that looks like this. So what this is saying is that, the gray Gaussian corresponds to class zero. The red Gaussian corresponds to class one. If a data point comes from the gray Gaussian, it's definitely class zero. But, its point could come from anywhere, according to this distribution. If it comes from the red Gaussian, it's definitely class one. But again, it could come from anywhere according to this distribution. And so the, Bayes classifier is going to essentially give us a line that partitions the space into one half, where it's going to label everything class zero, the other half, where it labels everything class one, and that's because this line minimizes the probability of making an error. So it minimizes the probability of having something come from the red Gaussian but falling on this half, or plus something coming from the gray Gaussian but falling on this half. Now if we look at the same thing except for each Gaussian has its own, covariance, then we get this quadratic separator. So we get something like this where we have, again, the two Gaussians, but they have their own covariance. Now, the decision boundary looks like this where, everything on this side will declare the gray class, and everything on this side will declare the red class. And this decision boundary, again, minimizes the probability that we're going to get it wrong. So we pick two simple class conditional densities, just Gaussians either Univariate or Multivariate in these two examples.

But in general, a Bayes classifier can be very complicated, if the class conditional density is very complicated. So here's an example of that where, we have a class conditional density for the orange class and one for the blue class. And this is the decision boundary that we end up, finding. So this is just an example. I haven't defined the density for it, yet. So in general, these class conditional decision boundaries can become very complicated, depending on how complex the class conditional density is. So here's, an example of that where we get this type of a decision boundary, because the class conditional density that's being used is very complex.

Video 9 (11.21): Plug-In Classifiers

So we've been assuming up until now that, we know the class conditional density of each class in the Bayes classifier. But in general, we don't know that density, we have to pick it and approximate it. So even though the Bayes classifier has the smallest prediction error of all classifiers, it's conditioned on our knowing, what the class conditional densities are, which of course we don't know. We also don't know what the class priors are–a priori. All we have are the labeled examples that we assume are generated iid, according to this distribution from nature. So, if we use a Bayes classifier in practice, what we do is something called, the plug-in classifier. And that's simply where we use the available data to approximate this class prior distribution. And, we use the data to approximate these class conditional densities. So of course if we are choosing distributions here to approximate these things, we no longer can say that we have the best classifier, because we are going to pick a distribution here that is not going to be the one nature uses, it's going to approximate the density that nature uses. And so, that error and that approximation means that we no longer can claim optimality. Previously, I presented the Bayes classifier in those two Gaussian examples where I assumed that that was the true distribution and then said what would the classifier look like. Now, we're going to approximate the class conditional distribution using a Gaussian, even though we know that the class conditional distribution is not a Gaussian. So here's an example of that.

If we had a 'K' class, classification problem, where we assume that each class had its own Multivariate Gaussian from which the observations are generated. So this is now an assumption on how the data is being generated, and we now want to do maximum likelihood to learn this classifier. So in this case, again, our data is in 'R^D', our covariates are in 'R^D', and the label is a number between one to K. We estimate our class priors and our class conditional Multivariate Gaussians using maximum likelihood. And so in this case, the maximum likelihood estimate of the probability, the prior probability that our observation is going to be class 'y' is simply equal to the fraction of times that happens in our data set, so that's maximum likelihood for the class prior, just the empirical distribution of the labels in our data set. So we sum up the number of times in our data set we see class 'y' and divide by the number of observations.

Then to approximate the class conditional density, we're-because we're choosing a class conditional Multivariate Gaussian, we need to approximate the mean and the covariance of that Gaussian density. And so what we do is, we simply calculate the empirical mean of all data assigned to class 'y', that's labeled class 'y', and we calculate the empirical covariance of all of the data that's coming from class 'y'. Now, the maximum likelihood that we're doing is for a Gaussian, limited to the data for the class that we're learning. And so when we've approximated these two distributions, we then use the plug-in classifier. So for a brand new incoming 'x', we want to assign it to one of other 'K' classes. We simply pick the 'arg max' over 'y' of the prior probability of it coming from that class, times the likelihood of the observation, given that it did come from that class. So we evaluate this function in the new 'x' for each value of 'y', and then we predict it to be the max. Okay. So let's look at an example of spam filtering, where we're using a Bayes classifier. So, the input is our emails and the output is a label 'spam' or 'not spam'. So we need a way to represent the input 'email'. And so what we'll choose is for an input 'email', we'll map it to a vector 'x' of word counts. So for example, if the index 'j' corresponds to the word 'car', and the word 'car' appears three times in our e-mail, then we're going to represent that e-mail by making the j^{th} dimension equal to three. So if the j^{th} dimension of 'x' equals three, then that means that the word 'car' appeared three times in our e-mail. And then, the label is just plus or minus one, depending on whether the e-mail is spam or not. So for example, here are several words that we might be modeling, and here is an example of a spam e-mail and a non-spam e-mail. And so in the spam e-mail, we have zero occurrences of the word 'george' or 'hp', five exclamation points, two instances of the word 'remove'. So this particular spam e-mail, used the words like this. And this particular non-spam e-mail, used the words like this. So these are two examples.

This would be maybe, 'x_1', and this would be 'x_2', and then 'y_1' would be plus one because it's spam, and 'y2' would be minus one because it's not spam. So now, we want to do a Bayes classifier for this, meaning we get a new e-mail, we extract the word count vector 'x', and we want to predict that label of that email to be the maximum of the likelihood of the e-mail, given the class, times the prior of the class. So the value of 'y' that maximizes the likelihood of the e-mail, given that class 'y', times the prior of the class 'y', so. Okay. So that means that we have to define these distributions. For the class prior, we simply can use binomial distribution that–which means that we need to learn the binomial parameter. We follow the previous discussion, where we do maximum likelihood for that, and that amounts to calculating the empirical distribution of our label data. Okay. So now we need to actually define these distributions. So for the class prior, we do just as before. But, for the class conditional distribution on the word histogram 'x', given that it comes from class 'y', we can't use a

Multivariate Gaussian anymore, because it's not a reasonable assumption. So this motivates something called, Naive Bayes, which is a general approach to Bayes classification. We're going to introduce it in the context of spam filtering. So in this case what we do is, we assume that we assume that our distribution on the histogram of word counts, given the class assignment, is independent, meaning that the probability of the number of times that I see a particular word 'j', given its class, is independent to the probability that I see any other word, which means that the vector 'X' of word counts can be written as a product over the likelihood of each individual count for each particular word, given the class. So this is called, Naive, because we're breaking up any correlations in our distribution and assuming that everything is independent, given the class, which is a Naive assumption, but also one that works fairly well, which is why it's still something that you might want to try.

Okay. So now we need to estimate some parameters. So for the class prior, we approximate the distribution of the e-mail being spam or not spam, a priori as being the fraction of observations in a particular class, divided by the total number of observations in our test set. So for example, for spam, the probability of 'y' being equal to one is equal to the number of non-spam spam e-mails that we have in our data set, divided by the number of e-mails in our data set. And the class prior probability of it being not spam, so minus one is equal to the number of non-spam e-mails, divided by the number of e-mails. For the class conditional distribution, we need to now actually–we make the naive assumption that the word counts are all independent of each other. So we can write it this way. And now we pick a particular probability distribution for each specific word. So we're going to pick a Poisson–we could pick others–for this example I pick a Poisson. And so what this distribution says, is that–given that an email comes from class 'y', the number of occurrences of word 'j' in that email is going to be Poisson distributed, with parameter equal to Lambda$_j$ for the class 'y'. So each word now has a parameter associated with it for its Poisson distribution that's also class dependent. So now if we approximated this parameter using maximum likelihood, what we could show is that the maximum likelihood update for our parameter Lambda$_j^{(y)}$ is equal to the number of unique uses of word 'j' and observations from class 'y', divided by the number of observations in class 'y'. So for the spam detection problem, to summarize the class conditional distribution on a spam e-mail, what we would do is we would count for every particular word in our vocabulary, like the word 'car'. We would count how many times does the word 'car' appear in an e-mail labeled 'spam'. So, take all of our spam e-mails, count the total number of occurrences of the word 'car', that's the numerator, and then divide that by the number of e-mails in that class–of the spam class. Do that for every single word in our vocabulary, and do that for both classes, and we now have the parameter for the class-specific distribution on the word histogram.

Video 10 (09:05): Linear Classification

So in this lecture I want to discuss something called linear classification, in generality. We're going to relate it to, the Bayes classifier we discussed previously. But we'll see that it's a more general framework for doing classification. So this lecture focuses, on binary classification. So a binary classification problem looks like this.

So specifically, for this lecture, we'll assume that our covariates are in 'Rd'. And we're going to assume, that with each covariate is associated, a label, plus or minus one, depending on whether, that point comes from class plus one or class minus one. So we're assuming the plus or minus one, not the one or zero for this problem. And we're going to define a classifier, 'f' to be one, which make predictions of the label 'y_i' given an input 'x_i' and some parameters Theta.

So, in other words, the function, 'f' takes in an 'x' takes in its parameters. And then maps that to an output that's either plus one or minus one, which we hope to be accurate, in assigning the input to the correct class. Last lecture, we discussed Bayes classification framework. So in that case, Theta, here was the class prior probabilities. And also parameters for class conditional distribution on 'x'.

So in this lecture, I want to introduce linear classification in a more general framework. And then, also connect the two. So let's actually motivate linear classification, from the Bayes classifier. So with the Bayes classifier, again, we predict the class for a new 'x' to be the most probable label given the model and given training data.

So in the binary case, what we have is we predict 'x' to be a class one. If the likelihood of that 'x', given it's coming from class one times the prior of class one is greater than the likelihood of 'x', given it comes from class zero. So I used zero, last lecture, times the prior probability of it coming from class zero. So it's just the most probable label. So, we can use simple algebra to rewrite this, we can bring the right-hand side in to the denominator, of the left-hand side, and say, we would predict one, if the ratio of the left and right is greater than one.

Take the log of both sides, and we arrive at something called the, log odds. So, if the log odds, which is the log of the probability of class one, divided by, the probability of class zero is greater than zero, then it's more probable that it's class one. We declare class one. We declare class zero, if this is less than zero. So, what is the log odds actually look like for a particular Bayes classifier? So let's consider Bayes classifier, where, the class conditional density is a Multivariate Gaussian with a class specific mean, but a shared covariance, Sigma. So both classes have the covariance, Sigma.

But there's a class- specific mean. If we, then write out the log odds, if we actually plug in the values for these distributions and calculate them, we'll see that, the log odds, is equal to this right-hand term. So it's equal to something that doesn't involve 'x'. So this is a function, but notice that whatever this function equals, it's a constant. It doesn't interact with 'x' at all. So call this constant, 'w_0', plus this additional term that does depend on 'x', which is a dot product between 'x' and this matrix-vector product, which is also a vector, call that vector 'w'. So this is something that you can calculate yourself by actually plugging in these values and verifying this, that the log odds, declares, class plus one, if this is positive, class zero, if this is negative, where, we have 'w_0', plus the dot product, between 'x' and some vector.

Okay, so we can write the decision rule for this particular Bayes classifier in this way. We can think of it like this. That we declare our classifier to be equal to the sign, of the dot product, between 'x' and some vector 'w' plus some constant 'w_0'. So we calculate 'x' transpose 'w' at a constant. And then we declare the class to be the sin. So previously it was zero, one.

In this case we're back to plus or minus one. And the value for 'w', is calculated in this way. And the value for 'w_0', is calculated in this way. So our Bayes classifier, our decision rule has this typeof a form. And we saw that last time, when we plotted the decision rule for Multivariate Gaussian, a two-dimensional Gaussian problem, where, we had the shared covariance. We saw that, indeed, the Bayes classifier, had a linear decision boundary. So this is actually an example of a very general classification framework, linear classification framework.

So a linear classifier, is simply one of this form, where, we declare the class of label 'x' to be the sign of the function 'x', transposed 'w' plus some constant 'w_0'. So this would be a binary linear classifier. And the Bayes classifier with the shared covariance, that we've been discussing is one specific example, where, 'w_0' is equal to this. And 'w' is equal to that. Okay, so the form of 'w' and 'w_0', is restricted to having, to be functions of the class-specific means, the shared covariance, and the class priors.

But we might want to think of a binary linear classification, in more general terms, where, we put no restrictions on 'w', and we put no restrictions on 'w_0'. So let's now, define a binary linear classifier, in this more general framework. So binary linear classifier, is a function of this form. It takes in 'x' and it predicts the label equal to be the sign of this function of the dot product between 'x' and some regression coefficients 'w' plus some constant 'w_0', where, 'w' is any vector in 'R^d', and 'w_0', is any point in 'R'.

So this assumes that there is a property in our data set called linear separability. For this to perform well, our data has to be able to be linearly separable, in the space that 'x' lives in. So what does this linear separability mean? So two sets, two subsets of 'R^d', 'A' and 'B'. So 'A' is a set of vectors in 'R^d' and 'B' is another set of vectors in 'R^d', are linearly separable, if it's possible to find a vector 'w' and a constant 'w_0', such that, this function is always positive for any 'x' in set 'A', and always negative for any 'x' in set 'B'.

In this case, we say that those two sets are linearly separable. So why this is true, depends on our geometric interpretation, of what this function is actually doing. So this vector, 'w' paired with 'w0', defines something called an Affine Hyperplane, in 'R^d' and it's very important to understand what that means geometrically. To know why we actually are doing a linear–we are trying to linearly separate these two classes with some sort of a line or hyperplane in 'R^d'.

Video 11 (13:59): Hyperplanes

So in the next few slides, I want to step back from the classification problem, and just think about the geometry of what we're doing– and the geometry of hyperplanes. And this is going to help us understand, what it is exactly that a linear classifier is trying to do with the data. So here, I've shown an example of a hyperplane in 'R^2'.

So every vector 'w' in 'R^d', defines what can be called a hyperplane in 'R^{d-1}'. So for example, when 'w' is in 'R^2', the hyperplane is a line. When 'w' is in 'R^3', the hyperplane is like a piece of paper slicing through the space. And we can define the hyperplane. We can define this line 'H', the hyperplane associated with the vector 'w' to be the set of all points 'x', such that the dot product between 'x' and 'w' is equal to zero. So that's just saying, give me all vectors 'x' that are orthogonal to 'w'. So in this particular example, where 'w' is in 'R^d', the only points in all of 'R^d', such that the dot product is equal to zero, are the points that falls along this line here.

So the points that fall along this line, any point that we pick, has a dot product with 'w' equal to zero. In other words, all of these points are orthogonal to 'w'. So that's how we can define this hyperplane. Any point such that the dot product between that point and 'w' is equal to zero. So given this definition of a hyperplane, is it possible to pick an arbitrary point and say which side of the hyperplane it's on? And so the answer is, yes! So first, let's define the distance to the hyperplane. So if we have a point 'x', and we have a vector 'w', which defines a hyperplane– and so this is the origin right here, can we say how close 'x' is to the hyperplane.

What is this distance of 'x' to the hyperplane– the Euclidian distance of this point to this plane. To do this, we use trigonometry. The vector 'x' and the vector 'w' have a certain angle between them, and we know from trigonometry that the cosine of that angle is equal to the adjacent length, divided by the hypotenuse length. So in this case, the adjacent length is the distance from 'x' to the hyperplane because this is a 90 degree angle. And the length of the hypotenuse is equal to the length of the vector 'x'. So therefore, this length of 'x' to the hyperplane is equal to the length of the vector 'x', times the cosine of the angle between 'x' and 'w'.

And now we use the cosine rule, and so I'm not going to derive this, but there's this equation that says that the dot product between 'x' and 'w' is equal to the magnitude of the vector 'x', times the magnitude of the vector 'w', times the cosine of the angle between those two vectors. So therefore, we can say that the distance of 'x' is equal to a function of this cosine rule. So we know that the distance of 'x' to the hyperplane, is equal to the magnitude of 'x', times the cosine of their angle. And from this equation, we know that the magnitude of 'x', times the cosine of the angle, is equal to the dot product, divided by the magnitude of 'w'.

And then finally, we have to take the absolute value because in this case the cosine can take negative values, as well. And so in reality, this cosine as viewed here, is the absolute value. So, the magnitude of the vector 'x' to a hyperplane defined by 'w' is equal to the absolute value of the dot product between 'x' and the vector 'w', divided by the magnitude of the vector 'w'. Okay, so since the vector 'w' is not changing for any particular 'x' that we choose, the absolute value of the dot product between 'x' and 'w', in a sense gives a measure of the distance of 'x' to the hyperplane.

Now we need to know, which side of the hyperplane is it on? And so this returns to, what I was saying before, where we take the absolute value to get the magnitude. If we now want to know which side of the hyperplane we're on, we know that the cosine of the angle between any vector 'x' and a vector 'w'–so if I pick any vector 'x'–that's in this half of the hyperplane, then the angle between that vector and 'w' will be between minus Pi over 2 and plus Pi over 2. And the cosine of any angle in this set is positive.

Similarly, if I pick any vector in this half of the hyperplane and take the dot product–and take the cosine of the angle between any vector here with 'w', then I'm going to be– then that cosine will be negative.

So any vector, falling on this half of the hyperplane will have a cosine that's positive, and any vector on this half of the hyperplane will have an angle to 'w', such that the cosine is negative. Therefore, if the sign of the cosine of the angle between these two vectors tells us the side that 'H' is on, by the cosine rule, we can therefore say that the sign of this right hand side–these are two nonnegative numbers so they don't affect the sign where here I'm saying sign with a g– of this right-hand side is equal to the sign of the left-hand side, trivially. But the sign of the right-hand side, tells us the half

of the hyperplane that 'x' is on. And so the sign of the right-hand side, tells us whether 'x' is on the side of the hyperplane pointing in the direction of 'w', or on the side of the hyperplane pointing in the opposite direction. Equivalently, the sign of the dot product tells us as well. So in short, if I have a hyperplane and I want to know which side of the hyperplane any particular vector falls on, I take the dot product of that point with 'w', take the sign, if it's positive then I know that 'x' has to be on the side pointing in the direction of 'w', if it's negative, I know it has to be on the side pointing in the opposite direction of 'w'.

Okay, we've defined a hyperplane. Now, what is an affine hyperplane? An affine hyperplane can be thought of as just a shifted hyperplane. So we let a vector 'w' define a hyperplane in our space. We now want to shift it in some direction. So we want to shift it, either parallel in the direction of 'w', or in the opposite direction of 'w', so that, for example, we can separate our data more easily. So 'w'– the vector 'w', as discussed previously, defines the hyperplane. It's 'w_0' that then, decides the shifting of the hyperplane. Think of the affine hyperplane 'H' as being all points 'x', such that 'x' transpose 'w', plus 'w_0' is equal to zero.

Previously, 'w_0' was equal to zero, and so the hyperplane was all the points, such that they were orthogonal to 'w'. Now we've added this constant term, and we want the values of 'x', such that the sum is equal to zero. If we work out the math, again, using the geometry and the trigonometry, what we can show is that the value 'w_0' shifts the hyperplane in the direction opposite its sign, according to 'w'. So, if 'w_0' is a negative number, then we're going to shift the hyperplane a certain amount in the direction that 'w's' pointing. If 'w_0' is a positive number, then we're going to shift that hyperplane in the opposite direction that 'w' is pointing. And the magnitude of the distance that we shift it, is equal to negative 'w_0'–is equal to 'w'–the absolute value of negative 'w_0', divided by the L2 magnitude of 'w'.

So if we shift it a certain amount in the positive direction, or a certain amount in the negative direction, this value determines, how much. So if we come back to this affine hyperplane defined by all points 'x', such that 'x' transpose 'w' plus 'w_0' is equal to zero, we get something that looks like this. So in this case, 'w_0' would be a negative number, which is shifting the hyperplane in the direction that 'w' is pointing.

So let's kind of parse this. Consider any vector 'x' that's on the right of the dotted line and the left of the solid line. We know from the previous slide that the dot product of a point 'x'–say it's this point here– the dot product of this point of this vector with 'w' is going to be positive. However 'w_0' in this case is also negative. So for all points within this slab, what that's saying is that the positive amount that you add from the dot product, is not greater than the negative amount that you add from 'w_0', and so the net sum of those two values is still negative. And so every point in here will have a dot product with 'w', that is less, that is positive, but not such that when we then add 'w_0' to it, we still get a negative number.

All vectors along this line, for example, this vector– if I took the dot product of this vector with 'w' that would be equal to the negative of 'w_0'. So when I would add those two together, I would get zero. And, every single point on the left of the dotted line has a negative dot product with 'w', and then I'm adding a negative number, and so of course that is also negative. So we have an affine hyperplane. 'w_0' is shifting the hyperplane, such that now it's this line that is defining what side is positive and what side is negative.

So let's look at this with classification. Imagine that we have this data set. The vectors 'xr' and 'R^2' and there are two classes, a red class and a blue class. We want to find a vector 'w' and a scalar 'w_0', such that the sign of this function with all blue points is positive, and the sign of the function with all red points is negative. Geometrically and intuitively, what we're asking for when we find these vector– this vector 'w' and this scalar 'w_0', is we're asking for an angle of the hyperplane defined by 'w', so this–the line has to be perpendicular to the vector 'w', and we're asking for a shift of that hyperplane defined by 'w_0'.

So the vector 'w' is going to say that we're going to create a hyperplane with this angle, except it's passing through the origin. And then 'w_0' is going to shift that hyperplane in some direction such that we can separate the data. So in this case now, we have defined the vector 'w' and a value for 'w_0', such that the sign of the function on this half is negative and the sign of this function on the right half is positive. And so, we can classify all the data that we have.

Video 12 (05:15): Polynomial Generalizations

This actually generalizes to two nonlinear hyperplanes, using polynomial generalizations. In the left hand side, I'm giving an example of points 'x' that are in 'R^2'. So here 'x' is a first dimension and a second dimension. Then I have a linear classifier, and I define all points on one side to be class one, all points on the other side to be class two. So for example, perhaps class one corresponds to the positive sign and class two corresponds to the negative sign.

Now, if I perform an expansion of my features, so for example, if I take my input vector, that's two-dimensional, and I transform it so that the first two dimensions are the original vector, and then the second two dimensions, so the third and the fourth dimension are the squares of those inputs. And now I do, linear classifier- binary linear classification in 'R^4'. What I'm doing is finding a hyperplane in 'R^4' to separate these four-dimensional vectors.

But if I look at the decision boundary projected back to 'R^2', I would get something that looks like this. So the feature expansion or this four-dimensional expansion of any two-dimensional point on the right hand side would be classed–classified as class one. And that same four-dimensional vector constructed from any point on the left hand side here, would be classified as class two. Because every point to the right of this decision boundary maps to 'R^4' in a way such that it's linearly separable in 'R^4', from any point here mapped to 'R^4', according to this expansion.

And we actually saw an example of this with the Bayes classifier, where each class conditional density was a Multivariate Gaussian, and had its own mean and also its own class-specific covariance. So let's go back and look at that– similar to how we did with the linear case. So if we look at the log odds for a specific case, where the class conditional density is a Multivariate Gaussian with a mean and covariance specific to that class. And we now assume that we have two classes, class zero and class one. And we input the class-specific Gaussian density here and here for the classes, as well as the priors, the class priors, and we actually calculate this, mathematically.

What we'll find is that we get something that is a constant, that doesn't involve 'x'. So that's– corresponds to 'w_0'. We get a linear term. We get a part that is a vector– a dot product with an input 'x', and we get also this quadratic term. We get a term that is 'x' transpose, times some matrix, times 'x'. So this is quadratic discriminant analysis. So we saw this last lecture, as well. In this case we would declare class one, if the sign of a quadratic function is positive, where 'A' is defined to be equal to this and 'b' is defined to be equal to this, and this is 'c'. So we have something that likes like this.

If the sign of that function is positive, we declare class one. If it's negative, we declare class zero or minus one. What we can show is that actually this is linear in all of the unknown weights 'A', 'b', and 'c'. Similar to our discussion in regression. This is also a linear classifier because the weights that we need to learn 'A', 'b', and 'c' are all linear. They all interact linearly with our data. A quick way to see that would be to imagine this type of a transform of 'x' in 'R^2'. Let two-dimensional input 'x' be mapped to five dimensions now, where the first two dimensions are the original data.

The third dimension is two times the product between those two dimensions and then, the fourth and fifth dimension is the square of the dimensions of our input 'x'. We can see that actually what we're doing with this quadratic function is a linear classifier, in the higher-dimensional problem. And so that's where we see that learning a linear classifier in that five-dimensional space maps to a nonlinear function in the original two-dimensional space. So again, an instance where a linear problem is hiding underneath something that looks likes a very nonlinear problem.

Video 13 (04:03): Least Squares on {-1, +1}

Okay. We've discussed the geometry of the binary linear classification problem. We've shown how specific instances of a base classifier can be viewed as a linear classifier. How do we now think about defining more general linear classifiers? So what we're looking for is a more general way of learning 'w' and 'w_0', such that we can predict the class accurately of an input 'x', using this function. One simple and straightforward idea is to just, literally, treat classification as a regression problem.

So in that case what we do is we let the vector 'y' of class labels, plus or minus one, be treated as if they were– it were a regression problem. So we construct the vector 'y' of class labels, where each dimension is either plus or minus one, depending on the class of the corresponding vector 'x'. We then add a vector of ones to each dimension of 'x' and construct the matrix 'X'. Exactly as for the regression problems we've discussed, where the i^{th} row of 'x' corresponds to the i^{th} data point.

We then perform, least squares linear regression, on 'y'. And get the solution that the vector 'w' is equal to this least squares solution. And then for a new point 'x_0', we declare its label to be the sign of the dot product between 'x_0' and the weight vector that we learned, where again remember that

'w_0' is included with the vector 'w', in this case. Another straightforward option would be instead of doing least squares, doing 'l_p', linear regression.

So these are also very quick algorithms that can be done given our previous discussion. They can be viewed as quick baselines to see, you know, what's the worst that we can do. But the issue with treating binary classification as a regression problem is that it's very sensitive to outliers. So this slide gives an example of that. Here's an example where we have two classes. We have inputs that are in two 'R^2'.

So each of these points is an input 'x', and then the class is either class one in which case it's a blue circle, or a class minus one in which case it's a red 'x'. And then we try to learn least squares linear regression classifier where we treat the labels as if they were responses, so as if they were actually from a Gaussian. In this type of a problem, here's the classifier we would learn. So, look at the purple line. The blue-the green line is a different classifier. We're not discussing right now.

So the interesting line is the purple line. So in that case, using least squares linear regression actually does very well, in predicting the two classes. However if we take some of the class one data points and then we bring them out here so they can be viewed as outliers, and we do exactly the same least squares linear regression– so the responses for all of these points is plus one and the responses for all of these points is minus one, except these are now the input covariates, we see that we've overfit by dragging the line in this way.

So doing least squares linear regression on this type of a problem is going to be very sensitive to these types of outliers. We want a method that's more robust. We want a method that doesn't care so much what the values of the covariates 'x' are. We really just care what side of the hyperplane they lie on.

Video 14 (13.33): Perceptron Algorithm

Okay. So the perceptron algorithm is one where we're going to assume that things are easy. So let's assume that we have a data set that's linearly separable–so one that looks like this, where we have vectors 'x_i' where we assume that we have two-dimensional inputs. We attach a one to create a third dimension–and the original data in 'R^2' is linearly separable. So the blue class and the red class are two classes, and we can draw a line separating, those two classes. We know what that means that we can find a vector 'w', such that the sin of 'x_i' transpose 'w' equals the corresponding label 'y_i'. In this case, our classes are linearly separable, and so we can find a line that will separate them.

In fact, we can find an infinite number of possible lines. The perceptron algorithm, which was developed in 1958, first go at binary linear classification. It's going to assume that the classes are linearly separable, which is not a good thing. But it makes that assumption, and it uses a linear classifier of this form, where we predict the label of 'x' to be equal to the sin of the dot product between 'x' and 'w', where I've done the trick again of assuming that we have a dimension equal to one for 'x', to account for the offset. Okay. So the perceptron is one way of learning a linear classifier of this form, where we predict the label to be the sin of the dot product between our covariate vector 'x' and coefficient vector 'w', where again, I've assumed that the vector 'x' has a dimension equal to one, to account for the offset. And the perceptron is defined by this objective function. And we're trying to minimize this thing. It's a function of the data and of the vector 'w'. We're trying to minimize it in 'w'. So to see why we would want to minimize this thing, let's look at what this term is equal to. We've already discussed how the dot product between 'x' and 'w' is going to be equal to a positive number, if 'x' is on the positive half of the plane defined by 'w', and it's going to be a negative number, if on the negative side. 'y' is the corresponding label for 'x', and so this value is going to be positive, if we predict correctly. If we predict the label 'y_i' correctly, this dot product will be equal to a number that has the same sin as 'y_i'. And so their product, whether they're both negative or they're both positive, their product is going to be a positive number. If we get 'y_i' wrong, if for our particular vector 'w', we predict 'x_i' wrong, then the sin of this dot product is not going to equal the sin of 'y_i'. And so, the product is going to be a negative number. One of them will be positive, and one of them will be negative. And here now, what we're doing is we're summing up a number times, an indicator of getting it wrong. So really, what we're doing is summing up this value over all points that we predict incorrectly, according to some vector 'w'. And all of those points are going to be negative. And so, the sum of those incorrectly reclassified points will be negative. We put a negative sin out front to make it positive, and now we want to minimize that. So we want to minimize this number of points that we get wrong, in this sense. So now how can we learn the perceptron? We have the objective function that we're going to try to minimize.

The next problem is to try to learn the vector 'w' that minimizes that objective function. Unlike previous techniques that we've discussed, we can't find a minimum by simply taking the derivative with respect to 'w' and setting it to zero. So we can take the derivative of the objective, but then when we try to solve for 'w', such that that derivative equals zero, we quickly find that there's no analytical solution. So we need some sort of an iterative algorithm. Even though we can't solve this derivative the root of this derivative analytically, the derivative does tell us something about the objective. What it tells us is the direction in which 'w' is increasing. So the derivative of the objective function with respect to 'w', evaluated at a particular point 'w', says move in this direction if you want to increase the objective function. Therefore, if we pick a sufficiently small Eta, what we can say is that if we update the vector 'w' by moving in the opposite direction of the derivative, scaled by some tiny value Theta, we will get a new value for the vector 'w', such that our objective function at that new value is less than the objective function at the old value. This is a general method for optimizing objective functions called gradient descent. And we're going to see an example of this for the perceptron, where they do this in a stochastic way. First, I'm going to discuss the algorithm for learning the perceptron. As an input, we have a training data, where we have–so we have labeled pairs, the covariate 'x' and its label positive or negative one for 'y', and we have a positive step size Theta. The first thing we do is we initialize the vector 'w' to a vector of all zeros. And then for the each step in our algorithm, we want to update the vector 'w'. The way that the perceptron does this is, first, it searches for all examples in our data set that we get wrong according to our current–vector 'w'. So find all of the data in our training set that we predict incorrectly, given the current vector 'w', if one exists, pick one at random, and then update 'w' by taking a step in the direction of the label, times the vector 'x' scaled by eta.

Otherwise if we can't find anything we get wrong, we found a hyperplane that separates our data, and so return the vector 'w' as the solution, because it can classify our training data set perfectly. So the question is what is this doing? Here's our original objective function that we're trying to minimize. If we took the gradient of this with respect to 'w', and we define the set 'M_t' at iteration 't' to be the set of misclassified training observations, then the gradient–it would be equal to this. So I'm taking the gradient of this with respect to 'w'. I'm only summing over the points that I get wrong. And we find out that the gradient is equal to this. If we did full gradient descent, what we would do is, we would update the vector 'w' to be the old value, minus Theta times the gradient of the objective evaluated at 'w_t'. Because we want to minimize, so we have a minus. In that case we can simply plug in this value, here, for the gradient, and quickly find that what's the stochastic optimization is doing, which is what the perceptron is using. It's just randomly picking one of these values instead of summing over all of them. But in practice, we could've also summed over all of the values, and constructed the entire gradient, and plugged this in here, and our algorithm would be just as correct. Let's look at an–a simple example of this of learning the perceptron for a binary classification problem, where the data comes from 'R^2'.

Again, here are our data points. They're in 'R^2', the 'x' is in 'R^2', and the labels are color coded. So red is positive one, blue is a negative one class. And we're going to just let Theta be equal to one. Our step size is equal to one, in this picture. So for the first iteration, let's assume that this is what we start out with. Our vector 'w' points in that direction–that's our classifier, meaning that we're going to declare everything on this half to be positive one, and everything on this half to be negative one. Meaning that we get anything blue on this side, wrong, and we get anything red on the side, wrong. The perceptron will first pick a misclassified example at random, in this example, it picks this vector–randomly picks this point, because it's misclassified as being a blue class. It then, updates the coefficient vector 'w', by adding eta, times the class value, times the vector. In this case, because the class is positive one, we're adding this vector, times plus one. Okay. So when we pick this as our misclassified example and we update 'w', we're essentially adding this vector to the end of our classifier vector, and we get this update. So this is our updated classifier after our first step. We then, want to pick another misclassified example. In this case, all of these red points are misclassified and all of these blue points are–this blue point is misclassified as well as this one. So we pick a misclassified example, at random. For example, maybe we pick this misclassified point. We then, update our regression–our classifier vector 'w', by taking the old value, plus the data points that we misclassified, and get a new classifier, and then we've converged. So we're done. We've now classified everything correctly. So we've gone through an example of the perceptron, in action. There a few drawbacks of the perceptron that make it not used, so much anymore, even though it's a good first algorithm to learn. In future lectures, we're going to discuss other algorithms that fix the problem that the perceptron encountered.

One drawback of the perceptron is that if the data is linearly separable, there are an infinite number of hyperplanes that can separate the data. But the perceptron is only going to return the first one, it finds. Once it finds a hyperplane to separate the data, according to the algorithm we've discussed, there are no more changes that can be made because there are no more misclassified examples. And so it's going to return the first hyperplane, that it finds. However, not all hyperplanes are equal. Some, you know, some may be better than others. For example, we might want to define the hyperplane so that the nearest point to the hyperplane is as far away as possible. In this example–for example, this blue line separates these two classes as well as this one. However, this blue line is extremely close to this green point. Maybe we want to find a hyperplane that separates the data, but is also far away from any of the data points. The perceptron's not going to do that. Also, important is when the data is not linearly separable, which is usually the case. This perceptron algorithm's not going to converge. So it's just going to continue to keep picking misclassified examples, and updating the hyperplane, but it's never going to converge. And so it's important to define an algorithm that will actually converge in a finite amount of time, and this is also what we're going to discuss in future lectures.