COLUMBIA | ENGINEERING
EXECUTIVE EDUCATION

Week 5
Video Transcripts

Video 1 (17.41): Logistic Regression

This is a similar setup to what we've been working with so far for binary classification. We assume that we have data pairs 'x_i' and 'y_i' where each 'x_i' is in 'R^d', and it's a covariant vector associated with an observation, and 'y_i' is then a binary class which we'll assume is labeled plus one or minus one, depending on which of the two classes the ith object belongs to. A linear classifier, as we've discussed last time, takes a vector 'w' in 'R^d' as well as a scalar 'w_0' in 'R', and it predicts the label associated with a particular observation to be the sign of the dot product between the covariant vector 'x_i' in 'w' plus this bias term 'w_0'.Last time, we discussed two linear classifiers. One was least squares where we simply perform least squares directly on the labels, but we saw how that was sensitive to outliers in 'x'. And also the perceptron algorithm which was an iterative method for doing linear classification where we minimized an objective function.

However, we saw last time that that one had– that approach had convergence issues, and made assumptions about the data that weren't necessarily reasonable such that, the two classes are linearly separable by a hyperplane. We've already seen two examples in the last lecture of linear classifiers. One is least squares classifier where we regress directly on the labels. However, we saw that was sensitive to outliers in 'x', and also the perceptron which was an iterative algorithm for minimizing one particular objective function. However, we saw that that had convergence issues, and made assumptions about the data that aren't necessarily reasonable, such that, the classes are perfectly separable by hyperplane. So in this lecture we're going to see how we can extend the ideas from the perceptron, combining it with probability to fix some of the issues from the perceptron. So this lecture is going to focus on something called logistic regression which we're going to see is a fairly straightforward extension of the perceptron. It's not going to be motivated that way, but the resulting algorithm will be something that looks similar to what we had with the perceptron. So we're going to work with probability distributions in this lecture to fix some of the perceptron issues. Okay. So we previously saw an example of a linear classifier that uses probability distributions in something called linear discriminant analysis which is a particular case of a Bayes classifier where we make the modeling assumption that the label is generated from a Bernoulli distribution, and then the associated covariates are generated from a class-specific Multivariate Gaussian with mean equal to mu y. So a class-specific mean, and a shared covariant's matrix.

Then, for a particular covariate vector 'x', we would want to predict the label for that 'x', and so we would be able to then use the log odds to do that. So in this case, the log odds is the log of the joint likelihood ratio in which the– if the label of a one is more probable, this term will be positive. If the label zero or minus one is more probable, then this term would be negative. When we actually plugged in the distributions into these two ratio, and saw what was the decision? How were we making decisions? We saw that it amounted to a constant there which we could call 'w_0', plus the dot product between the vector 'x' with a vector that we could call 'w', both of which had a very specific– were a very specific function of the parameters of the model that we would have to learn or approximate in some way, for example, through maximum likelihood. So let's go back and return to the original formulation of what it is that led to our use of Bayes classifiers to begin with. If you recall, what we want to do is define a probability on a label for a particular observation given the covariates associated with that observation 'x'. So equivalent to the log odds in the Bayes classifier, we could also say that we want to predict the label one if it's more probable than the label zero given 'x'. And to see how this is equivalent to the Bayes classifier using the log odds. Simply remember from Bayes rule that the probability of the label given the covariates is equal to the probability of the covariates given the label times the probability of the label, prior probability of the label, divided by the marginal probability of the covariates. And if we were to calculate the ratio for 'y' equals one or 'y' equals zero, this denominator simply cancels, and we have exactly what we have for the log odds of the Bayes classifier. If you remember, the reason we did Bayes classification is because we didn't know how to necessarily define this probability distribution, but we could define the conditional distribution on the covariates, the class conditional distribution, and a prior. So we did more easily–we were able to more easily come up with ways of defining those probabilities. So in this lecture we're going to start from the log odds, and rather than define these probability distributions on a label conditioned on the covariate, we're going to start by defining the log odds directly. So let's motivate how we might directly define the log odds, and from that definition then be able to define a probability distribution on a label for an observation conditioned on its covariates. So, if we classify 'x' based on the log odds written this way, notice that this function 'L' is going to become more and more positive the more confident we are that 'y' is equal to plus one, simply by inspection.

The more probable that 'y' is plus one given 'x', the more positive this term is going to be. And as it gets more and more positive, we're more and more confident that 'y' is equal to plus one. The more and more negative it becomes, the more and more this ratio shrinks to zero, and therefore the log of that becomes more and more negative, the more confident we are that the label is going to be minus one. And then in the case where this log odds is equal to zero, that's like saying we could go either way, because the probability of the label being plus one given 'x' is equal to the probability of the label being minus one. So this is the log of one which is zero. So in a sense we want to directly define a function that has those properties that intuitively make sense, that becomes more positive the more confident we are when it's plus one, more negative, the more confident we are when it's minus one, and the equal to zero, the when we don't know anything. So, last time we actually saw this function with a affine hyperplane. So separating hyperplane we saw how the geometric interpretation of this function, the dot product between 'x' and vector 'w' plus some offset, can be viewed in this way where the more that 'x' become– the more 'x' moves to the opposite side –one side of the hyperplane, the more positive it becomes. The more it moves to the other side of the hyperplane, the more negative it becomes, where this direction of this vector is determining which is positive and which is negative. So the vector points in the direction of becoming more positive. So as we go more and more in the direction of the vector 'w', the more positive it becomes. As it goes more and more in the opposite direction, the more negative it becomes, and all points along this line of the separating hyperplane are equal to zero. So in this sense we already have a function that makes sense, that has the properties that we desire, and so we start from there when defining the log odds. So when we directly want to define the log odds, it makes sense to simply define it to be equal to the separating hyperplane representation that we've been discussing. So this is called the logistic link function. What we do is we–when we want to define a probability distribution on the label, given the covariates associated with the observation, so when we want to define 'p' of 'y' given 'x', instead we start from defining the log odds to be equal to this. And we already know that this has the desirable properties, that as this becomes more positive, the more confident we are that the label is on one side of the hyperplane. As it becomes more negative, the more confident that it's on the farther it is on the other side and so on. So, one–the first question to ask is what exactly is different from the previous Bayes classifier. Maybe it's an obvious answer, but recall that with the Bayes classifier, the linear discriminant analysis approach to Bayes classification, we had essentially the same exact representation, except we had a very specific formula for calculating the offset 'w_0', and for calculating the vector 'w', and it was a function of the class priors, a function of the class-specific means, and a function of the shared covariance matrix between each class. So there was a parametric form for how 'w' and 'w_0' were calculated, and because of the values that those parameters can take because of the way they're calculated, calculated those parameters through MLE, for example, there was essentially a restriction on what these two values could take, and perhaps it's a reasonable restriction, but still it's a restriction. So now the difference here between this representation and the Bayes classifier representation through linear discriminant analysis is that, we're not going to put any restrictions at all on the vector 'w' or the offset 'w_0'. So we're going to let them be anything. So, now when we want to actually calculate the probability of a label given its covariates, and given the vector 'w' and 'w_0', we can simply reverse or invert this equality by first plugging in for the probability of 'y' being minus one, one minus the probability of it being plus one. Then, solving for this probability distribution in this equation to find that we end up with this definition. So what we've done here is we've come up with a way to directly define the probability that 'x' is in class one given the parameters vector 'w' and 'w_0', it's equal to this function, which is called a sigmoid function. So if now interpret this function, we can see what's going on is this becomes much more positive. So as we're much more confident that the observation is on one side of the hyperplane, this numerator blows up to infinity, but then we have this denominator which is one plus that same number. So this whole thing is converging to one as we become more confident that it's on the plus one side of the hyperplane. As this becomes more and more negative, so as 'x' is moving more and more to the negative side of the hyperplane, as defined by 'w' and 'w_0', this numerator is going to zero, and this denominator is going to one. So this entire thing goes to zero. So as 'x' goes to more and more to the plus one side, the probability here is going more and more to one of the label being plus one. As 'x' is going to the negative side of the hyperplane, the probability of a plus one is going to zero. And when 'x' is right on the decision boundary, this is dot product is equal to zero, and so this is one divided by two, and that uncertainty of which side the hyperplane the observation is on is represented by a probability of one half of each label. So that's simply showed here. What I'm showing, what this is plotting in the red line, is the sigmoid function as a function of the argument. So as it becomes more positive, we become more confident that the label will be plus one. As the argument becomes more negative, we become more confident that the label will be minus one. And when all data points exactly on the decision boundary, we don't know, which label it's going to take, and so that's represented with a fifty chance. Okay. So now we've motivated the sigmoid function as a way to map our observation, our covariates 'x', to a probability on a corresponding label. Let's now define logistic regression. This is the logistic regression classifier. Let's define it now from the start. So as we've done previously purely for notation, so that we write less, we're going to absorb the offset. So we're going to redefine the vector 'w' to be the original vector where we attach the offset to the first dimension. So we increase it by one dimension. And we're going to define the covariate vector 'x' now to actually have a dimension of ones which are going to interact with this offset. So the dot product of 'w' with 'x' is equal to the original function. It's equal to 'w_0' plus the dot product of these two vectors. So again we have labeled data. We have pairs 'xy' where 'x' is a covariate, 'y' is its label. The labels are now going to be plus or minus one. And so this is important for the way that we derive the algorithm in this lecture, that the labels are now plus or minus one. It's more convenient to do it that way. Okay, so logistic regression then defines the model of each label conditioned on the covariate vector 'x' to be equal to this. So we assume that all the labels are generated independently of each other. So, given the covariate vector 'x', and given the coefficients 'w', the data is entirely independent, and the probability for a particular observation, the probability of the label being plus one, is simply equal to the sigmoid function which we've discussion previously, motivated previously. So notice that what we've defined here is a discriminative classifier. So this is similar to some of the regression models we discussed that were also discriminative. We have no distribution on 'x' anymore. We've only defined the distribution on 'y'. The Bayes classifier, as we've discussed previously, are generative classifiers because we first define a distribution on 'y', and then we define a class- specific distribution on 'x' given 'y'. So here again we're back to having no distribution on 'x'. We only define a distribution on 'y'. So in that sense the logistic regression model can be viewed as a discriminative classifier.

Video 2 (08.00): Logistic Regression Likelihood

All right. So, because we've defined probability distributions on our data, that implicitly defines a joint likelihood of the data. So, let's look more specifically at what that is. So, we've defined a joint likelihood on all of the labels, given all of the covariates and the regression coefficient vector 'w'. The independence assumption allows to us say that this joint likelihood is equal to the product of these individual likelihoods of each observation pair. And we're going to use the notation Sigma 'iw' just for a few slides. So, the way that we can write this probability– we know that it's a discrete probability on two events. We're going to write it this way, where we take the probability of the label being plus one. Raise that to an indicator of the label being plus one times the probability of the label being minus one raised to an indicator of that event. So this simply picks out the correct event. For the true value, we're going to have the indicator equal one. For the false value, it might for the wrong value it would be equal zero, In which case, for example, if the label is equal to plus one, then this picks out Sigma 'i', whereas this term is raised to the power zero. So it equals one. So this picks out the correct value, Sigma or one minus Sigma, for each observation. And now notice that this distribution on observations is data-specific. So even though we can write it using one simple notation, these probabilities are actually changing for every single observation that we have. And they're parameterized by the covariates 'x' as much as they are by the coefficient vector 'w'. So we're going to discuss an algorithm for maximizing this in 'w'. Notice that when we do learn the vector 'w', prediction for new data is fairly straightforward. If we wanted to predict the most likely label, then we would simply take a new 'x' and evaluate the dot product with the learned 'w'. And declare the class to be plus one1 if it's positive, minus one if it's negative. But we actually also get a confidence of our prediction in a sense. So, we get this probability of the label. So some labels, that we're more confident in, we'll be able to say we're very confident it's plus one. Whereas the labels, whereas the data that fall close to the separating hyperplane will be less confident. And so, we could say we're, say 60 percent confident or something like that. Okay. So there's one more notation change to be made. Again, these sorts of changes are done purely for convenience so that there's less to write. But they make life a lot easier. So let's use this fact here. Let's build in the label into the Sigmoid function itself in order to pick out the correct one for us. So instead of originally having this–which is, again–remember that this is the probability for the i^{th} observation of these, corresponding label being equal to plus one. So that's the Sigmoid function of the probability of a plus one– one minus–that is the probability of a minus 1. Instead of writing it this way, what I'm going to do, is I'm going to feed as the argument the label times the dot product. And if you want to see why that makes sense, notice that, if 'y' is equal to plus one, then the right-hand side is equal to this times one. Because this is equal to zero in that case, so this is equal to one. And, again, 'y' is plus one. And so this, of course, is equal to that. And then, when 'y' is equal to minus one, notice that this right-hand side is equal to this term here. And then, you can quickly verify that if we actually replace 'y_i' with a minus one here, that actually this term is, in fact, equal to this term. So, therefore, using this simpler notation, we can write the joint likelihood as equal to the product of these Sigmoid functions where we also multiply the argument of the Sigmoid function by the label plus one or minus one. So that's why, for logistic regression it's often a lot easier to assume that it's plus one minus one, whereas for the Bayes classifier, we said, plus one or zero. Because subscript with a one or a zero looks nicer than a subscript with a plus one or minus one. These are just arbitrary choices that are made to make things easier to read and easier to write. Okay. So we've defined the model– we've defined the likelihood term of the data given the parameters of the model. Now let's look at one simple, straightforward way to learn this model. So if we've only got the likelihood to work with, naturally what we would want to do is maximize that likelihood with respect to the model parameters. So we end up with this sort of a situation where the maximum likelihood solution for the vector 'w' is equal to the vector that maximizes the log of the likelihood of 'y' given 'x' and 'w' from the previous notation. That's equal to this. Let's define this objective function as L. So L is equal to the log of the likelihood of 'y' given 'x' and 'w'. Again, as with the perceptron, we can't take the derivative of this thing with respect to 'w' and set it to zero and actually solve for 'w'. There's no analytic solution for finding the solution to this. So we're going to need some sort of an algorithm–an iterative algorithm for learning 'w'. So we'll look at the first thing to consider. And what we'll talk about in this lecture is gradient descent. So we want to iteratively refine the vector w. At each iteration we set the update to it– to the vector 'w' equal to the old value plus some small step size that's positive times the gradient of the objective function evaluated at the current value. And in this case, you can verify that given our definition of the Sigmoid function, when we take the log of it and sum overall of the data instances and take the gradients, we end up with a function that looks like this. Notice that this term is some number between zero and one.

'y' is just a sign basically. It's plus or minus one. So this is simply flipping the direction of 'x' which is a vector. So, this term is a vector. We're summing each of these data-specific vectors to get the gradient. This is the direction in which to move to increase the objective function. And so, we take a certain step in that direction to update our vector 'w'.

Video 3 (08.22): Logistic Regression Algorithm

Okay. So we've defined the likelihood, and we've discussed how we can use steepest ascent to maximize that likelihood in the vector 'w'. Here is a summary of the algorithm. We have as input our labeled training data, our labeled training pairs. So this should be a one. And some step size that we set that's nonnegative and close to zero, we'll initialize the vector to be a vector of zeroes- so we'll initialize 'w' to be a vector of zeroes. Then for each iteration, what we do, is we take the old value at iteration 't', and we modify it by adding this vector to get the new value at iteration '$^{t\ plus\ 1}$' where this term is equal to the gradient evaluated at 'w^t'. And so, how does this compare to the perceptron? Remember, for the perceptron, what we had here was something that looked very similar. The only difference was that we've replaced this by first searching for a misclassified example, say that's the ith example, and then replacing the sum over those misclassified samples with just one-randomly chosen one because we were doing stochastic optimization for perceptron. However, remember that we discussed that we could have equivalently summed over all of the misclassified examples, so, we could have a sum over all 'i' that were misclassified, and that would be an equally valid and accurate algorithm. So in a sense, the difference between the perceptron algorithm and logistic regression is this term here. With perceptron, we simply did not have this term, and we were only summing over the misclassified examples. Now, we sum over all of the examples, and we multiply each one by a term here. Notice that this term is going o be equal to one, or it's going to become closer and closer to one; the more, the worse and worse. Our current model predicts that data point. So if our data-our**,** labeled instance is plus one, but our model, given the current vector 'w' is very confident in predicting minus one, then this is going to be very close to zero, in which case we'll have something very close to one here. On the other side, if our labeled instance is say plus one and our model is very confident in predicting plus one for that instance, this will be very close to one, and then we'll have something that's essentially equal to zero. So this value here, that's pre-multiplying 'y' times 'x' is like a measure of how confident we are in our prediction. As we become more confident, we're going to have zero. So we're not going to want to change the hyperplane for the things that we get right, which means that for the instances that we very accurately predict, we don't want to modify the plane using that instance. But for the ones that we get very wrong, we want to use those instances to help, guide, and direct the hyperplane so that we can predict those values better. So in a sense, logistic regression has this fudge factor, if you will. It's simply a result of the probabilistic interpretation that we've given to the problem, whereas if you remember, perceptron had no such probabilistic interpretation at all. So there's one more issue, however, that we need to address, potential issue that can come up with logistic regression, and this is the case where the data is able to be separated by a linear hyperplane perfectly. So we can find a hyperplane to cut through the data, such that all of the plus 1s are on one side, and all of the minus 1s are on the other side. In that case, if we did maximum likelihood for that model, what we could show is that the magnitude of 'w' would blow up and go to infinity, because as 'w' goes to infinity, it's going to make all of the data on the plus side, it's going to make the dot product between all the data on the plus side with that 'w' become positive infinity, and it's going to make the dot product of all the data on the negative side with that vector 'w' become minus infinity. And in that sense, we can maximize the likelihood, because we can predict that all the data on the plus one side are going to be plus one with probability one, and all of the data on the minus one side are going to be minus one with probability one. So there's no reason to have any uncertainty– to use the uncertainty that the model gives us if we can do everything perfectly. So just as with the least squares case, where we made the transition to Ridge Regression, we can do the same thing here. We can add a regularization term that penalizes magnitudes of 'w' becoming too big. And just like with regression, we can also motivate that as adding a Multivariate Gaussian prior to the vector 'W', in which case we have, I should say, Lambda over two times the dot product of 'w' with itself as the additional term in the optimization where this corresponds to a Gaussian prior 'w', in which case we're not doing maximum likelihood anymore. Remember that we're doing a maximum posteriority because we now have this additional prior term regularizing magnitudes of the vector 'w'. So there's a parallel with regression. Remember, with regression, we saw that maximum likelihood and least squares had a certain relationship to each other. And then Ridge Regression was like MAP. And instead of doing MAP, we then discussed how we might want to find the posterior distribution of the unknown model parameter, and we found that we could do that with the Bayesian linear regression model. It is a Multivariate Gaussian with a meaning in covariance. Now the question is, we've defined the likelihood model for the labels to be this Sigmoid function. And we've defined a Gaussian prior on the coefficient vector 'w'. We can do MAP for that. But can we do Bayesian inference? Can we find the posterior of 'w'? Because we have defined likelihood in a prior. So the answer we'll see is no, we can't calculate the posterior in closed form, which is going to motivate something called the Laplace approximation. So here's the problem that we're dealing with. We've, again, defined a Gaussian prior on the vector 'w', and we have this likelihood of independent Sigmoid functions for the labels of an observation given the covariates and given 'w'. So by Bayes rule, we know that the posterior is equal to the likelihood term times the prior. And then, we divide that by the integral of the numerator over the unknown model variable. In this case, 'w'. If we were to actually plug in a Gaussian density here and plug in the Sigmoid function here, and try to calculate the denominator in this case, we would quickly find that it's not a tractable integral, that we can't find this normalizing constant, and so we can't solve Bayes rule and define and say what the posterior distribution is. So the next question in that case is, if we can't say what the posterior distribution is, can we somehow approximate it accurately?

Video 4 (12.25): Laplace Approximation

The Laplace approximation is one approach to approximating this posterior distribution. What the Laplace approximation does is, it takes the posterior and it says that since we can't calculate this analytically, we can't say what this is exactly we'll approximate it with a Multivariate Gaussian distribution. And that distribution is going to have a mean and a covariance. And now the question is, how do we set the mean and the covariance? We want to set these two parameters such that this distribution is approximately equal to this distribution. Actually making this type of an approximation alone does not actually define what the Laplace approximation is.

The Laplace approximation is where we make this Gaussian approximation to the posterior and then we have a specific way for finding these two parameters. And so, let's see how we would do that. Okay. So let's motivate–let's kind of the rewrite the problem in a way that will make it easier to see what the Laplace approximation is doing. So notice, from Bayes **rule** that we have the posterior is equal to the joint likelihood divided by the marginal over the model variable. And so, what we've done is we've become clever and written this joint likelihood of 'y' and 'w'. Remember from the rules of probability, this is also equal to the likelihood of 'y' times the prior of 'w'. I've written now where both terms are on the left side of the conditioning bar, just to have less to write.

And let's do the simple transformation of taking the log of that joint likelihood and then exponentiating it. And so, of course these two things cancel each other out and we're left with the likelihood again. However, what we're going to do now is using this representation we're going to approximate the log of the joint likelihood. So we're going to replace this log joint likelihood in the numerator and the denominator with an approximation of it. So again, there are many approximations that we could use. Since we're right now interested in discussing the Laplace approximation that means that we're going to use a second-order Taylor approximation.

So what we're going to do is define this log joint likelihood, remember it's a function of the data and also the unknown model variable. The model variable is the only free parameter that we get to change, obviously we can't change the data. So let's redefine this log joint likelihood as 'f' and because 'w' is the free parameter we'll keep 'w' and suppress 'y' and 'x'. So 'f' of 'w' is the log of the joint likelihood which is a function of 'w'. And now let's replace this function 'f' with a second-order Taylor expansion of that function. Okay. So a second-order Taylor expansion of a function 'f' –of a vector 'w'. In order to define this second-order Taylor expansion, we need to define one more vector 'z' that's in the same space as the vector 'w'.

So in our case, 'w' is an 'R^d' plus one and so 'z' is going to also be an 'R^d' plus one. And so, then we do a second-order Taylor expansion about the point 'z', which means that we say that 'f' of 'w'. So remember, this is a function of a free vector 'w', we can change 'w', it can take any value in the space 'R^d' plus one. Is approximately equal to 'f' of 'z'; now 'z' is fixed– we've defined one specific point 'z' and that's never going to change. And so, we have this, the function evaluated at that specific point 'z' plus this first-order term which is the difference between 'w' and 'z'. So 'w' is free to change, 'z' is fixed. So this is the difference between 'w' and 'z' times the first order derivative of 'f' of 'z', so this is a vector–so this is the gradient of 'f' evaluated at 'Z'. Plus a second-order term which is one half times the difference between 'w' and 'z' times the matrix of second derivatives.

So this is the second derivative of the function 'f' evaluated at 'z' times the same difference between 'w' and 'z'. So this is again on the right-hand side an approximation of the left-hand side. They're both functions of 'w' but the difference is that the right-hand side also takes a particular point 'z' that we have to pick. What the Laplace approximation does is, it picks 'z' to be the MAP solution. So, we find the MAP solution of the regularized logistic regression problem using gradient methods. And then, we define 'z' to be that MAP solution. We expand–we do a second-order Taylor expansion of the log of the joint likelihood about its MAP solution. So if we now use this approximation, and we plug it back into what we've been working with, recall that the function 'F' is the log of the joint likelihood. We're defining 'z' to be the MAP solution. Then using the Bayes rule, we simply have that the posterior of 'w' is equal to 'e', the exponential of 'f' of 'w' divided by that integral of that numerator over 'w'. We then replace the original function 'f' of 'w' with its second-order Taylor expansion. So we've got an equality here. We now do a second-order.

Taylor expansion of this–what's in the exponent and so that turns this into an approximation. But again, we've taken the function of 'w' up here and simply replaced it with this function of 'w' down here. And now, we can simplify this term in two ways. First, notice that this term here, which is a function of a fixed point that we pick in this case 'w' MAP. So this term here, appears in both the numerator and the denominator and it doesn't involve 'w' at all, it involves no free variables. So, we can simply represent this numerator as a product of this and the exponential of this, as well as in the denominator and therefore these two terms will cancel. So these terms cancel each other out in the numerator and the denominator. Also remember that we've defined 'z' to be the MAP solution, so there was a very specific reason for this that we picked the second-order Taylor expansion to be expanded about the MAP solution. And that's because the gradients of the log of the joint likelihood at the MAP solution is equal to zero. So imagine that the log of the joint likelihood looks like this, we find the MAP solution which is this point right here, so this is the MAP solution right here. Then the gradient, by definition of the MAP solution, which is the maximum is equal to zero. So the gradients of this function at the MAP solution equals zero. So by definition of how we pick 'z' here, this term and this term cancels, so we can cancel these two terms out because they both appear in the numerator and the denominator, they cancel each other out by division. This term is a function of 'w' so can't cancel each other out because they're both functions of 'w'. However, this gradient term is equal to zero. Therefore, whatever this difference is, we have a vector– we have zero as a solution to the stock product. Therefore we're left with one more term in both the numerator and the denominator, which is this. So now, I've gone back to–I've replaced 'z' and 'f' with what they actually are in this case. We have the Laplace approximation approximates the posterior of 'w' in this way. Where I've done one more thing, I brought out a negative here and then put it in a negative there, so these two negatives cancel. However writing it this way makes the distribution stand out very clearly. We can see that this is a Multivariate Gaussian distribution. We can just by inspection– we can see that it's a Multivariate Gaussian distribution.

That the mean of it is equal to the MAP solution. And the covariance is equal to the inverse of whatever this matrix is. So again, we approximate the posterior of 'w' to be a Gaussian with a mean and covariance. The mean is equal to the MAP solution and now the covariance is equal to this which has a name. This is equal to this matrix is equal to the Hessian. And so, this is the negative inverse of the Hessian. So therefore, for our specific model, we have to actually calculate this. If we wrote out the log of the joint likelihood and took the second derivative of it with respect to 'w', and then evaluated that second derivative at the MAP solution, we would find that it's equal to this term here. So again, notice that these are all things that we have.

We have the data 'x' and 'y', we solved the MAP solution. Therefore there is no more unknowns, we simply evaluate this matrix, take its negative and its inverse and define the covariance of the Gaussian to be equal to Sigma. So this slide summarizes the Laplace approximation for doing Bayes and logistic regression. We have labeled data pairs 'x' and 'y'. We've defined the likelihood of 'y' given 'x' and coefficient vector 'w' or regression coefficient 'w' to be the Sigmoid function, which is equal to this term here. We've defined a Multivariate Gaussian prior on a vector 'w' like this. If we then want to approximate the posterior distribution on 'w', we can run an algorithm, an iterative algorithm to find the MAP solution of this problem, which is the arg max of the log of the joint likelihood here. We then defined the posterior mean to be that MAP solution. And now we want to expand uncertainty around that MAP solution in the covariance of the Gaussian.

So, we then calculate this function of our log joint likelihood, take its matrix of second derivatives, evaluate at the MAP solution, and then invert it, take the negative and define that to be the covariance. And so what we can see is that with Laplace approximation we're essentially doing MAP except now to the MAP solution we attach also a covariance matrix that says how certain are we in our MAP solution. And then defined the distribution on our unknown vector 'w' to be a

Gaussian where the mean is MAP and the covariance is the inverse of the negative Hessian.

Video 5 (06.17): Feature Expansions I

In this lecture, we're going to look more deeply at what these feature expansions are, and how we can use them to define something called a kernel, which is a general purpose technique that's very useful for regression, classification, and so on. And so, we'll first discuss kernels a bit in the abstract, and then see some examples of how they can be used. So a feature expansion, also called a basis expansion, we'll say feature expansions, are something we've done before in this course without talking too much about them. The idea or the motivation is where a linear model in the original feature space doesn't work for some reason. So if our data is in 'R^d', and we want to do either a linear regression model or a linear classifier, doing a linear model in 'R^d' is not going to work because of the way that the data is structured. And so, the solution is to take each data point and map it to a higher dimension, and then do a linear model in that higher dimension. So we're going to now formalize this a bit more, and say that the mapping function is phi. So this function phi takes in a vector 'x' in R, little d, and maps it to a higher-dimensional space, R, capital D. It doesn't have to be R, capital D. It could be some subset of R, capital D, but we'll just say R, capital D where D is greater– Big D is greater than little D. And then once we embed each point in that higher-dimensional space, we're going to do our linear model there. So, for example, we already saw how with polynomial regression on R, so the original-dimensionality of the data is 1, we can do P, polynomial regression using a feature expansion in to RP where we map the one-dimensional 'x' in to a P-dimensional space like this. So in that case, the mapping of 'x' is from 'R_1' to 'R_P' where the dimensions are simply the square, the cube, up to the 'P^{th}' power of 'x'. Another example, we didn't really discuss it, but it's just as legitimate– would be to take our one--dimensional value for 'x' and map it to 'R_2' where the first dimension is 'x', and the second dimension might be an indicator of 'x' being less than some number 'a'. Imagine we know that there's some sort of a jump discontinuity that occurs at 'a'. The second dimension would then take care of that with the linear model. So, the motivation for this is that even though things don't look linear in the original space, in this higher-dimensional space that we map to, things do suddenly start to look linear. So let's–or a linear model can suddenly become useful. So let's look at an example for regression, and then we'll see a classification example next. So this is a toy problem; we want to do regression, a linear regression model, for this left data set where we have the input is 'x', the response is 'y', and so we plot the 'xy' pair like this so 'x' is in R. Clearly, the data doesn't really suit– look like it's suitable for a linear model because if we did a line through this, we would get something like this with least squares which wouldn't take account into account the fact that there's this sort of undulation in the data. So this is the original space. Now we take the data from the original space, and we map it to 'R^2' where we use this mapping. So we take 'x', and we map it from 'R^1' to 'R^2' where the first dimension remains 'x', but then the second dimension is the cosine of 'x'. So now this is our two-dimensional input, and we want to do a linear model there. So, we see now that we have a two-dimensional input space, and again a one-dimensional response space. And this is the type of a mapping that we have. And now if we want to do a linear model in this two-dimensional space, what that amounts to is learning three coefficients, 'w_0' which is the offset, 'w_1' which is the weight for 'x', and then 'w_2' which is the weight for the cosine of 'x'. And now if we plot the decision, the prediction, the regression, you know, function, in the higher-dimensional space, we get something that looks like this. So we get a plane cutting through the data. So what this is meant to show, is that this data, perhaps it's not clear from looking at this, but from this data is all– lies perfectly on this plane in a two-dimensional plane. And then if we go back to the original space, and see what does the prediction look like, what does this two-dimensional input– this two-dimensional input makes a prediction for each value of 'x'. We can show it in the two dimensions as a plane or we can go back and show that same prediction just as a function of the original 'x'. And we get a decision that looks like this. So we've done linear regression in a higher-dimensional space. When we come back down to the original space in 'R^1', we get something that looks like this. So it's nonlinear. It looks nonlinear in the original space, but it's– that's because it's linear in a higher-dimensional space.

Video 6 (06.34): Feature Expansions II

So, here's another example where we want to do classification. So, imagine that we have a data that's inputs are two-dimensional. So this is an input x, and then the label is binary. So, it's 01 classification problem. Let's say that blue is 1, red is 0. So, all of these are labelled 0, whereas all of these points in the outer circle are labeled 1. If we wanted to do a linear classifier in this space, what that amounts to is trying to find some line that cuts through the two classes, which, of course, is impossible, as we can see. However, if we take the original two-dimensional data and map it up to three dimensions using this function where we take– so, for a particular two-dimensional vector 'x', we map the first dimension to be the first dimension of 'x' squared; the third dimension is the second dimension of 'x' squared; and then the second dimension in this map space is the product of the two dimensions of the vector 'x'.

So, we take a two-dimensional 'x'. We project it into a three-dimensional space using this projection rule. And then we plot–What does the data look like now in R3 instead of R2? And so we get something like this. So, we see that, in this projection space, now suddenly we can come up with a– we can define a hyperpplane–a two-dimensional hyperplane in this space that's going to cut through the two classes. So, here we do a linear classifier, where we declare the sine of 'x'–we declare the label of 'x' to be the sine of the function 'w_0', which is the offset plus the dot product between this mapped feature and some coefficient vector 'w', which expanded looks like this. So, now we want to learn four weights, 'w_0' through 'w_3.

Three of those weights are going to interact with the projected data, and those, as we've discussed, this three-dimensional vector 'w' is going to define a two-dimensional hyperplane in 'R_3'. 'w_0' will shift that hyperplane, and so we get the hyperplane that looks like this, and 'R_3' that slices through the two classes. If we then come back to 'R_2' and look at what does this look like, we get something like this.

So, this picture is almost as if you're standing outside this plot and looking down from this direction; you get this sort of a decision boundary. So, it's obviously not a linear decision boundary. We declare everything inside the circle to be classed the red class, everything outside it to be the blue class. But we got this decision boundary using a linear classifier in a higher--dimensional mapped space. To introduce some notation, to give some intuitions, one for regression, another for classification, you probably already thought by now. So, it's nice that I know the answer. I know the mapping in those two cases. I'm looking at the data. I generated the data, so I know what the mapping should be. So, in my problem, however, which expansion should I use? What should I define this phi function of my inputs x to be to map it to a higher dimension? So, in reality, there's not an obvious answer to that question.

The illustrations I showed required knowledge about the data that we're probably not going to have in practice, especially if the data is in higher dimensions than two or three dimensions. So, one quick approach that could be tried is to use what we could call the kitchen sink. So we simply come up with every possibly feature expansion that we can think of. We take our original data in 'R_d', and we map it to as high a-dimensional space as we can possibly come up with, where each dimension is just some arbitrarily picked function of the data. We don't know if it's a good function or a bad function, but we have a lot of these functions. And then we put an L1 penalty on the weights in that space for the classifier or the regression model.

So we–if we take our data point xi and map it into a very high-dimensional space using every function of it that we can think of, all the polynomials, all the logarithms, all the indicators, etc. For each– for whatever-dimensionality that higher-dimensional space is, we're now going to have to learn a weight vector on each of those dimensions w. So, we've seen with regression that when we do an L1 penalty on w, that's like we're encouraging sparsity in the weights of w. And so if we define this so-called kitchen sink expansion–feature expansion and then put a sparse prior on the weights, we could imagine that we could then let the model decide which of the features–which of the functions that I chose are actually useful for my problem. So, that's one possible approach. However, to motivate the rest of the lecture, for many models, it actually can be shown that we don't actually need to work in the feature space–expansion space. That all we ever need to work with are these dot products between our feature expansions. And so I'll make that clear in the following slides. So, the key thing right now is to know that we define a dot product between the feature expansion for two data points to be something that can be called a kernel. So, we define this dot product to be a kernel function between 'xi' and 'xj'. And thinking about the problem this way can lead to some interesting results.

Video 7 (04.52): Feature Expansions II

So, remember, with the perceptron, that we have covariates 'x_i' for the ith data point in 'R^d', so 'R^d' plus 1, so that plus 1 implies that I've added a dimension to 'x' that's equal to 1 to count for the offset. And with each 'x_i', we have a label 'y_i' that's either plus 1 or minus 1. So, we are going to do plus 1 minus 1. And we have n of these pairs. So, if you recall, for the perceptron, we constructed a hyperplane w. We constructed our decision boundary defined by the vector 'w' as a sum of these misclassified observations. So, let M be a set that's sequentially constructed of misclassified examples. So, remember, with the perceptron, the way that we updated w at iteration$^{t \text{ plus } 1}$ is that we set it equal to 'w' at iteration 't' plus 'y_i' 'x_i' where we picked the pair 'y_i' 'x_i' to be a misclassified example according to our current vector 'w'. So, if '$w^{t \text{ plus } 1}$' is simply constructed by taking w^t and adding this misclassified example, the same holds for 'w^t'. And so if we unfold what each of these vectors are, we ultimately get to this sort of a representation. So, our final classifier, our final decision boundary, is defined by a sum of a sequence of misclassified examples times the label that is associated with that example. So, this is what we end up with. So, now let's think how we would use the result of the perceptron to predict a new label for new covariant vector that's coming in to us. So, we want to predict a 'y0' for a new 'x0'. And the way that we do this is we simply declare 'y0' to be the sign of the dot product between 'x0' and the classifier that we learned 'w'. Well, if we look at what 'w' actually is, this is simply equal to the dot product between 'x_0' and 'x_i' for 'i' and some set M. So, we do this for each 'x_i' in this set, multiply it by its label, sum them up, and then take the sign of that. So, up to now, we've taken feature expansions for granted. We said, "Let x just be some function of the original x." Let's now explicitly write this as the feature-mapping function phi. So if we want to–instead of working in the original space 'x' in 'Rd' plus 1, if we want to now map this to a much higher dimension, we define the mapping function to be phi. So, phi takes in 'x_0' and then it maps it to higher dimension according to some definition of the feature expansion that we want to use. And then we simply do linear classification in that higher-dimensional space. So, in the projected space what we're doing is we map 'x_0' into the higher-dimensional space. Then we take the dot product with 'w' that was learned in that higher-dimensional to be the sign of that dot product. Now, if we again unpack what 'w' actually is, remember, we're working in the higher--dimensional map space, so these 'xi's' suddenly become the function phi of 'x_i'. So, this is simply 'x<sub>i<sub>' projected into the same exact space that we project 'x_0' into. And so now our dot product is between the feature expanded vectors for 'x_0' and 'x_i'. And so this is simply what we're going to call the kernel between the pair 'x_0' and 'x_i'.

Video 8 (04.09): Kernels II

Okay. So now let's actually define a kernel in the abstract and then we'll come back to it and look at it and some examples. So a kernel, call it K, it's a function that takes two inputs both in the space R^d and maps that to an output in R. So it's a function of two inputs in some space R^d and it takes those two inputs, does a function of them and it maps that to just a real valued output, so R1. d is arbitrary, but the output is a real value number. Also it's a symmetric function, so K of x...x_1 x_2 is equal to K of x_2 x_1 And furthermore, if we take a set of n data points, x_1 to x_n, all of them in R^d and we construct an end by n by n matrix, so these are n arbitrarily chosen points. And we construct an n by n matrix K where the IJxth value in that matrix is just the kernel function between x_i and x_j. Then this n by n matrix is a positive semi-definite matrix. And so intuitively what that means is that K satisfies the properties of covariance matrix. You don't need to think in those terms, but that's some intuition about how we could think about K as a covariance matrix. So there's an important theorem that we're not going to prove but we're just going to state called Mercer's Theorem. And that theorem says that if we can come up with some kernel function K that satisfies the properties defined here, then there exists some mapping function phi that takes a vector in the original space R^d, maps it to some other space R capital D, such that the kernel function between x_i and x_j is the dot product between their respective mappings. And so it's important to know that this is for any pairs of x_i, x_j that we can come up with. If this is satisfied for every single set of n vectors in R_d where n is arbitrary and the vectors themselves are arbitrary, then there exists some function phi, which is a function that's one function it takes in any particular x and it performs the same function on that particular x to map it to another space. And we can then have this dot product representation of the kernel function. So this is not interesting when we actually define the mapping function phi. If we define phi first and then we define K to be the dot product so essentially we start from this definition, we start by defining phi and then we define K to be the dot product. Then this is trivially satisfied, we can show that this is just a natural conclusion that follows. However, the interesting, more interesting thing is when we start with the definition of K and then we avoid ever using phi, we never even bother with this mapping we just start with the definition of K. We only ever use K in our algorithms but we prove this certain property about the function that we define so that we can know that intuitively there is some mapping that we could have done that would've given us this K, even if we never find out what it is.

Video 9 (06.29): Kernels III

So, what is an example of a kernel function that can be defined directly that we would work with? It seems that the most popularly used one is what's called the 'Gaussian kernel' or the radial basis function. Those are two different names for the same thing. And, this is where we take in a point x and x prime, and then we perform the function of those two points like this. So this is essentially, it looks like a 'Gaussian distribution' which is why it's called a 'Gaussian kernel'. Notice that as the points x and x prime get closer and closer to each other, this becomes smaller, and the entire kernel converges to 'a' and the limit as they are equal. As the two points x and x prime get farther and farther apart, this become bigger and bigger. And so, this entire term goes to zero, and the kernel is, then converges to zero as these two go infinitely far apart. So, this kernel is like a measure of proximity between these two points that we input to it. It gets bigger as they become closer, and it gets smaller and goes to zero as they get farther and farther away. And, the largest value is 'a', and the smallest value is zero. So, it's a kernel that's between zero and 'a'. Okay, so we aren't going to go through proving that the result of this is a positive semi-definite matrix, and therefore, 'Mercer's theorem' holds. We're just going to state that that's the case. And so, in this case, because we can prove that for any set of n points, 'x_1' to 'x_n', the matrix constructed from this kernel has to be positive definite, actually. It, therefore, follows that there is some mapping of these vectors into a higher-dimensional space such that this function results from their dot product. So, this higher-dimensional space is actually an infinite-dimensional function space for this case. So, therefore, we can't write the function of 'x' as a vector. It would be infinite--dimensional, and it's actually also a function. So, we simply state, without explaining or expressing what this is mathematically, that there is a function of 'x' and 't' such that this kernel results by integrating the product of those two functions.

So, a dot product in finite space becomes an integral over a product of two functions in an infinite-dimensional space. So, we integrate the product of these two kernel mappings over this parameter 't'. So, we aren't going to discuss in detail what this mapping is, not because it's not interesting. It is, and it's not because we don't know what it is. We do. A good book on 'Gaussian processes' will discuss this. But, primarily, because we'll see that the algorithms that we're going to develop with this kernel, this idea of using kernels both today and in a few lectures down the road, never need this mapping. They only ever need this definition of the kernel function itself.

So, we only ever actually need to calculate this. We never actually need to calculate the mapping itself. Okay. So, before we do this, just not to give the impression that kernels are equivalent to the radial basis function of the Gaussian kernel, there are other kernels. For example, a kernel could map an input of vector 'x' like this. So, you can pick out the pattern of what this mapping is. And then, if have this higher-dimensional mapping, we take the dot product between two points, x and x prime, and we can show that the mapping, then, becomes one plus their dot product in the original space, 'quantity squared'. So, a higher-dimensional mapping that looks like this. If we took its dot product would result in this kernel function. So, instead of mapping to this space and taking dot products, we could simply, if we only ever need the dot products, we could simply calculate this function.

And, in fact, we can show that for all values of 'b' greater than zero. This function is a kernel as well. So, for 'b' equals three, there's some mapping of x and x prime to a higher-dimensional space such as the dot product in that space is equal to one plus x transpose x prime to the 'b', or for the value 'b' equals 3.245, there's also some higher-dimensional mapping. So, we can construct kernels using other functions such as this one. Also, important to know is that certain functions of kernels are still kernels. So, we won't necessarily use these or discuss these in more detail, but to give some examples. If we have two kernel functions, 'k_1' and 'k_2'. So, these are two different functions that we know, both of which are kernels.

For example, k_1 could be the 'Gaussian kernel', k_2 could be this kernel. And then, we perform some functions of these kernels. For example, we construct a new kernel where we take the function between x and x prime to be the product of these two separate kernels on those same two points. That's still a kernel. Or, if we add two kernels, we still have a kernel. Or, if we exponentiate a kernel. That's still a kernel. So, what that means is that if we take these two functions, multiply them together to get this, and then we do that for n points to construct a matrix. There is some high-dimensional mapping, that would give that original function. Maybe it's not going to be the same as the mappings for these two, but there is still some mapping that exists.

Video 9 (06.29): Kernels III

So, what is an example of a kernel function that can be defined directly that we would work with? It seems that the most popularly used one is what's called the 'Gaussian kernel' or the radial basis function. Those are two different names for the same thing. And, this is where we take in a point x and x prime, and then we perform the function of those two points like this. So this is essentially, it looks like a 'Gaussian distribution' which is why it's called a 'Gaussian kernel'. Notice that as the points x and x prime get closer and closer to each other, this becomes smaller, and the entire kernel converges to 'a' and the limit as they are equal. As the two points x and x prime get farther and farther apart, this become bigger and bigger. And so, this entire term goes to zero, and the kernel is, then converges to zero as these two go infinitely far apart. So, this kernel is like a measure of proximity between these two points that we input to it. It gets bigger as they become closer, and it gets smaller and goes to zero as they get farther and farther away. And, the largest value is 'a', and the smallest value is zero. So, it's a kernel that's between zero and 'a'. Okay, so we aren't going to go through proving that the result of this is a positive semi-definite matrix, and therefore, 'Mercer's theorem' holds. We're just going to state that that's the case. And so, in this case, because we can prove that for any set of n points, 'x_1' to 'x_n', the matrix constructed from this kernel has to be positive definite, actually. It, therefore, follows that there is some mapping of these vectors into a higher-dimensional space such that this function results from their dot product. So, this higher-dimensional space is actually an infinite-dimensional function space for this case. So, therefore, we can't write the function of 'x' as a vector. It would be infinite--dimensional, and it's actually also a function. So, we simply state, without explaining or expressing what this is mathematically, that there is a function of 'x' and 't' such that this kernel results by integrating the product of those two functions. So, a dot product in finite space becomes an integral over a product of two functions in an infinite-dimensional space. So, we integrate the product of these two kernel mappings over this parameter 't'. So, we aren't going to discuss in detail what this mapping is, not because it's not interesting. It is, and it's not because we don't know what it is. We do. A good book on 'Gaussian processes' will discuss this.

But, primarily, because we'll see that the algorithms that we're going to develop with this kernel, this idea of using kernels both today and in a few lectures down the road, never need this mapping. They only ever need this definition of the kernel function itself. So, we only ever actually need to calculate this. We never actually need to calculate the mapping itself. Okay. So, before we do this, just not to give the impression that kernels are equivalent to the radial basis function of the Gaussian kernel, there are other kernels. For example, a kernel could map an input of vector 'x' like this. So, you can pick out the pattern of what this mapping is. And then, if have this higher-dimensional mapping, we take the dot product between two points, x and x prime, and we can show that the mapping, then, becomes one plus their dot product in the original space, 'quantity squared'. So, a higher--dimensional mapping that looks like this. If we took its dot product would result in this kernel function. So, instead of mapping to this space and taking dot products, we could simply, if we only ever need the dot products, we could simply calculate this function. And, in fact, we can show that for all values of 'b' greater than zero.

This function is a kernel as well. So, for 'b' equals three, there's some mapping of x and x prime to a higher-dimensional space such that the dot product in that space is equal to one plus x transpose x prime to the 'b', or for the value 'b' equals 3.245, there's also some higher-dimensional mapping. So, we can construct kernels using other functions such as this one. Also, important to know is that certain functions of kernels are still kernels. So, we won't necessarily use these or discuss these in more detail, but to give some examples. If we have two kernel functions, 'k_1' and 'k_2'.

So, these are two different functions that we know, both of which are kernels. For example, k_1 could be the 'Gaussian kernel', k_2 could be this kernel. And then, we perform some functions of these kernels. For example, we construct a new kernel where we take the function between x and x prime to be the product of these two separate kernels on those same two points. That's still a kernel. Or, if we add two kernels, we still have a kernel. Or, if we exponentiate a kernel.

That's still a kernel. So, what that means is that if we take these two functions, multiply them together to get this, and then we do that for n points to construct a matrix. There is some high-dimensional mapping, that would give that original function. Maybe it's not going to be the same as the mappings for these two, but there is still some mapping that exists.

Video 10 (13.59): Kernelized Perceptron

Okay. So let's now return to the perceptron and think about how do kernels relate to that specific model? How can we use them in that specific model? So we saw that we could write the, feature expanded decision in this way. What we want to do is for a new 'x_0'. We project it into its higher-dimensional space using the function 'phi'. We project each of the data points that are in the set 'M' into a higher-dimensional space 'phi'. Remember that this set 'M' was constructed by sequentially picking misclassified examples from the data set. And now instead of actually taking these dot products between these points, we replace them with the kernel function, between these two points. And now let's pick the radial basis function and set an equals one so we have less to write. And see what is the decision that we make based on the radial basis function. So in that case, if we use, the radial basis function as, the kernel. Then we're making our decision to be the sign, of the sum of the kernel function between, the new point 'x_0'. And an old point 'x_i' in the set 'M'. So that kernel function times the label which is going to be either plus one or minus one, of the point 'x_i'. So again, notice that for this kernel function, the higher-dimensional mapping is actually, an infinite-dimensional thing. So we can never even write it out anyway. But we never have to write out that higher-dimensional mapping because we only need the dot product. So why could this possibly be useful? Why might we want to do this? So let's look at what is the decision that's being made in this particular case? So in a sense, we're letting each of the data points in the set 'M' vote for a label. For the new point 'x_0'. And if a point 'x_i' is close to 'x_0', then this function will be closer to one. And so we'll have the label times a number closer to one. Or if we have a point 'x_i' in the set 'M', that's very far away, in the space from x naught. Then this will be essentially zero. And so we then take the label and weight it by something that's essentially zero. And then we take the labels of all the points in our set 'M'. And we weight them by this function that takes into account how close he new point 'x_0' is to that point 'x_i'. And then we sum it up. So it's like saying that if x_0 is very close to ' x_i ', then I trust that it's going to show, more likely share the label of ' x_i '. Whereas if 'x_0' is very far away from ' x_i ', then I can't say whether or not it's going to share the label with ' x_i '. So I'm going to, I want to weight the labels of the data points that I am closer to, more highly. And weight the things that I'm far away from, less and less.

And then sum those weighted labels up. So now I'm actually summing the positive weights for all of the positive-labeled points in 'M'. And then I'm subtracting all of the weights for all of the negatively labeled points in class minus one that are in m. And I'm seeing which is, which is bigger. It's positive, then I know that I'm closer on average to plus-one labeled points. And if it's negative, I'm closer on average to minus-one labeled points. And so in a sense, it's like soft voting. I'm letting the labeled points in the set 'M' do a soft voting. It's almost like a soft nearest neighbors in a sense. Okay, so we've discussed making predictions with a perceptron and how we don't need to do the mapping. We can just use the kernel.

However, you might be thinking that there was also an algorithm that we had to run for learning the sequence of misclassified points. And that also will require a higher-dimensional mapping as well. So how can we have to actually do the high-dimensional mapping to learn the perceptron? To actually pick the set 'M', the sequence of misclassified points? And again, the answer is no. We never actually need to construct, need to calculate this feature mapping. We only have to actually work with the kernel function between different points in our data set. So let's go back to the original space. The, there's no mapping.

We're not mapping the data. We're in the original data space. Recall that at iteration 't' we had a separating hyperplane that's defined by this vector 'w' at iteration 't'. That we had constructed by taking a sum of the misclassified points up to iteration 't', like this. So this is our linear classifier at iteration 't' of the perceptron. We then updated this as follows. We first, found a new x prime in the data set. Such that we predicted incorrectly given this classifier. So go through our data set for each x prime 'c'. Does the classifier at iteration 't' protect-predict the associated label y prime correct-correctly or incorrectly? And the first one that we come across that's incorrect, pick that x prime. We then add the index of x prime to this misclassified set 'M'. To get 'm' 't' plus one now. Points. So we have 't' plus one points. And then we update the classifier by taking the old classifier plus the weight y prime, or the label y prime times x prime. Or equivalently, we can simply say, add up all of the misclassified covariates. Pre-multiplied by the sign of their label.

So again, if we look at how we're picking out the misclassified points, how are we constructing this, this set 'M', sequentially? We're, we're only ever using dot products. So again, we only ever need to look at dot products to decide whether we're going to misclass-whether our current classifier misclassifies a point or doesn't misclassify a point. So if we expand 'w_t' by replacing it with what it's a function of. And then we write the dot products out. And then we expand this sp-the features into a higher-dimensional space. We see that all we need to do is calculate the kernel function. So actually this is like what we've been discussing. Making predictions, on new data points. We're simply here trying to make predictions, on the data that we already have. And see which, which things in our current data set that we're learning the classifier on do we predict incorrectly? And so the intuition here is exactly the same as what we've been discussing. So really what we need to do is find a label that's not equal to the sign of the label-weighted kernel functions between the new point and some point already in our misclassified set. So in order to find a misclassified point, we only need to calculate the kernel between a proposed point, and the points in our current misclassified set. And then when we want to augment the mis-the set 'MT' to get the set 'MT' plus one, we can simply take the index of the misclassified point. But we never actually bothered to calculate the new classifier.

Because we never need it. We only ever need dot products. So we've seen that we can do 'kernelized perceptron' where all we ever need to do is to define the kernel function between any two points in our data set. In order to learn the kernel. In order to learn the classifier. Or between a new query point and the points in the set 'M' that we construct of indexes of data points that we pick out from our data set. That were misclassified at some point during the inference or the learning of the 'kernel perceptor'. So let's see how can we do a quick extension. So this is something that's a very quick extension of this idea. Something called 'kernel KNN'. So this is basically taking the 'kernelized perceptron' and generalizing it to a soft KNN with a very simple change. So instead of constructing this sequence of, of misclassified data points. Which we have in the set 'M' using the 'kernelized perceptron'. So instead of picking out a subset of our data points and saying, "I'm only going to use these data points to classify any new data points." And essentially throw away all the data that other, all of the other data that we have. Why don't we instead calculate, construct this set 'M' to have all of the data that we have? So instead of summing over all 'i' and some index set 'M', let's just sum over every single data point in our data set where we let each data point vote on the label of the query point x_0.

So we calculate the kernel. This is again, we'll pick the RBF kernel between a query point x_0 and the point xi in our data set. This will be big if x_0 is close to xi. It'll be very, it'll be very small, essentially zero, if x_0 is very far away from x_i. And that's how much we're going to weight the label of x_i in this voting scheme. Now if we want to intuitively think you know what does this mean? We could divide this number by the sum of the weights. So this is arbitrary. We define 'z' to be the sum of these weights. And then define a probability distribution that takes in a point x_0. Where the i^{th} dimension of that distribution can be viewed as like a probability for the i^{th} data point in our data set. So this is like a probability for a queried x_0 to the i^{th} data point in our set. And then we declare y_0 to be the sign of this weighted average of the labels in the data set. So this decision would be identical with this decision. But in a sense we can now interpret this as like a probability distribution on all of the data points in our data set.

For a given query point. And this distribution is going to be have high probability on the points that are closer to x_0. Essentially zero probability on the points that are far away from x_0. And then we're only going to take into consideration the labels to x_0 in making our decision. So in a sense, we can think of this as a soft or a 'kernelized KNN' where 'K' is just equal to the number of data points. We allow every single data point to give a vote on the label of a query point. But then we weight that vote according to a confidence score. And we're more confident in the votes of the data points that we're close to, according to our kernel. And not confident at all about the vote of the label of the points that we're far away from. So in a sense, we notice that the only difference between this classifier and the 'kernelized perceptron' is that the 'kernelized perceptron' only allows a small subset of the data points to vote for the label. And that subset is determined according to the rule of picking out a sequence of misclassified points. Whereas this allows every point to give a vote. However, when we define 'b', we're going to define it such that for any point x_0, we're only looking in a small window around x_0. And so we're only going to look at really a small subset of the data points in our data set to give a vote.

That are within a window, or within kind of a horizon to find by 'b'. And then the data points that are far away as according to the definition of 'b'. Essentially because this is decreasing extremely fast as this value gets bigger and bigger. Essentially those points are given zero weight.

Video 11 (09.24): Kernel Regression

The developments are essentially limitless. I just want to give another very quick example to give that impression of how it can be developed, something called the Nadaraya-Watson model. So this is a regression model instead of a classification model, where we want to do kernel regression now. So, again, regression is also– we can do linear regression, and so we can use the same rules. And here what we have is that essentially the only difference is that before 'y' was a plus one or minus one label, now 'y' is a value in 'R'. It's a real value number, and we want to predict a real value number. So what this model does is very similar, to the soft essentially the same as the soft k-NN model from the previous slide. Remember the soft k-NN model declared 'y' not to be the sign of this function, where these 'y_i's' were plus or minus one.

Now 'y' is a real value number, and we're going to predict a real value number, so we simply declare our prediction for 'y' not first and associate 'x' not to be the weighted average of this labels, or, I'm sorry, the weighted average of the responses of all the data in the that we possess that's weighted according to the kernel function to that data point. So here we're taking locally weighted average, if we choose the RBF kernel function, we're basically taking a weighted average of our dataset of the responses that's weighted according to proximity to the point 'x_0' . So another example of this type of a model, which will require some more discussion, and it's also been developed quite a bit, is called the Gaussian process. We can think of the Gaussian process as a kernelized Bayesian Regression model.

So before we can define it, we need to see where in the Bayesian Linear Regression model, that we've discussed already, where do the dot products between the data points come into play. Remember, before we can put a kernel of our data into the model, we need to first have dot products between our data points where we can then justify the kernel by saying we're going to project those dot product the data into a higher dimension, and then take the dot products. So let's recall the regression set up. We have 'n' observations. We're going to put all of the responses for those 'n' observations in an n-dimensional vector, that's in R^n. And then we construct the feature matrix X, where we put the covariates of each data point along the rows of this matrix X. So the ith row of 'x' corresponds to the covariant factor ' x_i '. And the ith dimension of this vector 'y' corresponds to the response associated with ' x_i '. We then define the likelihood in the prior in this way. So the likelihood model that we used was we hypothesized that 'y' is a Multivariate Gaussian, where the mean is the dot product is the product between the feature matrix X with some regression vector 'w'. And then we assumed iid.

Gaussian noise for 'y'. And we placed the prior on 'w' that was zero mean Gaussian with some diagonal covariance. We have to now see where dot products come into play. And so let's look at what happens when we integrate out 'w' in this model. What that means is that we want a distribution of 'y' given 'x', but not given 'w', because 'w' has been integrated out in this way, where we multiplied the likelihood function, that's condition on 'w', times the prior on 'w', and then we integrate out 'w'. We integrate over all values of 'w' to get a marginal likelihood of 'y' given 'x'. I'm not going to derive this here, but what can be shown, and it's a standard result of working with Gaussians, Multivariate Gaussians, is that the result is a Multivariate normal, that's zero mean, and the covariance is equal to the covariance noise covariance plus this term, which results ah from the fact that we're integrating up the 'w' here, and the fact that we have a prior covariance on 'w' there.

So we have this term – Lambda inverse times the product of the matrix X with the matrix X transpose. So this is a result that we can show. Now, notice that we have dot products here. So the value of this matrix, remember, 'x' is 'n' by 'd' plus one. So this matrix product is now 'n' by 'n'. So it's the size of the data. If we have 'n' data points, this is an 'n' by 'n' matrix because 'y' is an n--dimensional vector. The ijth entry in this matrix, by definition of how we made these matrices, is just the dot product between the covariate vector for the ith data point with the covariate vector for the jth data point. So here's where we have our dot products between our data points. They're found and then used to construct this matrix. So now if we take each data point 'X', and we project it into some higher-dimensional space according to the function phi, we haven't defined phi yet, we can then say by following exactly the same set up, where now the ith row of this matrix is phi for x_i, put along that row, we then get a matrix that looks like this.

Again, it's 'n' by 'n'. So this is a matrix where I'm using this notation to indicate that we take each row of the matrix X and perform the function phi on it to get a new row. So this is 'n' rows, because we have 'n' data points, and then however many columns result from our function phi, it could be infinite-dimensional. But then we take the product of that matrix with its transpose, and, again, we end up with an 'n' by 'n' matrix. So this is still an 'n' by n matrix, where the ijth entry is the dot product between phi of ' x_i ', with phi of 'x_j', which is the kernel function between ' x_i 'and 'x_j'. So we only need to define a kernel function between all of our data points. And then what we can say is that if we map 'X' to a higher-dimensional space using some function phi, that the marginal distribution of 'y' given 'X' and given the mapping, integrating out 'w' is now a Multivariate Gaussian, where all I've done is replace this matrix of dot products between our data points with this matrix of kernel functions between pairs of our data.

So this is called the Gaussian process. The Gaussian process comes by integrating out the vector 'w', and then replacing this dot product with a kernel function. So let's notice, before we move on, why this could possibly be useful, at least right after that something obvious about it. So imagine that this function is extremely high-dimensional. Maybe it's even infinite-dimensional. Then this vector 'w' is infinite-dimensional as well, or it could even be a function. There's no chance that we could possibly learn this vector 'w'. We can't even store it in memory. And there's also no chance that we could construct this matrix X or the function phi of 'X' because each column would be infinite-dimensional. However, if we then use this marginalization fact where we integrate out 'w', and it doesn't actually matter that it's in infinite-dimensional, and that we have an infinite-dimensional Gaussian noise prior here, the result is something that we can easily calculate. So we can just calculate the kernel function between all of our data points, and we get a nice n-dimensional Multivariate Gaussian distribution, where we have a way of actually calculating this covariance portion of the covariance 'k'.

Video 12(10.10): Gaussian Process I

The way that we've defined it is not necessarily the way that it's defined in the literature, so let's see how Gaussian processes are often presented. We'll have this function of 'x'. So 'f' is some function, it's a random function of our data point 'x'. It's a function that takes in an 'x' in 'R^d', or 'R^d' plus one, if we have a one–a dimension of ones. And maps it to a value in R. So 'f' is a function that takes in a vector in 'R^d' and maps it to R. We define the kernel function K between any two points, 'x' and 'x'. So we don't actually construct the kernel necessarily, at least in the definition. We simply define the kernel function between two arbitrary points, and it could be any points that we pick in 'R^d'. Then the function 'f' of 'x', it's a continuous valued function, because 'x' is a continuous valued input.

Again, we aren't restricting ourselves to the data in our definition, we're allowing this to be a function of any point in 'R^d', is a Gaussian process. And 'y' of 'x', this kind of shadow function 'y' of 'x', is a noise added process associated with a Gaussian process if we have this, if for any 'N' points that we choose in 'R^d'. So we pick 'N' points where 'N' is an arbitrary but finite number. And then we pick the locations of those 'N' points arbitrarily. We construct the 'K'–an n by n kernel matrix K, where the ijth entry is the kernel between point ' x_i ' and ' x_j '. Then we have that the function 'f' evaluated at those 'N' points. So this is a n-dimensional vector 'f', where we evaluate the ith dimension of this vector, this vector is the function f of ' x_i '. That function is a n-dimensional Gaussian with mean zero.

Sometimes you'll see this mean also be a function. We're just going to say mean is zero. And covariance equal to the kernel. And then the noise added process given the function 'f' is simply a process where we take the function and add iid. noise to each of the dimensions. So notice that actually these two things are equivalent. That if I integrate out 'f', I can actually generate 'y' in this way where I take my original covariance and I add the two covariances together. So the reason that we separate 'f' is because 'f' is the definition of a Gaussian process. So a Gaussian process, we don't assume– we aren't going to assume has noise. We're assuming there's some underlying Gaussian process. And then the data that we observe is a noisy subsampling of this infinite-dimensional function at 'N' points.

So in principle this is a function over all of 'R^d'. It's an infinite-dimensional function. This is an infinite-dimensional thing. But we evaluate it only at 'N' data points. And then we observe 'y', which is the corruption of those 'N' Gaussian process evaluations with some white noise. So the important thing is that the fact that we've defined the kernel and that we've defined this to be on 'N' arbitrary points where 'N' can be as large as we want, is what makes this not just trivially a Multivariate Gaussian. It's more than just a Multivariate Gaussian. So let's look at an example of how I could generate a Gaussian process. So what I have here is I assume that my original data X is a real valued number between zero and one. So I'll just tell you how I generated this.

I then took this and I partitioned it into 1000 points. So I picked 1000 points from this between zero and one that are equally spaced. So the first point was 1 by 1000, and the second point was 2 by 1000. I then constructed the kernel, which was 1000 by 1000 matrix on these points, where the kernel between the ith point and the jth point–think of this as 'i' over 1000 and 'j' over 1000–was equal to the radial basis function, where I took the different—the distance between these two points, squared it, and then calculated this function of it. And so I ended up with a covariance matrix that looks like this. So really this is 1000 by 1000 matrix. It looks like a continuous image, but it's really 1000 by 1000 matrix shown as an image, where points that are farther away–for example, this point here– so this point corresponds to the covariance between let's say this point here and say this point here. They're too far away according to my definition of 'b'. So their covariance is essentially zero. So there's no covariance—there's no correlation between these two points. However, if I get closer and closer– if I pick say two other points, for example, this point here and this point here, say these two points then suddenly my covariance between those two points is large, it's closer to one, because the distance squared is small compared to how I define 'b'.

So these two points, according to my Gaussian process, should be very correlated. Now, there's a lot of theory underlying kernels and Gaussian processes that we won't go through in this class. But what we can show is that with this kernel, as two points go closer and closer together– so as a point–as we have one point at say .7 and then another point gets closer and closer to that point–we can see that this distance between those two points is going to zero, and so they're perfectly correlated. This function is going to one. They're going to be perfectly correlated. with that of what that leads at .7, and I look at a point epsilon, you know, infinitesimally far away from it– because they're perfectly correlated, their function– their random function 'f' of 'x' that I generate is going to be essentially the same at those two points. What that amounts to is that we have a continuous function. So that's the key thing. So what I'm showing here is actually a 1000-dimensional vector generated from a Gaussian process, from 1000-dimensional Multivariate Gaussian, with zero mean and covariance equal to this 1000 by 1000 matrix. And the perfect correlation that happens between two very close points –or the very high correlation between two close points manifests itself in this vector by those two points being essentially the same value.

The value of the function vector 'f' at those two points are very close to each other. So we can really plot it like it's as if it were a continuous function. And it is. I could have partitioned this into one million equally-spaced points and it would've been even more refined. But we get a random function. If I generated this one function and I got something that looked like this, if I generated another vector from exactly the same distribution, I might get something else. I might get something that looked like this. So this way I can generate random functions. And now I'm going to learn those functions based on the data that I have. So what that means is that my model hypothesis is that there's some function underlying the data.

For example, this function was a randomly generated function that's now going to be underlying how the data is generated. But I don't get to observe this. What I get to observe is that randomly-separated points– so here I picked 25 points in the space in which X lives. I think get a noisy observation of this function at those points. So for example, at this point right around here, I missed it. I get a random observation of the function 'f', where I have a mean equal to the function at that point plus some noise that is going to account for the fact that perhaps my measurements aren't exactly right. There's some noise in the system.

So I get noisy samples of this function at different points in X. And so that's my data set. So my data set would be a pair here ' x_i ', ' y_i ', where ' x_i ' is the evaluation of this Gaussian process at the point x. And then ' y_i ' is a noisy observation of that function evaluated at that point.

Video 13 (14.55): Gaussian Process II

What Gaussian processes are useful for is predicting new data using this framework. So we have a hypothesis for how the data is generated. We assume some underlying function, continuous valued function. And we get random samples of this function, and that's the data that we have. Now we want to infer this function so that we can predict the evaluation of it at a new point. 'x_0' now is some new point. So the new point that we have is, for example, say, here we get a new point, and we want to predict what is 'y' evaluated at this new point, 'x_0. We want to learn a posterior of this function so that we can learn that at this point our function should about our function should be around here.

So we'll predict a point that's in this region for the associated 'y' with this input 'x'. We can actually calculate this distribution on this prediction analytically. To see this, let's return to the finite-dimensional space. So we're not going to define the kernels yet. We're going to do it with no kernel definition. We're back to the x matrix to make it a bit more transparent. So we have this set up. Imagine we have 'n' observation pairs, ' x_i ', ' y_i ', and we want to predict y_0 given a new x_0. So according to our model, this observed data and this unobserved or partially observed data is all IID, or, I'm sorry, not IID, it's all independent condition on the vector 'w'. So we can write the joint distribution of 'y' and 'y_0' and y using condition on 'w', and then integrate out 'w'. What that means is, let's imagine now, that I have an augmented factor, where I take the observed 'y', and I have my unobserved 'y_0', now that I make the first dimension of that 'n' plus one-dimensional vector, then according to my model, the distribution on this vector is zero mean, it's a Multivariate Gaussian with zero mean, and covariance equal to Sigma squared I, which accounts for the noise.

So every single value in here has some iid component. That's just white noise. Plus now this additional matrix here there should be a Lambda minus one, according to the original model. So, remember, according to the original model, 'y' was a zero mean Gaussian, so if I got rid of this 'y_0', 'y' was a zero mean Gaussian with covariance equal to Sigma squared I plus Lambda inverse times XX transpose. That's on a previous slide. Now if I view an extended vector, an augmented vector, where I have one additional point here, 'y_0', and an additional covariate vector, 'x_0', then we simply augment this covariance. The mean remains zero mean. And now the covariance becomes this extended matrix, where I have along the lower right the original XX transpose. And now I have to add one row and one column to that matrix.

So in my first entry here I have the dot product between 'x_0' and itself. Then in the lower right I have the product 'X' 'x_0', and then I have that transpose. So what this matrix looks like is I have one point here, which corresponds to this point. I have one vector here, which corresponds to this. I have one vector here, which corresponds to this. And then I have the original matrix from my original problem. So now I've extended my covariance matrix to account for the one additional point that I want to model. A standard result now from Gaussians, which, again, I won't derive it, but what we can say is what is the conditional distribution of 'y_0', given the data and given 'x_0'. So if you tell me the vector 'y', and you tell me the matrix x and the vector 'x_0', but I know this part of the vector 'y', and now I want to predict this missing dimension of that vector conditioned on knowing all of the values in the remaining part of the factor, we can calculate this. It's a univariate Gaussian with mean mu_0 and standard deviation Sigma_0 squared, where the mean mu not is simply, so I'm going to get rid of this Lambda squared because otherwise my equations here aren't necessarily correct, well, imagine that Lambda squared is absorbed in here, I'm sorry, Lambda inverse, the distribution on this remaining dimension conditioned on knowing the vector 'y' and knowing all of the vectors 'x' not through 'x' and 'n', is it a Univariate Gaussian with mean equal to mu_0, where mu_0 is a function of the vector 'y', and this matrix.

Specifically, we take this row vector here, so that's this part, multiply it by the inverse of this matrix here, and then multiply that by 'y'. So that's the mean. The variance of this is now equal to the original noise variance plus another function of this matrix here. We take this point here, and then we subtract from that the product of this vector times the inverse of this matrix times this vector, which is what this is saying. So we can take all of the entries in this matrix and this vector, perform functions of them to get a distribution on the missing value 'y_0'. Okay. So the reason why I wanted to go through that example with finite-dimensional Gaussians, where we aren't thinking about kernels, is that the Gaussian process is simply no different.

We take all of—the only difference is we take all of the dot products, and replace them with kernels. So, remember, we took all of these dot products in this matrix, and replaced it with kernels, so we can replace this matrix with the kernel matrix constructed on the original data. We can take this scalar and replace it with the kernel function of these points.

We can similarly replace these dot products with their kernel functions. So let's do that. So we'll define for a new point 'x', so I'm not saying 'x_0' on the slide, I'm just saying for a new point 'x', this is a continuous valued thing. We can pick any point 'x' that we want. Any point that comes into us, we evaluate it at that point. Or if we want to make predictions at an arbitrary point, we can pick an arbitrary point. We calculate the kernel function between an arbitrary point 'x' with the entire dataset that we have. So that's—we're going to give this a notation the meaning that we take a point 'x' and construct a row vector, where we construct the kernel between that arbitrary point 'x' and each data point, so the first dimension is the kernel between 'x' and 'x_1', and so on.

So this is a 1 by n--dimensional vector. And then we construct the kernel function between all of the data in our datasets. So this is an n by n matrix, where the $ij^{th's}$ value of this matrix, kn, is the kernel function between ' x_i ' and ' x_j '. So those are the definitions of those two things. Then if we want to predict what is the distribution on 'y', the corresponding of 'y' for an arbitrary 'x'. So now we have a function of the response 'y', it's a function of the covariate 'x', given the dataset, and given our evaluation at a point 'x'. That's a Multivariate Gaussian with the mean mu that's a function of 'x', and a co—variance Sigma that's a function of x, where mu is now calculated this way, where we take the kernel of the arbitrary new point to all of the data that we have times the inverse of the kernel on the data that we have times the observations that we have, 'y'. That's the mean. And we have as a variance, again, the noise variance plus the kernel of the new point to itself minus this.

So if we look at the previous slide, we're simply mapping points. Here we had a vector times an inverse matrix times y. In this slide we also have a vector times an inverse of a matrix, where we simply redefine these dot products to be the kernel function. And similarly with the variance. Let's look at a toy example of what we would learn, the posterior of this. So I took a Gaussian process, I randomly generated one, that's the black line. This is the ground truth function that I generated using the same procedure from the previous slide. I set the noise variance Sigma squared to be extremely small, almost zero. So that the samples that we would get would be very close to this Gaussian process. I picked nine points equally spaced out between the two extremes. So, for example, the first point would be this as the input. So my first point would have this as the input and this as the response. So this point is xx1 and y1. And I have that for nine points. I then learned the posterior of this y according to the previous slide. And, remember, I can evaluate now that posterior at any point that I choose according—along this X axis. So I evaluated my prediction I would make for any arbitrary point. I picked essentially 1,000 equally spaced points in this region to say what would I predict for the response if my input were, for example, this point here. The mean, as we saw from the previous slide the prediction is a Univariate Gaussian for each arbitrary point I pick, my distribution on the responses, the Univariate Gaussian, with a mean and a covariance that's a function of kernels between that arbitrary point in all of my nine data points. So I'm showing here, first, the mean function as the blue line. So this blue line is mu of x from the previous slide. So this is the mean of my prediction, which you can see is shadowing the underlying Gaussian process. So I have a prediction of the underlying Gaussian process. mu is the mean of that prediction. Then this red dotted line corresponds to Sigma of 'x'. This is my uncertainty. So what you can see is that as I predict closer to a point that I've observed, my uncertainty gets less. This essentially pinches the function. This point here pinches the function, my prediction, at that point, because I had data there. So I'm very confident about this point. But as I move away from point, I get less and less confident about my prediction. So, for example, in between these two points my confidence in my prediction is much less. And so you can see that my variance increases, but then gradually as I move towards another point, it becomes more and more confidence. Again, I'm pinched at this point. So here the blue line is my prediction of the black line, and the red dotted line is my uncertainty of that prediction. And it gets more uncertain as I move away from points, and more certain as I move towards points. Okay. So what we have here now is a way of making a prediction for any arbitrary data point that's a function. We're really learning an underlying function. And this function, can take any arbitrary form. Notice that is essentially—with this we're doing regression. We're not doing classification for this particular case. And so we're doing, in a sense, nonparametric regression. So we can learn any arbitrary function. If we wanted to do a linear classifier on this data, we would only be able to learn a line through the data. With the Gaussian process what makes it so versatile and so popular and widely used is that we're learning essentially an arbitrary function that interpolates the data, but then tries to smooth the data as well. So we're interpolating the data because here I created a noise-free dataset. I'm moving through each data point. If there was a lot of noise in my dataset, I would be kind of doing a running, smoothing function through the dataset. And now we can learn arbitrary functions as defined by the dataset to make our prediction.