MachineLearning-Lecture08

Instructor (Andrew Ng):Okay. Good morning. Welcome back. If you haven't given me the homework yet, you can just give it to me at the end of class. That's fine. Let's see. And also just a quick reminder – I've actually seen project proposals start to trickle in already, which is great. As a reminder, project proposals are due this Friday, and if any of you want to meet and chat more about project ideas, I also have office hours immediately after lecture today. Are there any questions about any of that before I get started today? Great.

Okay. Welcome back. What I want to do today is wrap up our discussion on support vector machines and in particular we'll also talk about the idea of kernels and then talk about [inaudible] and then I'll talk about the SMO algorithm, which is an algorithm for solving the optimization problem that we posed last time.

To recap, we wrote down the following context optimization problem. All this is assuming that the data is linearly separable, which is an assumption that I'll fix later, and so with this optimization problem, given a training set, this will find the optimal margin classifier for the data set that maximizes this geometric margin from your training examples.

And so in the previous lecture, we also derived the dual of this problem, which was to maximize this. And this is the dual of our primal [inaudible] optimization problem. Here, I'm using these angle brackets to denote inner product, so this is just XI transpose XJ for vectors XI and XJ. We also worked out the ways W would be given by sum over I alpha I YI XI.

Therefore, when you need to make a prediction of classification time, you need to compute the value of the hypothesis applied to an [inaudible], which is G of W transpose X plus B where G is that threshold function that outputs plus one and minus one. And so this is G of sum over I alpha I. So that can also be written in terms of inner products between input vectors X.

So what I want to do is now talk about the idea of kernels, which will make use of this property because it turns out you can take the only dependers of the algorithm on X is through these inner products. In fact, you can write the entire algorithm without ever explicitly referring to an X vector [inaudible] between input feature vectors. And the idea of a high kernel is as following – let's say that you have an input attribute. Let's just say for now it's a real number. Maybe this is the living area of a house that you're trying to make a prediction on, like whether it will be sold in the next six months.

Quite often, we'll take this feature X and we'll map it to a richer set of features. So for example, we will take X and map it to these four polynomial features, and let me acutely call this mapping Phi. So we'll let Phi of X denote the mapping from your original features to some higher dimensional set of features.

So if you do this and you want to use the features Phi of X, then all you need to do is go back to the learning algorithm and everywhere you see XI, XJ, we'll replace it with the inner product between Phi of XI and Phi of XJ. So this corresponds to running a support vector machine with the features given by Phi of X rather than with your original one-dimensional input feature X.

And in a scenario that I want to consider, sometimes Phi of X will be very high dimensional, and in fact sometimes Phi of X – so for example, Phi of X may contain very high degree polynomial features. Sometimes Phi of X will actually even be an infinite dimensional vector of features, and the question is if Phi of X is an extremely high dimensional, then you can't actually compute to these inner products very efficiently, it seems, because computers need to represent an extremely high dimensional feature vector and then take [inaudible] inefficient.

It turns out that in many important special cases, we can write down – let's call the kernel function, denoted by K, which will be this, which would be inner product between those feature vectors. It turns out there will be important special cases where computing Phi of X is computationally very expensive – maybe is impossible.

There's an infinite dimensional vector, and you can't compute infinite dimensional vectors. There will be important special cases where Phi of X is very expensive to represent because it is so high dimensional, but nonetheless, you can actually compute a kernel between XI and XJ. You can compute the inner product between these two vectors very inexpensively.

And so the idea of the support vector machine is that everywhere in the algorithm that you see these inner products, we're going to replace it with a kernel function that you can compute efficiently, and that lets you work in feature spaces Phi of X even if Phi of X are very high dimensional. Let me now say how that's done. A little bit later today, we'll actually see some concrete examples of Phi of X and of kernels. For now, let's just think about constructing kernels explicitly. This best illustrates my example.

Let's say you have two inputs, X and Z. Normally I should write those as XI and XJ, but I'm just going to write X and Z to save on writing. Let's say my kernel is K of X, Z equals X transpose Z squared. And so this is – right? X transpose Z – this thing here is X transpose Z and this thing is X transpose Z, so this is X transpose Z squared. And that's equal to that. And so this kernel corresponds to the feature mapping where Phi of X is equal to – and I'll write this down for the case of N equals free, I guess.

And so with this definition of Phi of X, you can verify for yourself that this thing becomes the inner product between Phi of X and Phi of Z, because to get an inner product between two vectors is – you can just take a sum of the corresponding elements of the vectors. You multiply them. So if this is Phi of X, then the inner product between Phi of X and Phi of Z will be the sum over all the elements of this vector times the corresponding elements of Phi of Z, and what you get is this one.

And so the cool thing about this is that in order to compute Phi of X, you need [inaudible] just to compute Phi of X. If N is a dimension of X and Z, then Phi of X is a vector of all pairs of XI XJ multiplied of each other, and so the length of Phi of X is N squared. You need order N squared time just to compute Phi of X.

But to compute K – to compute the kernel function, all you need is order N time, because the kernel function is defined as X transpose Z squared, so you just take the inner product between X and Z, which is order N time and you square that and you've computed this kernel function, and so you just computed the inner product between two vectors where each vector has N squared elements, but you did it in N square time.

Student: For any kernel you find for X and Z, does Phi exist for X and Z?

Instructor (Andrew Ng):Let me talk about that later. We'll talk about what is a valid kernel later. Please raise your hand if this makes sense. So let me just describe a couple of quick generalizations to this. One is that if you define KXZ to be equal to X transpose Z plus C squared, so again, you can compute this kernel in order N time, then that turns out to correspond to a feature vector where I'm just going to add a few more elements at the bottom where you add root 2. Let me read that. That was root 2 CX1 root 2 CX2 root 2 CX3 and C.

And so this is a way of creating a feature vector with both the monomials, meaning the first order terms, as well as the quadratic or the inner product terms between XI and XJ, and the parameter C here allows you to control the relative waiting between the monomial terms, so the first order terms, and the quadratic terms. Again, this is still inner product between vectors of length and square [inaudible] in order N time.

More generally, here are some other examples of kernels. Actually, a generalization of the one I just derived right now would be the following kernel. And so this corresponds to using all N plus DQZ features of all monomials. Monomials just mean the products of XI XJ XK. Just all the polynomial terms up to degree D and plus [inaudible] so on the order of N plus D to the power of D, so this grows exponentially in D.

This is a very high dimensional feature vector, but again, you can implicitly construct the feature vector and take inner products between them. It's very computationally efficient, because you just compute the inner product between X and Z, add C and you take that real number to the power of D and by plugging this in as a kernel, you're implicitly working in an extremely high dimensional computing space.

So what I've given is just a few specific examples of how to create kernels. I want to go over just a few specific examples of kernels. So let's you ask you more generally if you're faced with a new machine-learning problem, how do you come up with a kernel? There are many ways to think about it, but here's one intuition that's sort of useful. So given a set of attributes of X, you're going to use a feature vector of Phi of X and given a set of attributes Z, you're going to use an input feature vector Phi of Z, and so the kernel is computing the inner product between Phi of X and Phi of Z.

And so one intuition – this is a partial intuition. This isn't as rigorous intuition that it is used for. It is that if X and Z are very similar, then Phi of X and Phi of Z will be pointing in the same direction, and therefore the inner product would be large. Whereas in contrast, if X and Z are very dissimilar, then Phi of X and Phi of Z may be pointing different directions, and so the inner product may be small. That intuition is not a rigorous one, but it's sort of a useful one to think about.

If you're faced with a new learning problem – if I give you some random thing to classify and you want to decide how to come up with a kernel, one way is to try to come up with the function P of XZ that is large, if you want to learn the algorithm to think of X and Z as similar and small. Again, this isn't always true, but this is one of several intuitions. So if you're trying to classify some brand new thing – you're trying to classify [inaudible] or DNA sequences or something, some strange thing you want to classify, one thing you could do is try to come up with a kernel that's large when you want the algorithm to think these are similar things or these are dissimilar.

And so this answers the question of let's say I have something I want to classify, and let's say I write down the function that I think is a good measure of how similar or dissimilar X and Z are for my specific problem. Let's say I write down K of XZ equals E to the minus. Let's say I write down this function, and I think this is a good measure of how similar X and Z are. The question, then, is is this really a valid kernel? In other words, to understand how we can construct kernels – if I write down the function like that, the question is does there really exist some Phi such that KXZ is equal to the inner product?

And that's the question of is K a valid kernel. It turns out that there is a result that characterizes necessary and sufficient conditions for when a function K that you might choose is a valid kernel. I should go ahead show part of that result now.

Suppose K is a valid kernel, and when I say K is a kernel, what I mean is there does indeed exist some function Phi for which this holds true. Then let any set of points X1 up to XM be given. Let me define a matrix K. I apologize for overloading notation. K I'm going to use to denote both the kernel function, which is the function of X and Z as well as a matrix. Unfortunately, there aren't enough alphabets. Well, that's not true.

We need to find the kernel matrix to be an M-by-M matrix such that K subscript IJ is equal to the kernel function applied to two of my examples. Then it turns out that for any vector Z that's indimensional, I want you to consider Z transpose KZ. By definition of matrix multiplication, this is that, and so KIJ is a kernel function between XI and XJ, so that must equal to this. I assume that K is a valid kernel function, and so there must exist such a value for Phi. This is the inner product between two feature vectors, so let me just make that inner product the explicit.

I'm going to sum over the elements of this vector, and I'm going to use Phi XI subscript K just to denote the K element of this vector. Just rearrange sums. You get sum over K. This next set may look familiar to some of you, which is just – right? Therefore, this is the sum of squares and it must therefore be greater than or equal to zero. Do you want to

take a minute to look for all the steps and just make sure you buy them all? Oh, this is the inner product between the vector of Phi of XI and Phi of XJ, so the inner product between two vectors is the sum over all the elements of the vectors of the corresponding element.

Student:[Inaudible].

Instructor (Andrew Ng):Oh, yes it is. This is just A transpose B equals sum over K, AK BK, so that's just this. This is the sum of K of the K elements of this vector. Take a look at this and make sure it makes sense. Questions about this? So just to summarize, what we showed was that for any vector Z, Z transpose KZ is greater than or equal to zero, and this is one of the standard definitions of a matrix, the matrix K being posisemidefinite when a matrix K is posisemidefinite, that is, K is equal to zero.

Just to summarize, what was shown is that if K is a valid kernel – in other words, if K is a function for which there exists some Phi such that K of XI XJ is the inner product between Phi of XI and Phi of XJ. So if K is a valid kernel, we showed, then, that the kernel matrix must be posisemidefinite. It turns out that the converse [inaudible] and so this gives you a test for whether a function K is a valid kernel.

So this is a theorem due to Mercer, and so kernels are also sometimes called Mercer kernels. It means the same thing. It just means it's a valid kernel. Let K of XZ be given. Then K is a valid kernel – in other words, it's a Mercer kernel, i.e., there exists a Phi such that KXZ equals Phi of X transpose Phi of Z – if and only if for any set of M examples, and this really means for any set of M points. It's not necessarily a training set. It's just any set of M points you may choose. It holds true that the kernel matrix, capital K that I defined just now, is symmetric posisemidefinite.

And so I proved only one direction of this result. I proved that if it's a valid kernel, then K is symmetry posisemidefinite, but the converse I didn't show. It turns out that this is necessary and a sufficient condition. And so this gives you a useful test for whether any function that you might want to choose is a kernel.

A concrete example of something that's clearly not a valid kernel would be if you find an input X such that K of X, X – and this is minus one, for example – then this is an example of something that's clearly not a valid kernel, because minus one cannot possibly be equal to Phi of X transpose Phi of X, and so this would be one of many examples of functions that will fail to meet the conditions of this theorem, because inner products of a vector itself are always greater than zero.

So just to tie this back explicitly to an SVM, let's say to use a support vector machine with a kernel, what you do is you choose some function K of XZ, and so you can choose – and it turns out that function I wrote down just now – this is, indeed, a valid kernel. It is called the Galcean kernel because of the similarity to Galceans. So you choose some kernel function like this, or you may choose X transpose Z plus C to the D vector.

To apply a support vector machine kernel, you choose one of these functions, and the choice of this would depend on your problem. It depends on what is a good measure of one or two examples similar and one or two examples different for your problem. Then you go back to our formulation of support vector machine, and you have to use the dual formulation, and you then replace everywhere you see these things, you replace it with K of XI, XJ.

And you then run exactly the same support vector machine algorithm, only everywhere you see these inner products, you replace them with that, and what you've just done is you've taken a support vector machine and you've taken each of your feature vectors X and you've replaced it with implicitly a very high dimensional feature vector.

It turns out that the Galcean kernel corresponds to a feature vector that's infinite dimensional. Nonetheless, you can run a support vector machine in a finite amount of time, even though you're working with infinite dimensional feature vectors, because all you ever need to do is compute these things, and you don't ever need to represent these infinite dimensional feature vectors explicitly.

Why is this a good idea? It turns out – I think I started off talking about support vector machines. I started saying that we wanted to start to develop non-linear learning algorithms. So here's one useful picture to keep in mind, which is that let's say your original data – I didn't mean to draw that slanted. Let's say you have one-dimensional input data. You just have one feature X and R. What a kernel does is the following. It takes your original input data and maps it to a very high dimensional feature space.

In the case of Galcean kernels, an infinite dimensional feature space – for pedagogical reasons, I'll draw two dimensions here. So say [inaudible] very high dimensional feature space where – like so. So it takes all your data in R1 and maps it to R infinity, and then you run a support vector machine in this infinite dimensional space and also exponentially high dimensional space, and you'll find the optimal margin classifier – in other words, the classifier that separates your data in this very high dimensional space with the largest possible geometric margin.

In this example that you just drew anyway, whereas your data was not linearly separable in your originally one dimensional space, when you map it to this much higher dimensional space, it becomes linearly separable, so you can use your linear classifier to [inaudible] which data is not really separable in your original space. This is what support vector machines output nonlinear decision boundaries and in the entire process, all you ever need to do is solve complex optimization problems. Questions about any of this?

Student:[Inaudible] sigmer?

Instructor (Andrew Ng):Yeah, so sigmer is – let's see. Well, I was going to talk about [inaudible] later. One way to choose sigmer is save aside a small amount of your data and try different values of sigmer and train an SVM using, say, two thirds of your data. Try

different values of sigmer, then see what works best on a separate hold out cross validation set – on a separate set that you're testing.

Something about learning algorithms we talked about – locally [inaudible] linear aggressions [inaudible] bandwidth parameter, so there are a number of parameters to some of these algorithms that you can choose IDs by saving aside some data to test on. I'll talk more about model selection [inaudible] explicitly. Are there other questions?

Student: So how do you know that moving it up to high dimensional space is going to give you that kind of separation?

Instructor (Andrew Ng):Good question. Usually, you don't know [inaudible]. Sometimes you can know, but in most cases, you won't know [inaudible] actually going to linearly separable, so the next topic will be [inaudible], which is what [inaudible] SVMs that work even though the data is not linearly separable.

Student: If you tend linearly separated by mapping a higher dimension, couldn't you also just use [inaudible] higher dimension?

Instructor (Andrew Ng):So very right. This is a question about what to do if you can't separate it in higher dimensional space. Let me try to address that work with a discussion of [inaudible] soft margin SVMs. Okay.

Student: What if you run an SVM algorithm that assumes the data are linearly separable on data that is not actually linearly separable?

Instructor (Andrew Ng):You guys are really giving me a hard time about whether the data's linearly separable. It turns out this algorithm won't work if the data is not linearly separable, but I'll change that in a second and make it work. If I move on to talk about that, let me just say one final word about kernels, which is that I talked about kernels in a context of support vector machines, and the idea of kernels was what really made support vector machines a very powerful learning algorithm, and actually towards the end of today's lecture if I have time, I'll actually give a couple more [inaudible] examples of how to choose kernels as well.

It turns out that the idea of kernels is actually more general than support vector machines, and in particular, we took this SVM algorithm and we derived a dual, and that was what let us write the entire algorithm in terms of inner products of these. It turns out that you can take many of the other algorithms that you've seen in this class – in fact, it turns out you can take most of the linear algorithms we talked about, such as linear regression, logistic regression [inaudible] and it turns out you can take all of these algorithms and rewrite them entirely in terms of these inner products.

So if you have any algorithm that you can rewrite in terms of inner products, then that means you can replace it with K of XI XJ and that means that you can take any of theses

algorithms and implicitly map the features vectors of these very high dimensional feature spaces and have the algorithm still work.

The idea of kernels is perhaps most widely used with support vector machines, but it is actually more general than that, and you can take many of the other algorithms that you've seen and many of the algorithms that we'll see later this quarter as well and write them in terms of inner products and thereby kernalize them and apply them to infinite dimensional feature spaces. You'll actually get to play with many of these ideas more in the next problem set.

Let's talk about non-linear decision boundaries, and this is the idea of – it's called the L1 norm soft margin SVM. Machine only people sometimes aren't great at coming up with good names, but here's the idea. Let's say I have a data set. This is a linearly separable data set, but what I do if I have a couple of other examples there that makes the data non-linearly separable, and in fact, sometimes even if the data is linearly separable, maybe you might not want to.

So for example, this is a very nice data set. It looks like there's a great decision boundary that separates the two [inaudible]. Well, what if I had just one outlier down here? I could still linearly separate this data set with something like that, but I'm somehow letting one slightly suspicious example skew my entire decision boundary by a lot, and so what I'm going to talk about now is the L1 norm soft margin SVM, which is a slightly modified formulation of the SVM optimization problem.

They will let us deal with both of these cases – one where one of the data's just not linearly separable and two, what if you have some examples that you'd rather not get [inaudible] in a training set. Maybe with an outlier here, maybe you actually prefer to choose that original decision boundary and not try so hard to get that training example. Here's the formulation. Our SVM primal problem was to minimize one-half [inaudible] W squared.

So this is our original problem, and I'm going to modify this by adding the following. In other words, I'm gonna add these penalty terms, CIs, and I'm going to demand that each of my training examples is separated with functional margin greater than or equal to one minus CI, and you remember if this is greater than zero – was it two lectures ago that I said that if the function margin is greater than zero, that implies you classified it correctly. If it's less than zero, then you misclassified it.

By setting some of the CIs to be larger than one, I can actually have some examples with functional margin negative, and therefore I'm allowing my algorithm to misclassify some of the examples of the training set. However, I'll encourage the algorithm not to do that by adding to the optimization objective, this sort of penalty term that penalizes setting CIs to be large. This is an optimization problem where the parameters are WB and all of the CIs and this is also a convex optimization problem. It turns out that similar to how we worked on the dual of the support vector machine, we can also work out the dual for this optimization problem.

I won't actually do it, but just to show you the steps, what you do is you construct [inaudible] Alpha R, and I'm going to use Alpha and R to denote the [inaudible] multipliers no corresponding to this set of constraints that we had previously and this new set of constraints on the CI [inaudible] zero. This gives us a use of the [inaudible] multipliers. The [inaudible] will be optimization objective minus sum from plus CI minus – and so there's our [inaudible] optimization objective minus or plus Alpha times each of these constraints, which are greater or equal to zero.

I won't redivide the entire dual again, but it's really the same, and when you derive the dual of this optimization problem and when you simplify, you find that you get the following. You have to maximize [inaudible], which is actually the same as before. So it turns out when you derive the dual and simply, it turns out that the only way the dual changes compared to the previous one is that rather than the constraint that the Alpha [inaudible] are greater than or equal to zero, we now have a constraint that the Alphas are between zero and C.

This derivation isn't very hard, and you're encouraged to go home and try to do it yourself. It's really essentially the same math, and when you simply, it turns out you can simply the R of the [inaudible] multiplier away and you end up with just these constraints of the Alphas.

Just as an aside, I won't derive these, either. It turns out that – remember, I wrote down the [inaudible] conditions in the last lecture. The necessary conditions for something to be an optimal solution to constrain optimization problems. So if you used the [inaudible] conditions, it turns out you can actually derive conversions conditions, so we want to solve this optimization problem. When do we know the Alphas have converged to the global optimum?

It turns out you can use the following. I don't want to say a lot about these. It turns out from the [inaudible] conditions you can derive these as the conversion conditions for an algorithm that you might choose to use to try to solve the optimization problem in terms of the Alphas.

That's the L1 norm soft margin SVM, and this is the change the algorithm that lets us handle non-linearly separable data sets as well as single outliers that may still be linearly separable but you may choose not to separate [inaudible]. Questions about this? Raise your hand if this stuff makes sense at all. Great.

So the last thing I want to do is talk about an algorithm for actually solving this optimization problem. We wrote down this dual optimization problem with convergence criteria, so let's come up with an efficient algorithm to actually solve this optimization problem. I want to do this partly to give me an excuse to talk about an algorithm called coordinate assent, which is useful to do.

What I actually want to do is tell you about an algorithm called coordinate assent, which is a useful algorithm to know about, and it'll turn out that it won't apply in the simplest

form to this problem, but we'll then be able to modify it slightly and then it'll give us a very efficient algorithm for solving this [inaudible] optimization problem. That was the other reason that I had to derive the dual, not only so that we could use kernels but also so that we can apply an algorithm like the SMO algorithm.

First, let's talk about coordinate assent, which is another [inaudible] optimization algorithm. To describe coordinate assent, I just want you to consider the problem of if we want to maximize some function W, which is a function of Alpha one through Alpha M with no constraints. So for now, forget about the constraint that the Alpha [inaudible] must be between zero and C. Forget about the constraint that some of YI Alpha I must be equal to zero. Then this is the coordinate assent algorithm.

It will repeat until convergence and will do for I equals one to M. The [inaudible] of coordinate assent essentially holds all the parameters except Alpha I fixed and then it just maximizes this function with respect to just one of the parameters. Let me write that as Alpha I gets updated as [inaudible] over Alpha I hat of W Alpha one Alpha I minus one. This is really the fancy way of saying hold everything except Alpha I fixed. Just optimize W by optimization objective with respect to only Alpha I. This is just a fancy way of writing it.

This is coordinate assent. One picture that's kind of useful for coordinate assent is if you imagine you're trying to optimize a quadratic function, it really looks like that. These are the contours of the quadratic function and the minimums here. This is what coordinate assent would look like. These are my [inaudible] call this Alpha two and I'll call this Alpha one. My Alpha one Alpha two axis, and so let's say I start down here. Then I'm going to begin by minimizing this with respect to Alpha one. I go there. And then at my new point, I'll minimize with respect to Alpha two, and so I might go to someplace like that.

Then, I'll minimize with respect to Alpha one goes back to Alpha two and so on. You're always taking these axis-aligned steps to get to the minimum. It turns out that there's a modification to this. There are variations of this as well. The way I describe the algorithm, we're always doing this in alternating order. We always optimize with respect to Alpha one then Alpha two, then Alpha one, then Alpha two. What I'm about to say applies only in higher dimensions, but it turns out if you have a lot of parameters, Alpha one through Alpha M, you may not choose to always visit them in a fixed order.

You may choose which Alphas update next depending on what you think will allow you to make the most progress. If you have only two parameters, it makes sense to alternate between them. If you have higher dimensional parameters, sometimes you may choose to update them in a different order if you think doing so would let you make faster progress towards the maximum.

It turns out that coordinate assent compared to some of the algorithms we saw previously – compared to, say, Newton's method, coordinate assent will usually take a lot more steps, but the chief advantage of coordinate assent when it works well is that sometimes

the optimization objective W sometimes is very inexpensive to optimize W with respect to any one of your parameters, and so coordinate assent has to take many more iterations than, say, Newton's method in order to converge.

It turns out that there are many optimization problems for which it's particularly easy to fix all but one of the parameters and optimize with respect to just that one parameter, and if that's true, then the inner group of coordinate assent with optimizing with respect to Alpha I can be done very quickly and cause [inaudible]. It turns out that this will be true when we modify this algorithm to solve the SVM optimization problem. Questions about this? Okay.

Let's go ahead and apply this to our support vector machine dual optimization problem. It turns out that coordinate assent in its basic form does not work for the following reason. The reason is we have constrains on the Alpha Is. Mainly, what we can recall from what we worked out previously, we have a constraint that the sum of [inaudible] Y Alpha I must be equal to zero, and so if you fix all the Alphas except for one, then you can't change one Alpha without violating the constraint.

If I just try to change Alpha one, Alpha one is actually exactly determined as a function of the other Alphas because this was sum to zero. The SMO algorithm, by the way, is due to John Platt, a colleague at Microsoft. The SMO algorithm, therefore, instead of trying to change one Alpha at a time, we will try to change two Alphas at a time. This is called the SMO algorithm, in a sense the sequential minimal optimization and the term minimal refers to the fact that we're choosing the smallest number of Alpha Is to change at a time, which in this case, we need to change at least two at a time.

So then go ahead and outline the algorithm. We will select two Alphas to change, some Alpha I and Alpha J [inaudible] – it just means a rule of thumb. We'll hold all the Alpha Ks fixed except Alpha I and Alpha J and optimize W [inaudible] Alpha with respect to Alpha I and Alpha J subject to all the constraints. It turns out the key step which I'm going to work out is this one, which is how do you optimize W of Alpha with respect to the two parameters that you just chose to update and subject to the constraints? I'll do that in a second.

You would keep running this algorithm until you have satisfied these convergence criteria up to Epsilon. What I want to do now is describe how to do this [inaudible] – how to optimize W of Alpha with respect to Alpha I and Alpha J, and it turns out that it's because you can do this extremely efficiently that the SMO algorithm works well. The [inaudible] for the SMO algorithm can be done extremely efficiently, so it may take a large number of iterations to converge, but each iteration is very cheap.

Let's talk about that. So in order to derive that step where we update in respect to Alpha I and Alpha J, I'm actually going to use Alpha one and Alpha two as my example. I'm gonna update Alpha one and Alpha two. In general, this could be any Alpha I and Alpha J, but just to make my notation on the board easier, I'm going to derive the derivation for Alpha one and Alpha two and the general [inaudible] completely analogous.

On every step of the algorithm with respect to constraint, that sum over I Alpha I YI is equal to zero. This is one of the constraints we had previously for our dual optimization problem. This means that Alpha one Y1 plus Alpha two Y2 must be equal to this, to which I'm going to denote by Zeta. So we also have the constraint that the Alpha Is must be between zero and C. We had two constraints on our dual. This was one of the constraints. This was the other one.

In pictures, the constraint that the Alpha Is is between zero and C, that is often called the Bosk constraint, and so if I draw Alpha one and Alpha two, then I have a constraint that the values of Alpha one and Alpha two must lie within this box that ranges from zero to C. And so the picture of the algorithm is as follows. I have some constraint that Alpha one Y1 plus Alpha two Y2 must equal to Zeta, and so this implies that Alpha one must be equal to Zeta minus Alpha two Y2 over Y1, and so what I want to do is I want to optimize the objective with respect to this.

What I can do is plug in my definition for Alpha one as a function of Alpha two and so this becomes W of Alpha one must be equal to Zeta minus Alpha two Y2 over Y1, Alpha two, Alpha three and so on, and it turns out that because W is a quadratic function – if you look back to our earlier definition of W, you find it's a quadratic function in all the Alphas – it turns out that if you look at this expression for W and if you view it as just a function of Alpha two, you find that this is a one dimensional quadratic function of Alpha two if you hold Alpha three, Alpha four and so on fixed, and so this can be simplified to some expression of the form A Alpha two squared plus B Alpha two plus C.

This is a standard quadratic function. This is really easy to optimize. We know how to optimize – when did we learn this? This was high school or undergrad or something. You know how to optimize quadratic functions like these. You just do that and that gives you the optimal value for Alpha two. The last step with a Bosk constraint like this – just in pictures, you know your solution must lie on this line, and so there'll be some sort of quadratic function over this line, say, and so if you minimize the quadratic function, maybe you get a value that lies in the box, and if so, then you're done.

Or if your quadratic function looks like this, maybe when you optimize your quadratic function, you may end up with a value outside, so you end up with a solution like that. If that happens, you clip your solution just to map it back inside the box. That'll give you the optimal solution of this quadratic optimization problem subject to your solution satisfying this box constraint and lying on this straight line – in other words, subject to the solution lying on this line segment within the box.

Having solved the Alpha two this way, you can clip it if necessary to get it back within the box constraint and then we have Alpha one as a function of Alpha two and this allows you to optimize W with respect to Alpha one and Alpha two quickly, subject to all the constraints, and the key step is really just sort of one dequadratic optimization, which we do very quickly, which is what makes the inner loop of the SMO algorithm very efficient.

Student: You mentioned here that we can change whatever, but the SMO algorithm, we can change two at a time, so how is that [inaudible] understand that.

Instructor (Andrew Ng):Right. Let's say I want to change – as I run optimization algorithm, I have to respect the constraint that sum over I Alpha I YI must be equal to zero, so this is a linear constraint that I didn't have when I was talking about [inaudible] ascent. Suppose I tried to change just Alpha one. Then I know that Alpha one must be equal to the sum from I equals two to M Alpha I YI divided by Y1, right, and so Alpha one can actually be written exactly as a function of Alpha two, Alpha three and so on through Alpha M. And so if I hold Alpha two, Alpha three, Alpha four through Alpha M fixed, then I can't change Alpha one, because Alpha one is the final [inaudible].

Whereas in contrast, if I choose to change Alpha one and Alpha two at the same time, then I still have a constraint and so I know Alpha one and Alpha two must satisfy that linear constraint but at least this way I can change Alpha one if I also change Alpha two accordingly to make sure this satisfies the constraint.

Student:[Inaudible].

Instructor (Andrew Ng):So Zeta was defined [inaudible]. So on each iteration, I have some setting of the parameters, Alpha one, Alpha two, Alpha three and so on, and I want to change Alpha one and Alpha two, say. So from the previous iteration, let's say I had not validated the constraint, so that holds true, and so I'm just defining Zeta to be equal to this, because Alpha one Y1 plus Alpha two Y2 must be equal to sum from I equals [inaudible] to M of that, and so I'm just defining this to be Zeta.

Student:[Inaudible].

Instructor (Andrew Ng):On every iteration, you change maybe a different pair of Alphas to update. The way you do this is something I don't want to talk about. I'll say a couple more words about that, but the basic outline of the algorithm is on every iteration of the algorithm, you're going to select some Alpha I and Alpha J to update like on this board.

So that's an Alpha I and an Alpha J to update via some [inaudible] and then you use the procedure I just described to actually update Alpha I and Alpha J. What I actually just talked about was the procedure to optimize W with respect to Alpha I and Alpha J. I didn't actually talk about the [inaudible] for choosing Alpha I and Alpha J.

Student: What is the function MW?

Instructor (Andrew Ng):MW is way up there. I'll just write it again. W of Alpha is that function we had previously. W of Alpha was the sum over I – this is about solving the – it was that thing. All of this is about solving the optimization problem for the SVM, so this is the objective function we had, so that's W of Alpha.

Student:[Inaudible]? Exchanging one of the Alphas – optimizing that one, you can make the other one that you have to change work, right?

Instructor (Andrew Ng):What do you mean works?

Student: It will get farther from its optimal.

Instructor (Andrew Ng):Let me translate it differently. What we're trying to do is we're trying to optimize the objective function W of Alpha, so the metric of progress that we care about is whether W of Alpha is getting better on every iteration, and so what is true for coordinate assent and for SMO is on every iteration; W of Alpha can only increase. It may stay the same or it may increase. It can't get worse.

It's true that eventually, the Alphas will converge at some value. It's true that in intervening iterations, one of the Alphas may move further away and then closer and further and closer to its final value, but what we really care about is that W of Alpha is getting better every time, which is true.

Just a couple more words on SMO before I wrap up on this. One is that John Platt's original algorithm talked about a [inaudible] for choosing which values or pairs, Alpha I and Alpha J, to update next is one of those things that's not conceptually complicated but it's very complicated to explain in words.

I won't talk about that here. If you want to learn about it, go ahead and look up John Platt's paper on the SMO algorithm. The [inaudible] is pretty easy to read, and later on, we'll also posting a handout on the course homepage with some of a simplified version of this [inaudible] that you can use in problems. You can see some of the process readings in more details.

One other thing that I didn't talk about was how to update the parameter B. So this is solving all your Alphas. This is also the Alpha that allows us to get W. The other thing I didn't talk about was how to compute the parameter B, and it turns out that again is actually not very difficult. I'll let you read about that yourself with the notes that we'll post along with the next problems.

To wrap up today's lecture, what I want to do is just tell you briefly about a couple of examples of applications of SVMs. Let's consider the problem of Handler's Integer Recognition. In Handler's Integer Recognition, you're given a pixel array with a scanned image of, say, a zip code somewhere in Britain. This is an array of pixels, and some of these pixels will be on and other pixels will be off. This combination of pixels being on maybe represents the character one. The question is given an input feature vector like this, if you have, say, ten pixels by ten pixels, then you have a hundred dimensional feature vector, then [inaudible].

If you have ten pixels by ten pixels, you have 100 features, and maybe these are binary features of XB01 or maybe the Xs are gray scale values corresponding to how dark each

of these pixels was. [Inaudible]. Turns out for many years, there was a neuronetwork that was a champion algorithm for Handler's Integer Recognition. And it turns out that you can apply an SVM with the following kernel. It turns out either the polynomial kernel or the Galcean kernel works fine for this problem, and just by writing down this kernel and throwing an SVM at it, an SVM gave performance comparable to the very best neuronetworks.

This is surprising because support vector machine doesn't take into account any knowledge about the pixels, and in particular, it doesn't know that this pixel is next to that pixel because it's just representing the pixel intensity value as a vector. And so this means the performance of SVM would be the same even if you were to shuffle all the pixels around. [Inaudible] let's say comparable to the very best neuronetworks, which had been under very careful development for many years.

I want to tell you about one other cool example, which is SVMs are also used also to classify other fairly esoteric objects. So for example, let's say you want to classify protein sequences into different classes of proteins. Every time I do this, I suspect that biologists in the room cringe, so I apologize for that. There are 20 amino acids, and proteins in our bodies are made up by sequences of amino acids. Even though there are 20 amino acids and 26 alphabets, I'm going to denote amino acids by the alphabet A through Z with apologizes to the biologists.

Here's an amino acid sequence represented by a series of alphabets. So suppose I want to assign this protein into a few classes depending on what type of protein it is. The question is how do I construct my feature vector? This is challenging for many reasons, one of which is that protein sequences can be of different lengths. There are some very long protein sequences and some very short ones, and so you can't have a feature saying what is the amino acid in the 100th position, because maybe there is no 100th position in this protein sequence. Some are very long. Some are very short.

Here's my feature representation, which is I'm going to write down all possible combinations of four alphabets. I'm going to write down AAAA, AAAB, AAAC down to AAAZ and then AABA and so on. You get the idea. Write down all possible combinations of four alphabets and my feature representation will be I'm going to scan through this sequence of amino acids and count how often each of these subsequences occur. So for example, BAJT occurs twice and so I'll put a two there, and none of these sequences occur, so I'll put a zero there. I guess I have a one here and a zero there.

This very long vector will be my feature representation for protein. This representation applies no matter how long my protein sequence is. How large is this? Well, it turns out this is going to be in R20 to the four, and so you have a 160,000 dimensional feature vector, which is reasonably large, even by modern computer standards. Clearly, we don't want to explicitly represent these high dimensional feature vectors. Imagine you have 1,000 examples and you store this as double [inaudible]. Even on modern day computers, this is big.

It turns out that there's an efficient dynamic programming algorithm that can efficiently compute inner products between these feature vectors, and so we can apply this feature representation, even though it's a ridiculously high feature vector to classify protein sequences. I won't talk about the [inaudible] algorithm. If any of you have seen the [inaudible] algorithm for finding subsequences, it's kind of reminiscent of that. You can look those up if you're interested.

This is just another example of a cool kernel, and more generally, if you're faced with some new machine-learning problem, sometimes you can choose a standard kernel like a Galcean kernel, and sometimes there are research papers written on how to come up with a new kernel for a new problem. Two last sentences I want to say. Where are we now? That wraps up SVMs, which many people would consider one of the most effective off the shelf learning algorithms, and so as of today, you've actually seen a lot of learning algorithms.

I want to close this class by saying congrats. You're now well qualified to actually go and apply learning algorithms to a lot of problems. We're still in week four of the quarter, so there's more to come. In particular, what I want to do next is talk about how to really understand the learning algorithms and when they work well and when they work poorly and to take the tools which you now have and really talk about how you can use them really well. We'll start to do that in the next lecture. Thanks.

[End of Audio]

Duration: 78 minutes