Support Vector Machines

Andrew Ng (video tutorial from Coursera's "Machine Learning" class)

Transcript written* by *José Soares Augusto*, June 2012 (V1.0)

1 Optimization Objective

By now you've seen a range of different learning algorithms. The performance of many supervised learning algorithms will be pretty similar, and it is more often less important whether you use learning algorithm A, or learning algorithm B, than other things like: the amount of data you have to train the algorithms; your skills in applying these algorithms; the choice of features that you design to give to the learning algorithms; how you choose the regularization parameter; etc....

But there's one more algorithm that is very powerful and it is very widely used both within industry and in Academia, that's called the **Support Vector Machine (SVM)**. Compared to both the logistic regression (LR) and neural networks (NN), the SVM sometimes gives a cleaner and more powerful way of learning complex nonlinear functions. And so I'd like to take the next videos to talk about the SVM.

Later in this course, I will do a quick survey of the range of different supervised learning algorithms, just to very briefly describe them. And the SVM, given its popularity, will be the last of the supervised learning algorithms that I'll spend a significant amount of time with in this course.

As with our development of other learning algorithms, we are going to start by talking about the optimization objective.

In order to describe the SVM, I'm actually going to start with logistic regression and show how we can modify it a bit and get what is essentially the SVM.

So, in logistic regression we had our familiar form of the hypothesis

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}} \tag{1}$$

*With help from the English subtitles which are available in the "Machine Learning" site, `https://class.coursera.org/ml/lecture/`.

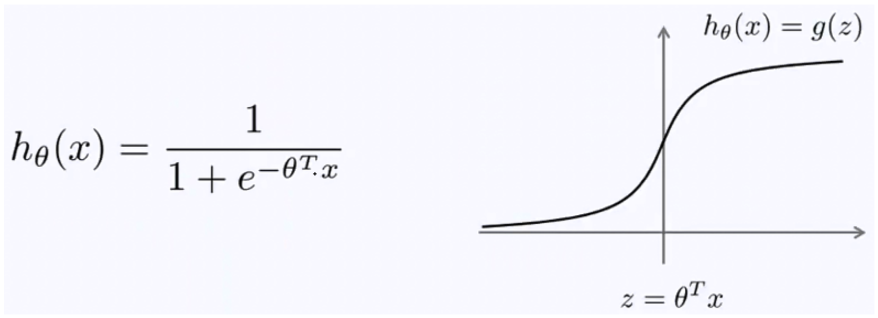


Figure 1: Equation and graph of the sigmoid hypothesis in logistic regression.

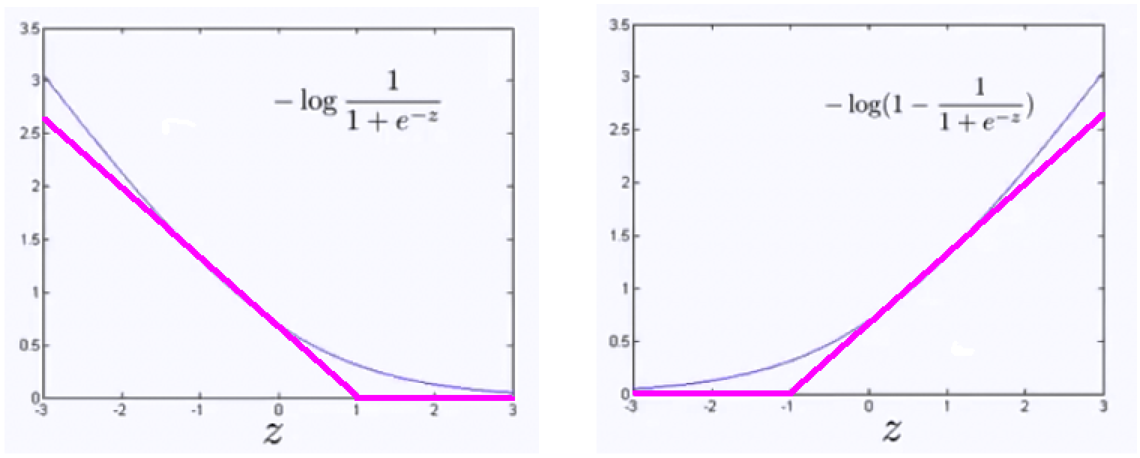


Figure 2: The two partial cost terms belonging to the cost function $J(\theta)$ for logistic regression: in the left, the positive case for $y = 1$ is $-\log\left(\frac{1}{1+e^{-z}}\right)$; in the right, the negative case for $y = 0$, is $-\log\left(1 - \frac{1}{1+e^{-z}}\right)$. The PWL magenta approximations are $\text{cost}_1(z)$ in the left picture, and $\text{cost}_0(z)$ in the right picture. Recall that $z = \theta^T x$.

which is based in the sigmoid activation function $g(z) = [1 + \exp(-z)]^{-1}$ shown in Fig. 1. And, in order to explain some of the math, I'm going to define again $z = \theta^T x$.

Now let's think about what we will like the logistic regression to do. If we have a positive example, where $y = 1$, in either the training set, or the test set, or the cross validation set, then we are sort of hoping that $h_\theta(x) \approx 1$ to correctly classify that example, and that means that $z = \theta^T x \gg 0$. And that's because it is when z is much bigger than 0, that is, when z is far to the right in the curve of logistic regression (Fig. 1), that $g(z)$'s output becomes close to 1.

Conversely, if we have a negative example, where $y = 0$, then what we are hoping for is that the hypothesis will output a value close to 0, and that corresponds to $z = \theta^T x \ll 0$ because it is when $z \ll 0$, in the far left of the curve of the sigmoidal hypothesis $g(z)$ in Fig. 1, that $g(z)$ will be outputting a value close to 0.

If you look at the **cost function of logistic regression**

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2 \tag{2}$$

you find that each example, (x, y), contributes the term (forgetting averaging with the $1/m$ weight)

$$-\left(y \log(h_\theta(x)) + (1 - y) \log(1 - h_\theta(x))\right)$$

to the overall cost function, $J(\theta)$. If I take the definition of my hypothesis (1), and plug it in the above cost term, what I get is that each training example contributes with the quantity

$$-y \log\left(\frac{1}{1 + e^{-\theta^T x}}\right) - (1 - y) \log\left(1 - \frac{1}{1 + e^{-\theta^T x}}\right) \tag{3}$$

(ignoring again the $1/m$ weight) to my overall cost function for logistic regression.

Now let's consider the two cases of when $y = 1$ and when $y = 0$.

In the first case, $y = 1$, only the first term $-y \log\left(\frac{1}{1+e^{-\theta^T x}}\right) = -\log\left(\frac{1}{1+e^{-\theta^T x}}\right)$ in the objective function quantity (3) matters, because $(1 - y)$ nulls the rightmost term.

Recall that $z = \theta^T x$. If we plot $-\log\left(\frac{1}{1+e^{-z}}\right)$, you get the continous curve shown in the left of Fig. 2, and thus we see that when $z = \theta^T x$ is large this value of z will give a very small contribution to the cost function, and this kind of explains why when logistic regression sees a positive example, where $y = 1$, it tries to set $\theta^T x$ to be very large because that corresponds to the partial cost term $-\log\left(\frac{1}{1+e^{-z}}\right)$ in the cost function being very small.

Now, to build the SVM here is what we are going to do. We are going to take the cost function $-\log\left(\frac{1}{1+e^{-z}}\right)$, and modify it a little bit.

The new partial cost function is going to be piecewise linear (PWL) with two segments[1]: we define $\text{cost}_1(z) = 0$ for $z \geq 1$, and then, from $z = 1$ to the left, I'm going to draw something that grows as a *straight line* similar to logistic regression (magenta segments in the left graph of Fig. 2). So, it's a pretty close approximation to the continous cost function used by logistic regression, except that it is now made out of two line segments. And don't worry too much about the slope of the straight line portion. It doesn't matter that much. This PWL function is shown also in the left picture of Fig. 3.

That's the new cost function we're going to use when $y = 1$, and you can imagine it should do something pretty similar to LR. But it turns out that this PWL shape will give the SVM computational advantage, because that will give us, later on, an easier optimization problem in the SVM, that will be easier to solve for classifying stock trades, and so on.

We just talked about the case of $y = 1$. The other case, the negative case, is when $y = 0$. In this case, if you look at the cost function (3) then only the term

$$-\log\left(1 - \frac{1}{1+e^{-z}}\right)$$

will apply, because the first term in (3) goes away when $y = 0$. The contribution to the cost function of the negative example is shown in the right of Fig. 2 as a function of $z = \theta^T x$.

For the SVM, once again, we're going to replace the continuous cost function associated to the negative example, $y = 0$, with a PWL approximation: a flat line for $z \leq -1$, and then it grows as a straight line for $z > -1$ (see the magenta lines in the right of Fig. 3).

So, let me give these two partial cost functions names. The PWL cost function for the $y = 1$ case is named $\mathbf{cost}_1(z)$; and the PWL cost function for $y = 0$ is named $\mathbf{cost}_0(z)$. The subscript just refers to the cost being 1 or 0.

Armed with these definitions, we are now ready to build the cost function for the SVM. Here is the cost function $J(\theta)$ that we have for logistic regression

$$\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\left(-\log h_\theta(x^{(i)})\right) + (1-y^{(i)})\left(-\log\left(1 - h_\theta(x^{(i)})\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \qquad (4)$$

(the minus sign that was in the left of (2) was moved to inside the summation here.)

For the SVM, what we are going to start to do is essentially plug $\text{cost}_1(z)$ and $\text{cost}_0(z)$ in the two terms inside the brackets, replacing $z = \theta^T x$ in their arguments:

$$\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\text{cost}_0(\theta^T x^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \qquad (5)$$

Now, by convention, for the SVM we actually write things slightly differently. First, we're going to get rid of the $1/m$ weights, and this just happens to be a slightly different convention that people use for the SVM compared to logistic regression. So, whether I solve this minimization problem with $1/m$ in front or not, I should end up with the same optimal value of θ.

[1]The name $\text{cost}_1(z)$ given to this partial cost is discussed below.

Here is what I mean. To give you a concrete example, suppose I had a minimization problem

$$\min_u [(u-5)^2+1]$$

the minimum of this happens to be $u = 5$. Now, if I want to take this same objective function and multiply it by 10, so my minimization problem is now

$$\min_u 10[(u-5)^2+1] = \min_u [10(u-5)^2+10]$$

well, the value of u that minimizes this is still $u = 5$, right?

In the same way, what I do by removing $1/m$ in (5) is to multiply my objective function by the constant m, and it doesn't change the value of θ that achieves the minimum. The cost function will become then:

$$\min_\theta \left[\sum_{i=1}^{m} y^{(i)}\text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\text{cost}_0(\theta^T x^{(i)})\right] + \frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2 \qquad (6)$$

There will be a second notational change in the SVM cost function. In logistic regression, the total cost is the "normal" cost A plus a regularization term R times λ

$$\min_\theta \; A + \lambda R \qquad \text{(logistic regression)}$$

and so, instead of prioritizing directly A, in LR we instead prioritize R by setting different values for the regularization parameter λ.

For the SVM, just by convention, we're going to use a different parameter for prioritizing, which by convention is called C. The SVM cost function that we are going to minimize has the shape

$$\min_\theta \; CA + R \qquad \text{(SVM)}$$

So, for logistic regression, if we send a very large value of λ, that means giving R (the regularization) a very high weight; in the SVM that corresponds to set C to be a very small value. So, this is just a different way of controlling the trade-off of how much we care about optimizing the first term A versus how much we care about optimizing the regularization term R.

And, if you want, you can think of this as the parameter C playing a role similar to $1/\lambda$.

So, we end up with our **final overall objective function for training the SVM**

$$\min_\theta C\sum_{i=1}^{m}\left[y^{(i)}\text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\text{cost}_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 \qquad (7)$$

and **when you minimize that function you get the parameters θ learned by SVM – your classification model**.

Finally, on light of how logistic regression worked, the SVM doesn't output the probability. Instead, there is a cost function which we minimize to get the parameters θ. And what the SVM does is it just makes the prediction of y being equal to 1 or equal to 0, directly. So, the hypothesis of the SVM, where y is predicted, is

$$h_\theta(x) = \begin{cases} 1 & \text{if } \theta^T x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

And so, after having learned the parameters θ, this is the form of the hypothesis produced by the SVM.

So, this was a mathematical definition of what a SVM does. In the next few videos, we'll try to get intuition about what this optimization objective leads to, what kind of hypothesis a SVM will learn and also talk about how to modify this just a little bit to learn complex, nonlinear functions.

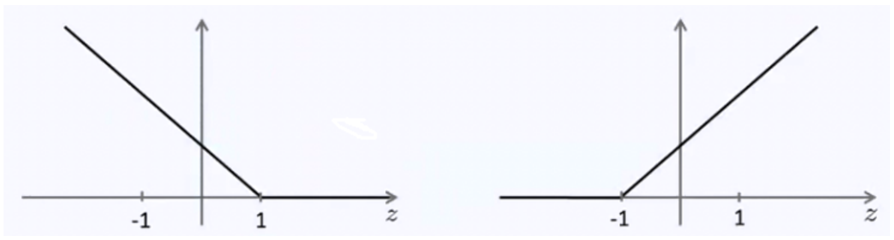


Figure 3: Piecewise linear partial costs for the SVM. In the left, the positive case for $y = 1$; in the right, the negative case for $y = 0$. Recall again that $z = \theta^T x$.

2 Large Margin Intuition

Sometimes people talk about SVMs as large margin classifiers. In this video I'd like to tell you what that means, and this will also give us a useful picture of what an hypothesis may look like.

Here's my cost function for the SVM

$$\min_\theta C \sum_{i=1}^{m} \left[y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 \tag{8}$$

and in the left of Fig. 3 is plotted the $\text{cost}_1(z)$ function that I used for positive examples, and on the right I've plotted the $\text{cost}_0(z)$ function for negative examples. The variable z is in the horizontal axis.

Now let's think about what it takes to make these cost functions small. If you have a positive example, $y = 1$, then $\text{cost}_1(z)$ is zero only when $z \geq 1$, that is, when $\theta^T x \geq 1$. And, conversely, if $y = 0$, this $\text{cost}_0(z)$ function is zero if $z = \theta^T x \leq -1$.

And this is an interesting property of the SVM.

In logistic regression, if you have a positive example, so if $y = 1$, then all we really need is that $\theta^T x \geq 0$, and that would mean we classify correctly, because if $\theta^T x \geq 0$ our hypothesis will predict one. And, similarly, if you have a negative example, $y = 0$, then you want $\theta^T x < 0$ and that will make sure we get the example classified right with LR.

But the SVM wants a bit more than that. Then we don't just have $\theta^T x$ just a little bit bigger than zero, in positive examples, but quite a lot bigger than 0, say perhaps bigger than 1! We want $\theta^T x \geq 1$.

And in negative examples, $y = 0$, I don't want only that $\theta^T x < 0$, I want it to be less than -1, that is, I want that $\theta^T x < -1$.

And so this builds in an extra safety margin factor to the SVM.

LR is something similar too, of course, but let's see what the consequences of this are, in the context of the SVM.

Concretely, what we'd like to do next is consider a case where we set this constant C to be a very large value, so let's imagine it's 100000, some huge number. Lets see what the SVM will do.

If C is very, very large, then when minimizing the SVM cost, or objective function, we're going to be highly motivated to choose θ values such that this first term

$$\sum_{i=1}^{m} \left[y^{(i)} \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T x^{(i)}) \right] \tag{9}$$

from (8) be close to zero.

So let's try to understand the optimization problem in the context of what would it take to make that first term in the objective function equal to 0, because, maybe, we'll set C to some huge constant. This should give us additional intuition about what sort of hypotheses a SVM learns.

So, we saw already that whenever you have a training example with a label of $y^{(i)} = 1$, if you want to make that first term involving $\text{cost}_1(z)$ to be zero, what you need is to find a value of θ such that $\theta^T x^{(i)} \geq 1$.

Similarly, whenever we have an example with label zero, i.e. $y^{(i)} = 0$, in order to make sure that $\text{cost}_0(z)$ is zero we need that $\theta^T x^{(i)} \leq -1$.

So, if we think of our optimization problem as choosing parameters θ such that (9) is close to zero, what we're left with is the following (approximate) optimization problem,

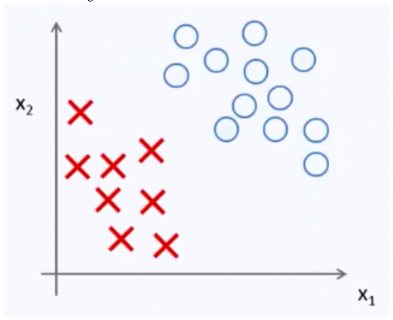
$$\min_\theta \; C \times 0 + \frac{1}{2}\sum_{j=1}^{n} \theta_j^2$$

but as the first term will be (almost equal to) 0 due to the large C, this minimization problem will be equivalent to the **constrained minimization problem**

$$\begin{aligned} \min_\theta \quad & \frac{1}{2}\sum_{j=1}^{n} \theta_j^2 \\ \text{s.t.} \quad & \theta^T x^{(i)} \geq 1 \quad \text{if } y^{(i)} = 1 \\ & \theta^T x^{(i)} \leq -1 \quad \text{if } y^{(i)} = 0 \end{aligned}$$

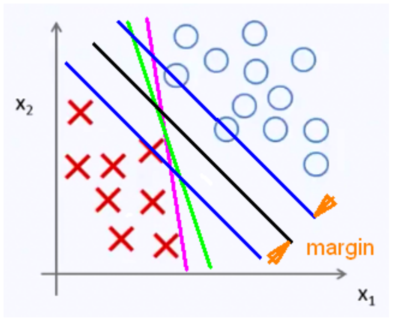
where the constraints are $\theta^T x^{(i)} \geq 1$ for positive examples, and $\theta^T x^{(i)} \leq -1$ for negative examples.

And it turns out that when you solve this optimization problem, as a function of the parameters θ, you get a very interesting decision boundary.



Concretely, if you look at a data set like this above, with positive and negative examples, this data is **linearly separable** and by that I mean that there exists a straight line, indeed many straight lines, that can separate the positive and negative examples perfectly.

For example, the magenta and green boundaries below separate the positive and negative examples, but somehow they don't look like very natural ones.

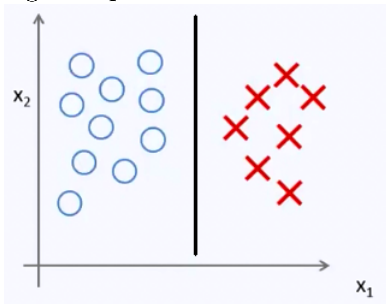


The SVMs will instead choose something like the decision boundary which is drawn in black. And that black boundary seems like a much better boundary then either of the magenta or green ones. The black line seems like a more robust separator, it does a better job of separating the positive and negative examples and, mathematically, this black boundary has a larger distance from the data points.

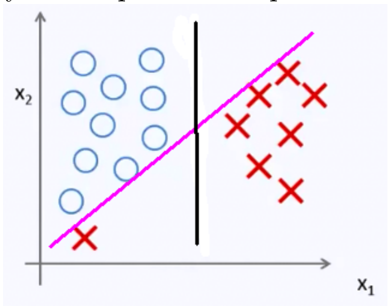
That distance is called the **margin**, and when I draw up two extra blue lines, we see that the black decision boundary has some larger minimum distance from any of the blue lines, what doesn't happen with the magenta and green lines that stick very close to the training examples, and that seems to do a less great job separating the positive and negative classes than my black line.

This **distance between the black and the blue lines is called the margin of the support vector machine** and this gives the SVM a certain robustness, because it tries to separate the data with a margin as large as possible.

So **the SVM is sometimes also called a large margin classifier** and this is actually a consequence of the optimization problem we wrote down previously. I know that you might be wondering how does that lead to this large margin classifier. I know I haven't explained that yet. And in a later video I'm going to sketch a little bit of the intuition about why that optimization problem gives us this large margin classifier. But this is a useful feature to keep in mind if you are trying to understand what are the source of hypothesis that the SVM will choose. That is, it is trying to separate the positive and negative examples with as big a margin as possible.



I want to say one last thing about large margin classifiers in this intuition, so we write out this large margin classification setting in the case of when C, the regularization constant, was very large, I think C is a hundred thousand or something. So, given a dataset like this one above, maybe the SVM will choose that black decision boundary that separates the possible examples with a large margin.



Now, the SVM is actually slightly more sophisticated than what this large margin view might suggest. And, in particular, if all you're doing is to use a large margin classifier, then your learning algorithms can be sensitive to outliers. So let's just add an extra positive example on the down left corner of the figure above. If we had one example like this, and still wanted to separate the positive and negative cases, maybe I'd end up learning a decision boundary like the magenta line. And it's really not clear that, based on a single outlier, it's actually a good idea to change the decision boundary from the black one over to the magenta one.

So, if the regularization parameter C were very large, then this is actually what the SVM will do, it will change the decision boundary from the black to the magenta line. But if C is reasonably small, or not too large, then you still end up with the black line decision boundary.

And, of course, if the data were not linearly separable, i.e. if you had some positive examples mixed with the negatives, and vice-versa, the SVM will also do the right thing. And so, this picture of the SVM as a large margin classifier is only for the case of when the regularization parameter C is very large. And just to remind you, C plays a role similar to $1/\lambda$ when λ was the regularization parameter we had previously in logistic regression. So, C very large is the same that λ very small, and in this case you end up with things like the magenta decision boundary.

In practice, in SVMs where C is not very, very large, we can do a better job ignoring the few outliers in the data set like here. And the SVM will do fine, and will do reasonable things even if your data is not linearly separable.

But when we will talk about bias and variance in the context of SVMs, which we'll do a little bit later, hopefully all of these tradeoffs involved in the regularization parameter will become clearer at that time.

So I hope that gives some intuition about **how the SVM functions as a large margin classifier** that tries to separate the data with a large margin. Technically, **this view is true only when the parameter C is very large**, which is a useful way to think about SVMs.

There was one missing step in this video which is: why is it that the optimization problem we wrote down actually leads to the large margin classifier?

I didn't do it in this video; in a later video I will sketch a little bit more of the mathematics behind that to explain the reasoning of how the optimization problem we wrote out results in a large margin classifier.

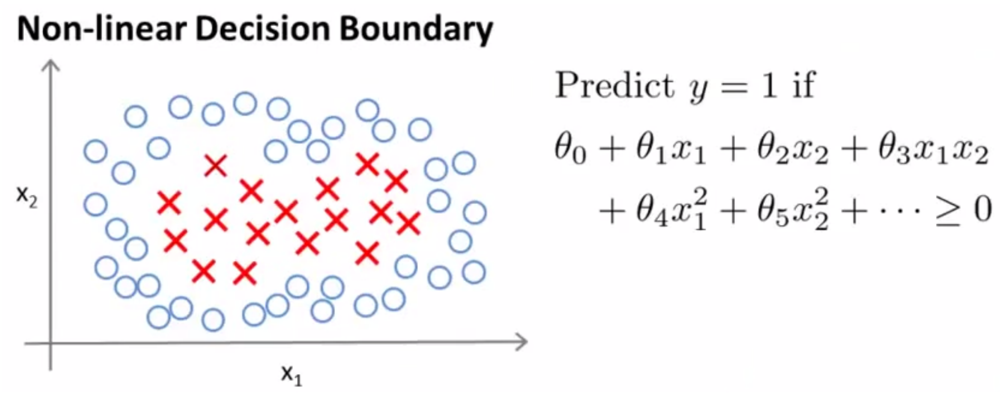


Figure 4: Dataset really asking for a nonlinear decision boundary.

3 Kernels I

In this video, I'd like to start adapting SVMs in order to develop complex nonlinear classifiers. The main technique for doing that is something called **kernels**. Let's see what these kernels are and how to use them.

If you have a training set that looks like Fig. 4, and you want to find a nonlinear decision boundary to distinguish the positive and negative examples, one way to do so is to come up with a set of complex polynomial features, such as to predict $y = 1$ if

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2 + \cdots \geq 0 \tag{10}$$

and predict $y = 0$, otherwise. And another way of writing this is

$$h_\theta(x) = \begin{cases} 1 \text{ if } \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2 + \cdots \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

We can think of a more general form of the hypothesis for computing a decision boundary

$$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 + \cdots$$

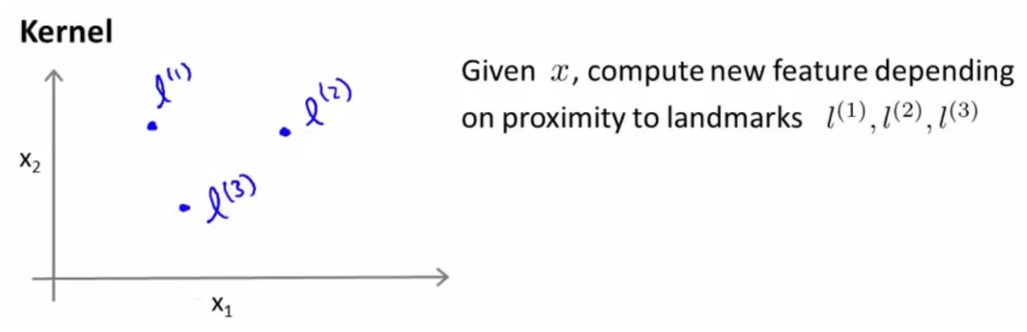
where I use a new notation f_1, f_2, f_3, ... , to denote this new sort of features that I'm computing.

Comparing with (10), $f_1 = x_1$, $f_2 = x_2$, $f_3 = x_1 x_2$, $f_4 = x_1^2$, $f_5 = x_2^2$, and so on.

And we've seen previously that coming up with these high order polynomials, which is one way to come up with lots of more features, that's usually not a good thing. The question is: is there a better sort of features f_1, f_2, f_3, and so on, than these high order polynomials?

This anguish about polynomials is because it's not clear that an high order polynomial is what we want in most of the situations. When we talked about computer vision, the input was an image with lots of pixels, and we saw how using high order polynomials becomes computationally very expensive because there are a lot of higher order "crossed" polynomial terms. So, is there a better choice of the features that we can use to plug into that sort of hypothesis pattern?

Here is one idea for how to define new features, f_1, f_2, f_3. I am going to define only three new features, but for real problems we can define a much larger number. In this graph of features (x_1, x_2) below, – and I'm going to leave the intercept x_0 out of this, – I'm going to just *manually* pick a few points marked as $l^{(1)}$, $l^{(2)}$ and $l^{(3)}$, and I'm going to call these points **landmarks**.


Kernel
$l^{(1)}$
$l^{(2)}$
$l^{(3)}$
x_2
x_1
Given x, compute new feature depending on proximity to landmarks $l^{(1)}, l^{(2)}, l^{(3)}$

I define my new features as follows: given a training example x, my first feature f_1 will be some measure of the similarity between my example x and my first landmark $l^{(1)}$

$$f_1 = \text{similarity}\left(x, l^{(1)}\right) = \exp\left(-\frac{||x - l^{(1)}||^2}{2\sigma^2}\right) \tag{11}$$

This notation $||w||$, called the **norm** of w, is the length of the vector w (explained in the SVM mathematics video). And so, $||x - l^{(1)}||^2$ is actually just the squared Euclidean distance between the point x and the landmark $l^{(1)}$.

And my second feature, f_2, is going to be

$$f_2 = \text{similarity}\left(x, l^{(2)}\right) = \exp\left(-\frac{||x - l^{(2)}||^2}{2\sigma^2}\right)$$

and the similarly f_3, the similarity between x and the landmark $l^{(3)}$, would be a similar formula.

And the mathematical term for the similarity function is **kernel function**. And the specific kernel formula I'm using here

$$\exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$

is actually called the **Gaussian kernel**. This is the way the terminology goes. I'm showing here the Gaussian kernel, but we'll see other examples of kernels.

For now, just think of these kernels as similarity functions. And so, instead of writing

$$\text{similarity}\left(x, l^{(i)}\right)$$

sometimes we also write

$$k\left(x, l^{(i)}\right)$$

So, let's see what these kernels actually do and why these sorts of similarity functions make sense.

Let's take my first landmark, $l^{(1)}$, which is one of those points in the figure. So, the similarity, or the kernel, between x and $l^{(1)}$ is given by (11). This similarity can also be written as

$$\exp\left(-\frac{||x - l^{(1)}||^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum_{j=1}^{n}\left(x_j - l_j^{(1)}\right)^2}{2\sigma^2}\right) \tag{12}$$

where the index j denotes the j-th component of the vector x, or of the vector landmark $l^{(1)}$. So, the numerator in the exponential is just the component-wise squared distance between x and $l^{(1)}$. And, again, I'm still ignoring the intercept term $x_0 = 1$ (recall x_0 is always equal to 1.)

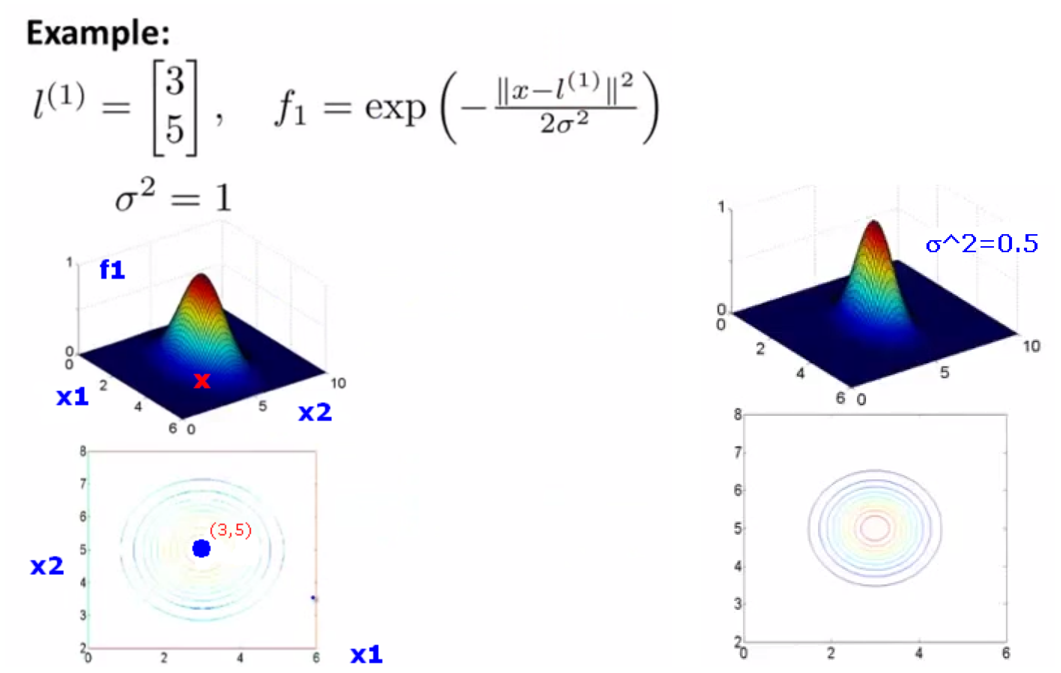


Figure 5: Example of Gaussian kernel feature, or similarity function. In the left, $\sigma^2 = 1.0$; in the right, $\sigma^2 = 0.5$. In both cases, the center "of the hat" is at $(x_1, x_2) = (3, 5)$.

So this is how you compute the kernel or similarity between x and a landmark $l^{(i)}$. Let's see what the similarity function does.

Suppose $x \approx l^{(1)}$, that is, x is close to the first landmark. Then the Euclidean distance formula in the numerator of (12) will be close to 0. So

$$f_1 = \exp\left(-\frac{(\sim 0)^2}{2\sigma^2}\right) \approx 1$$

where ~ 0 means "approximately zero". I use the approximation symbol because the distance between x and $l^{(1)}$ may not be exactly 0, but if x is close to the landmark, then $f_1 \approx 1$.

Conversely, if x is far from $l^{(1)}$ the first feature f_1 will be

$$f_1 = \exp\left(-\frac{(\text{large number})^2}{2\sigma^2}\right) \approx 0$$

thus f_1 is going to be close to 0.

So, what these features f_i do is they measure how similar x is from one of the landmarks, and the feature f_i is going to be close to 1 when x is close to one landmark, and is going to be close to 0 when x is far from all your landmarks.

On the previous graphic I drew three landmarks, $l^{(1)}$, $l^{(2)}$ and $l^{(3)}$. Each of these landmarks defines a new feature f_1, f_2 and f_3. That is, given the the training example x, we can now compute three new features, f_1, f_2 and f_3, given the three landmarks that I wrote.

But first let's look at this exponentiation function, let's look at this similarity function and plot in some figures to understand better what this really looks like.

In the example shown in Fig. 5 I have two features, x_1 and x_2. And let's say my first landmark, $l^{(1)}$, is at a location (3,5) and let's say $\sigma^2 = 1$, for now (left graphics). If I plot what this feature f_1 looks like, what I get is the hat-shaped landscape in Fig. 5. So, the vertical axis, the height of the surface, is the value of f_1 and on the horizontal axis are, x_1 (left axis) and x_2 (right axis). Given a certain training example, x (in red), the height above the surface at that point x shows the corresponding value of f_1.

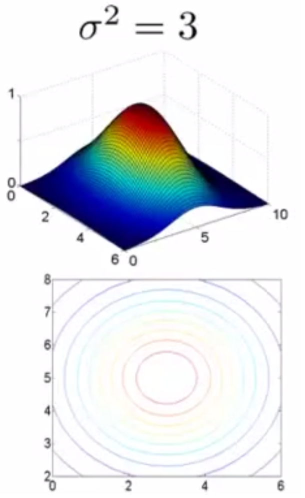
The graphics on the bottom-left of Fig. 5 is just a contour plot of the 3D surface shown in the middle. The center of the Gaussian, the landmark $l^{(1)} = (3, 5)$, is shown with the blue dot.

You notice that when $x = l^{(1)} = (3, 5)$ exactly, then the feature f_1 takes on the value 1, because that's at the maximum. If x moves away, and as x goes further away, then this feature f_1 takes on values that are close to 0. And so, this is really a feature, f_1, which measures how close x is to the first landmark, $l^{(1)}$, and it varies between 0 and 1 depending on how close x is to the first landmark, $l^{(1)}$.

Now let's show the effects of varying σ^2. So, σ^2 is the parameter of the Gaussian kernel and, as you vary it, you get slightly different effects.

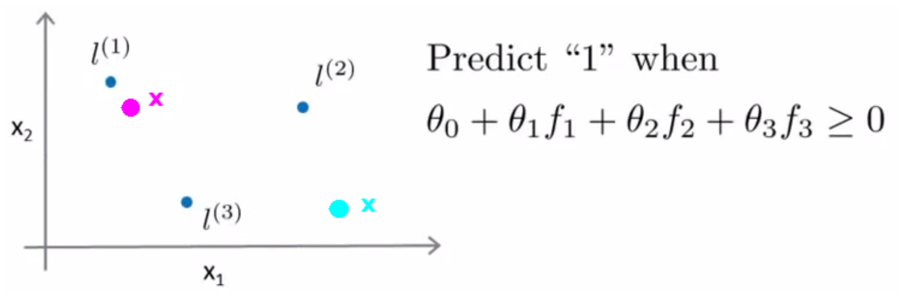
Let's set $\sigma^2 = 0.5$ and see what we get (Fig. 5 right). What you find is that the kernel looks similar, except that now the width of the bump becomes narrower. The contours, in the lower graphics, shrink a bit too. So, if $\sigma^2 = 0.5$, then as x moves away from $(3, 5)$ the feature f_1 falls to 0 much more rapidly than when $\sigma^2 = 1.0$.

And, conversely, if you increase σ^2 for $\sigma^2 = 3$, as you move away from the location (3,5), i.e. from $l^{(1)}$, the value of the feature f_1 falls away much more slowly (picture below).



So, given this definition of the features, let's see what sort of hypothesis we can learn. Given the training example x, we are going to compute the features f_1, f_2 and f_3, and the hypothesis is going to predict $y = 1$ when

$$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0 \tag{13}$$



$l^{(1)}$
x
$l^{(2)}$
x_2
$l^{(3)}$
x
x_1
Predict "1" when
$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$

For this particular example, let's say that I've already found the θ_j with a learning algorithm, and let's say that somehow I ended up with

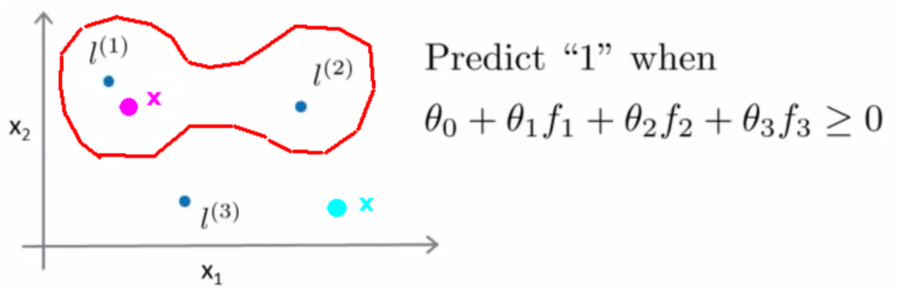
$$\theta_0 = -0.5, \quad \theta_1 = 1, \quad \theta_2 = 1, \quad \theta_3 = 0$$

And what I want to do is consider what happens if we have a training example x that takes the location of the *magenta* dot in the figure above: what would my hypothesis predict?

Well, If I look at $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3$, because my training example x is close to $l^{(1)}$ we have that $f_1 \approx 1$, and because x is far from $l^{(2)}$ and $l^{(3)}$ I have that $f_2 \approx 0$ and $f_3 \approx 0$. So, if I look at that formula (13), and if I plug in the values of the θ_j in it, I have $\theta_0 + \theta_1 \times 1 + \theta_2 \times 0 + \theta_3 \times 0 = -0.5 + 1 = 0.5 \geq 0$. So, at this point we're going to predict $y = 1$ because the hyphotesis is greater than zero.

Now let's take a different point x, drawn in a different color, in *cyan* say. If that were my training example, x, then if you make a similar computation you find that f_1, f_2 and f_3 are all going to be close to 0, and so $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \approx \theta_0 = -0.5 < 0$. And so, at this point drawn in cyan, we're going to predict $y = 0$.

And if you do this for a range of different points, be sure to convince yourself that if you have a training example that's close to $l^{(2)}$, say, then at that point we'll also predict $y = 1$. And, in fact, if we look around the red boundary below what we'll find is that for points near $l^{(1)}$ or $l^{(2)}$, inside the red boundary, we end up predicting $y = 1$. That doesn't happen near $l^{(3)}$ because $\theta_3 = 0$. And for points far away from the landmarks $l^{(1)}$ and $l^{(2)}$, outside the red closed boundary, we end up predicting that $y = 0$.


$l^{(1)}$
$l^{(2)}$
$l^{(3)}$
x
x
x_2
x_1
Predict "1" when
$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$

The decision boundary of this hypothesis would end up looking something like the red closed line, where inside the red decision boundary we would predict $y = 1$ and outside we would predict $y = 0$. And so this is how with this definition of the landmarks, and of the kernel function, we can learn pretty complex nonlinear decision boundaryies like that drawn in the figure, where we predict positive when we're close to either one of the two landmarks. And we predict negative when we're very far away from any of those two landmarks.

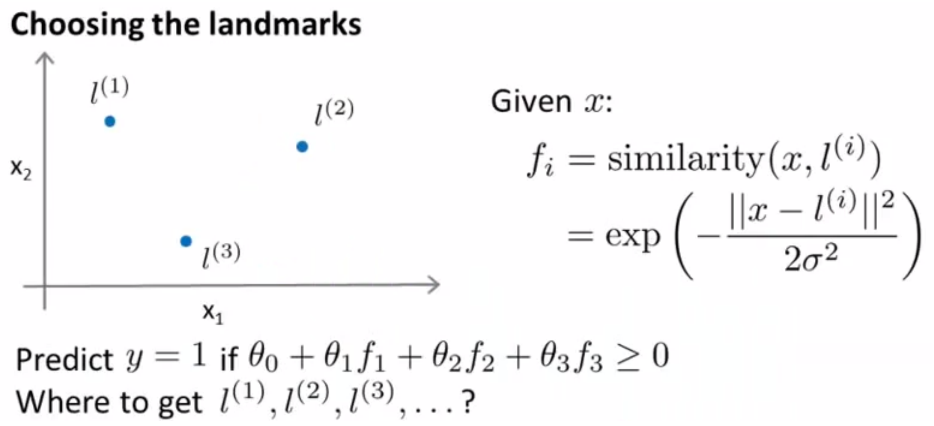
And so this is part of the idea of kernels, and how we use them with the SVM, which is we define these extra features using landmarks and similarity functions to learn more complex nonlinear classifiers.

So, hopefully that gives you a sense of the idea of kernels and how we could use it to define new features for the SVM. But there are a couple of questions that we haven't answered yet. One is, how do we get, how do we choose these landmarks? And another is, what other similarity functions, if any, can we use besides (11) or (12) which is called the Gaussian kernel?

In the next video we give answers to these questions and put everything together to show how SVMs with kernels can be a powerful way to learn very complex nonlinear functions.

4 Kernels II

In the last section we started to talk about the kernels idea and how it can be used to define new features for the SVM. Now I'd like to throw in some of the missing details and, also, to say a few words about how to use these ideas in practice, such as, for example, how to tackle the bias vs. variance trade-off in SVMs.



Choosing the landmarks

Given x:

$$f_i = \text{similarity}(x, l^{(i)}) = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$

Predict $y = 1$ if $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$

Where to get $l^{(1)}, l^{(2)}, l^{(3)}, \ldots$?

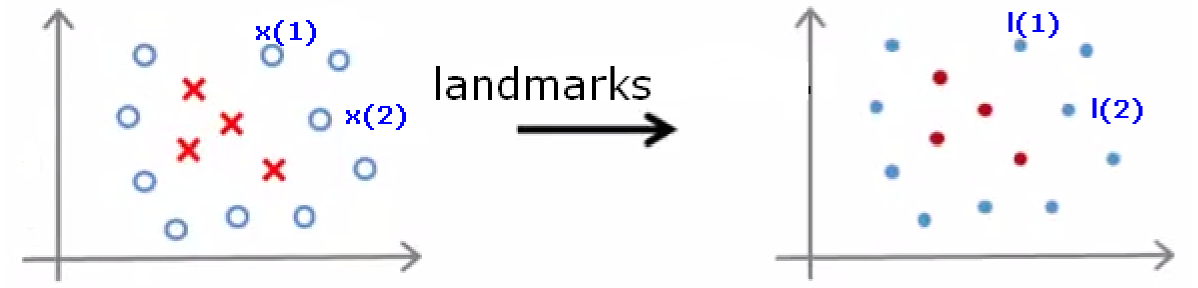
Before, I talked about the process of picking a few landmarks, $l^{(1)}$, $l^{(2)}$ and $l^{(3)}$, and that allowed us also to define the similarity function, also called the kernel. In this first example we have a similarity function which is a **Gaussian kernel**

$$\begin{aligned} f_i &= \text{similarity}(x, l^{(i)}) \\ &= \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right) \end{aligned}$$

and that allows us to build this form of a hypothesis function

$$\text{Predict } y = 1 \text{ if } \theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$$

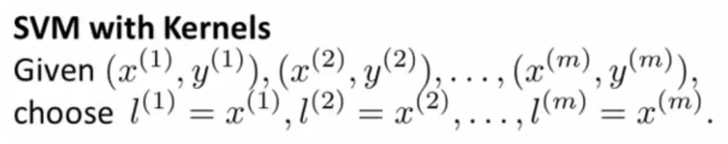
But where do we get these landmarks $l^{(1)}$, $l^{(2)}$ and $l^{(3)}$, from? And, for complex learning problems, maybe we want a lot more landmarks than just three of them that we might choose by hand.



So, in practice, given the machine learning problem we have some dataset of some positive and negative examples. So, this is the idea: we are just going to **put landmarks as exactly the same locations as the training examples**. So if I have one training example, $x^{(1)}$, well then I'm going to choose my first landmark, $l^{(1)}$, to be at exactly the same location of $x^{(1)}$; and for the training example $x^{(2)}$, we set the second landmark, $l^{(2)}$, to be in the location of $x^{(2)}$, and so on...

(I used red and blue dots just as illustration, the color of the dots in the figure is not important.)

I'm going to end up with m landmarks, $l^{(1)}$, $l^{(2)}$, $l^{(3)}$, ..., $l^{(m)}$, if I have m training examples, $x^{(1)}$, ..., $x^{(m)}$. One landmark for each of my training examples.



SVM with Kernels

Given $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$,
choose $l^{(1)} = x^{(1)}, l^{(2)} = x^{(2)}, \ldots, l^{(m)} = x^{(m)}$.

And this is nice, because my features are basically going to measure how close a new example x is to one of the things I saw in my training set.

So, just to write this outline a little more concretely, given m training examples, $(x^{(1)}, y^{(1)})$, $(x^{(2)}, y^{(2)})$, ... , $(x^{(m)}, y^{(m)})$, I'm going to **choose the locations of my landmarks to be exactly near the locations of my m training examples**

$$l^{(1)} = x^{(1)},\ l^{(2)} = x^{(2)},\ \ldots\ l^{(m)} = x^{(m)}$$

When you are given an example x, which can be something in the training, or cross-validation, or test sets, we are going to compute the features

$$\begin{aligned} f_1 &= \text{similarity}(x, l^{(1)}) \\ f_2 &= \text{similarity}(x, l^{(2)}) \\ &\ldots \end{aligned}$$

where $l^{(1)} = x^{(1)}$, $l^{(2)} = x^{(2)}$, and so on.

And these give me a feature vector, f

$$f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}$$

If we want, just by convention, we can add an extra feature f_0, which is always equal to 1. So, f_0 plays a role similar to what we had previously for x_0, which was our intercept.

So, for example, if we have a training example $(x^{(i)}, y^{(i)})$ the features we would compute for this training example would be as follows: given $x^{(i)}$, we will then map it to

$$\begin{aligned} f_1^{(i)} &= \text{sim}(x^{(i)}, l^{(i)}) \\ f_2^{(i)} &= \text{sim}(x^{(i)}, l^{(2)}) \\ &\ldots \\ f_m^{(i)} &= \text{sim}(x^{(i)}, l^{(m)}) \end{aligned}$$

where *sim()* is the abbreviation of similarity.

And somewhere in this list, at the i-th component, I will actually have one feature component which is $f_i^{(i)} = \text{sim}(x^{(i)}, l^{(i)})$ where $l^{(i)} = x^{(i)}$. And so, $f_i^{(i)}$ is just going to be the similarity between $x^{(i)}$ and itself, and if you're using the Gaussian kernel this is actually equal to $exp(-0) = 1$.

So, one of my features for this training example is going to be equal to 1. And I predict $y = 1$.

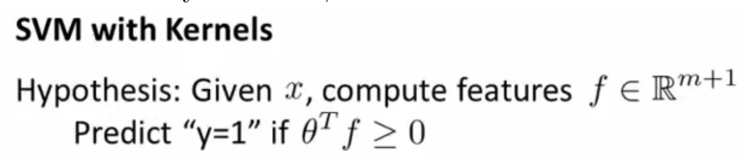
I can take all of these m features and group them into a feature vector. So, instead of representing my example using $x^{(i)} \in \mathbb{R}^{n+1}$ (or $\in \mathbb{R}^n$, depending on whether or not we use $f_0^{(i)} = 1$), we can now instead represent the training example $x^{(i)}$ using the feature vector $f^{(i)}$

$$f^{(i)} = \begin{bmatrix} f_0^{(i)} \\ f_1^{(i)} \\ f_2^{(i)} \\ \vdots \\ f_m^{(i)} \end{bmatrix}$$

and note that usually we'll also add this feature $f_0^{(i)} = 1$ to the vector $f^{(i)}$.

And so this gives me my new feature vector, $f^{(i)}$, with which I represent my training example.

So, given these kernels and similarity functions, here's how we use a SVM.



SVM with Kernels

Hypothesis: Given x, compute features $f \in \mathbb{R}^{m+1}$

Predict "y=1" if $\theta^T f \geq 0$

If you already have learned a set of parameters θ, if you are given a value of x and you want **to make a prediction, what you do is compute the feature vector** $f \in \mathbb{R}^{m+1}$. And we have m here, because we have m training examples, and thus m landmarks, and what we do is we predict "y=1" if $\theta^T f \geq 0$. But

$$\theta^T f = \theta_0 f_0 + \theta_1 f_1 + \cdots + \theta_m f_m$$

and so my parameter vector $\theta \in \mathbb{R}^{m+1}$. And we have m here because the number of landmarks, and the number of features f_i are equal to the training set size, m.

So that's how you make a prediction if you already have a setting for the parameters vector θ. How do you get θ? Well, you get it by using the SVM learning algorithm and, specifically, **what you do for training is you would solve this next minimization problem** (similar to (7), but with $x^{(i)}$ replaced by $f^{(i)}$)

$$\min_\theta C \sum_{i=1}^{m} \left[y^{(i)} \text{cost}_1(\theta^T f^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T f^{(i)}) \right] + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 \tag{14}$$

Only now, instead of making predictions using $\theta^T x^{(i)}$, using our original features $x^{(i)}$, instead we are using $\theta^T f^{(i)}$ to make a prediction on the i-th training example, and it's by solving this minimization problem that you get the parameters for your SVM.

And one last detail is that $n = m$, because the number of features is the same as the number of training examples. So, the regularization sum can be just written as $\frac{1}{2} \sum_{j=1}^{m} \theta_j^2$.

And then we have $m + 1$ features, with the $+1$ coming from the intercept θ_0, but we still do not regularize the parameter θ_0 which is why the sum $\sum_{j=1}^{m}$ runs from 1 (instead of 0) through m.

So that's the SVM learning algorithm.

That's one mathematical detail aside that I should mention: in the way the SVM is implemented, the regularization is actually done a little bit differently. You don't really need to know about this last detail in order to use SVMs, and in fact the term $\frac{1}{2} \sum_{j=1}^{m} \theta_j^2$ should give you all the intuitions that you should need. But in the way the SVM is implemented in practice, another way to write the sum is

$$\sum_{j=1}^{m} \theta_j^2 = \theta^T \theta$$

if we ignore the parameter θ_0, that is, using here $\theta = [\theta_1 \; \theta_2 \; \cdots \; \theta_m]^T$.

And what most SVM implementations do is to actually replace $\theta^T \theta$ with $\theta^T M \theta$, where the matrix M inside this product depends on the kernel you use. So the SVM optimization function is

$$\min_\theta C \sum_{i=1}^{m} \left[y^{(i)} \text{cost}_1(\theta^T f^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T f^{(i)}) \right] + \frac{1}{2} \theta^T M \theta \tag{15}$$

And so, this gives us a slightly different distance metric. Instead of minimizing exactly $||\theta||^2$ squared, it means that we minimize something slightly similar to it that depends on the kernel. But this is kind of a mathematical detail. That allows the SVM software to run much more efficiently.

The reason the SVM does this is because it allows it to scale to much bigger training sets. Because, for example, if you have a training set with 10000 training examples then, by the way we define landmarks, we end up with 10000 landmarks. And so θ becomes a 10000-dimensional vector. And maybe that works, but when m becomes really, really big, if m were say 50000 or 100000 then, for all of these parameters, solving the minimization problem can become very expensive for the SVM optimization software.

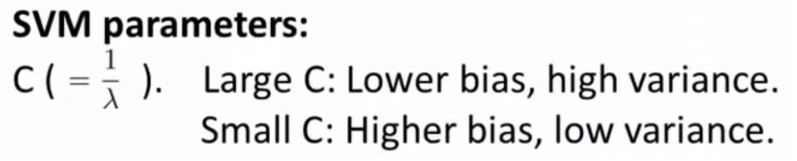
So this is kind of a mathematical detail, which again you really don't need to know about. It actually modifies that last term a little bit to optimize something slightly different than $||\theta||^2$. But you can feel free to think of this as a kind of an implementational detail that does change the objective function a bit, but is done primarily for reasons of computational efficiency, so usually you don't really have to worry about this.

And by the way, in case you're wondering why we don't apply the kernel's idea to other algorithms as well, like LR, it turns out that you can actually apply the kernel's idea and define the sets of features using landmarks and so on for logistic regression. But the computational tricks that apply for SVMs don't generalize well to other algorithms like LR. Using kernels with LR is going to be very slow, whereas, because of computational tricks, like that embodied in the regularization with $\theta^T M \theta$ and the details of how the SVM software is implemented, SVMs and kernels tend go particularly well together. Logistic regression and kernels, you can do it but this would run very slowly. And it won't be able to take advantage of advanced optimization techniques that people have figured out for the particular case of running a SVM with a kernel.

But, all this we just talked about pertains only to how you actually implement software to minimize the cost function. I will say more about that in the next video, but you really don't need to know about how to write software to minimize this cost function because you can find very good off-the-shelf software for doing so.

And just as I wouldn't recommend writing code to invert a matrix or to compute a square root, I actually do not recommend writing software to minimize the cost function (15) yourself, but instead to use off-the-shelf software packages that people have developed, which already embody these numerical optimization tricks, so you don't really have to worry about them.

But one other thing that is worth knowing about when you're applying a SVM, is how do you choose the parameters of it? And the last thing I want to do in this video is say a little word about the bias and variance tradeoffs when using a SVM.



SVM parameters:

C ($= \frac{1}{\lambda}$). Large C: Lower bias, high variance.

Small C: Higher bias, low variance.

When using an SVM, one of the things you need to choose is the parameter C in the optimization objective in (14) or (15), and you recall that C played a role similar to $1/\lambda$, where λ was the regularization parameter we had for logistic regression.

So, if you have a large value of C, this corresponds to what we had back in LR of a small value of λ, meaning not using much regularization, and if you do that you tend to have a hypothesis with lower bias and higher variance.

Whereas if you use a smaller value of C then this corresponds to when we are using LR with a large value of λ, and that corresponds to bet in strong regularization and in a hypothesis with higher bias and lower variance.

And so, the hypothesis with large C has a higher variance, and is more prone to overfitting, whereas the hypothesis with small C has higher bias and is thus more prone to underfitting.

So this parameter C is one of the parameters we need to choose. The other one is the parameter σ^2, which appears in the Gaussian kernel.

σ^2 Large σ^2: Features f_i vary more smoothly.
Higher bias, lower variance.

So, if in the Gaussian kernel σ^2 is large, then if in the similarity function

$$f^{(i)} = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$

if I have only one feature, $x^{(1)}$, if I have a landmark $l^{(i)}$ at a given location (Fig. 6 in the top), then the Gaussian kernel would tend to fall off relatively slowly, and so also my feature $f^{(i)}$, and so this would be a smooth function that varies smoothly, and so this will give you a hypothesis with higher bias and lower variance, because if the Gaussian kernel falls off smoothly you tend to get a hypothesis that varies slowly, or varies smoothly, as you change the input x.

Small σ^2: Features f_i vary less smoothly.
Lower bias, higher variance.

Whereas in contrast, if σ^2 was small (bottom of Fig. 6) my Gaussian kernel, my similarity function, will vary more abruptly. And in both cases I'd pick out 1 as the maximum value. And so, if σ^2 is small, then my features vary less smoothly, there are higher slopes and higher derivatives. And using this you end up fitting hypothesis of lower bias and you can have higher variance.

And if you look at this week's programming exercise, you actually get to play around with some of these ideas yourself and see these effects yourself.

So, that was the SVM with the kernels algorithm. And hopefully this discussion of bias and variance will give you some sense of how you can expect this algorithm to behave as well.

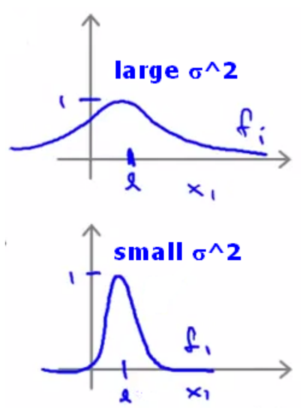


Figure 6: Effect of σ^2 in the Gaussian kernel.

5 Using An SVM

So far we've been talking about SVMs in a fairly abstract level. In this video I'd like to talk about what you actually need to do in order to run or use an SVM. The SVM poses a particular optimization problem. I really do not recommend writing your own software to do the training and solve for the parameter's θ yourself. So, just as essentially none of us would write code ourselves to invert a matrix, and so on, and we just call some library function to do that, in the same way the software for solving the SVM optimization problem is very complex, and there have been researchers that have been doing essentially numerical optimization research for many years to come up with good software libraries and good software packages to do this.

And I then strongly recommend just using one of the highly optimized software libraries rather than trying to implement something yourself. And there are lots of good software libraries out there. The two that I happen to use most often are the `liblinear` and the `libsvm`, but there are really lots of good software libraries; you can think of many of the major programming languages to code the SVM learning algorithm.

Even though you shouldn't be writing your own SVM optimization software, there are a few things you need to do.

First is to come up with some choice of the parameter C. We talked before a little bit of the bias vs. variance properties of C.

Second, you also need to choose the kernel or the similarity function that you want to use. So, one choice might be if we decide *not to use any kernel*, and this is *also called a* **linear kernel**. That means

$$\text{"predict } y = 1 \text{ if } \theta^T x \geq 0\text{"}$$

i.e. if

$$\theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n \geq 0$$

This term, linear kernel, you can think of it as the version of the SVM that just gives you a standard linear classifier. It would be one reasonable choice for some problems, and there would be many software libraries available, like `liblinear`, which is one example of a software library that can train an SVM without using a kernel, which is also called a SVM with a linear kernel.

So, why would you want to do this? If you have a large number of features x_i, with $x \in \mathbb{R}^n$, if n is large, and m, the number of training examples, is small, maybe you want to just fit a linear decision boundary and not try to fit a very complicated nonlinear function because you might not have enough data. And you might risk overfitting. So, this would be one reasonable setting where you might decide to just not use a kernel, or to use what's called a linear kernel.

A second choice for the kernel that you might make is this **Gaussian kernel**

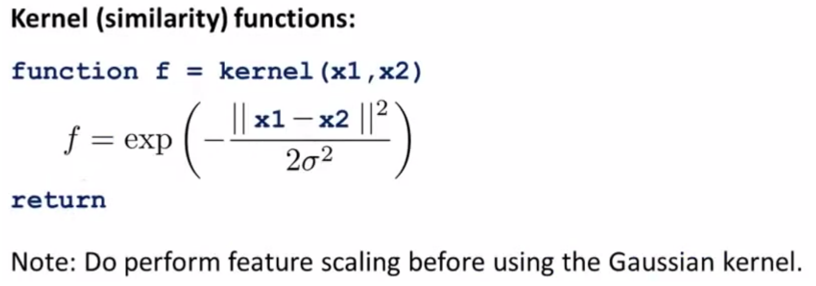
$$f_i = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$$

where $l^{(i)} = x^{(i)}$, and then the other choice you need to make is to choose σ^2. We already talked about the bias vs. variance tradeoff; if σ^2 is large, then you tend to have a higher bias, lower variance classifier, but if σ^2 is small, then you have a higher variance, lower bias classifier.

So, when would you choose a Gaussian kernel? Well, if your features are $x \in \mathbb{R}^n$, if n is small, and if m is large, right, so e.g. if $n = 2$ but you have a pretty large training set, then maybe you want to use a kernel to fit a more complex nonlinear decision boundary; and the Gaussian kernel would be a fine way to do this.

I'll say more towards the end of the video, a little bit more about when you might choose a linear kernel, a Gaussian kernel and so on.

But, concretely, if you decide to use a Gaussian kernel, then here's what you need to do. Depending on what SVM software package you use, it may ask you to implement a kernel function, or to implement the similarity function. So, if you're using an Octave or MATLAB implementation of an SVM, it may ask you to provide a function to compute a particular feature of the kernel



Kernel (similarity) functions:

```
function f = kernel(x1,x2)
```

$$f = \exp\left(-\frac{||\,\mathbf{x1} - \mathbf{x2}\,||^2}{2\sigma^2}\right)$$

```
return
```

Note: Do perform feature scaling before using the Gaussian kernel.

So, this is really computing f_i for one particular value of i, where f inside the function is just a single real number. What you need to do is to write a kernel function that takes as input a training example, or a test example; or it takes in some vector $x^{(i)}$ and one of the landmarks $l^{(i)}$, because the landmarks are really the training examples as well.

But what you need to do is write software that takes this input, `(x1, x2)`, and computes this sort of similarity function f between `x1` and `x2` and returns a real number.

And so, what some SVM packages do is they expect you to provide the `kernel(x1,x2)` function, that returns a real number, and then from there they will automatically generate all the features, and so automatically take x and map it to f_1, f_2, down to f_m, using the function you wrote, so they generate all the features and train the SVM from there.

Some SVM implementations will also include the Gaussian kernel (and a few other kernels as well) since it is probably the most common kernel. Gaussian and linear kernels are really the two most popular kernels by far.

Just one implementational note. **If you have features of very different scales, it is important to perform feature scaling before using the Gaussian kernel**. And here's why. Imagine computing $||x - l||^2$. If we define the vector $v = x - l$, then $||v||^2 = v_1^2 + v_2^2 + \cdots + v_n^2$, because here $x \in \mathbb{R}^n$ (or $x \in \mathbb{R}^{n+1}$, but I'm going to ignore x_0, so let's pretend $x \in \mathbb{R}^n$). So

$$||v||^2 = ||x - l||^2 = (x_1 - l_1)^2 + (x_2 - l_2)^2 + \cdots + (x_n - l_n)^2$$

What if your features, x_j, take on very different ranges of values?

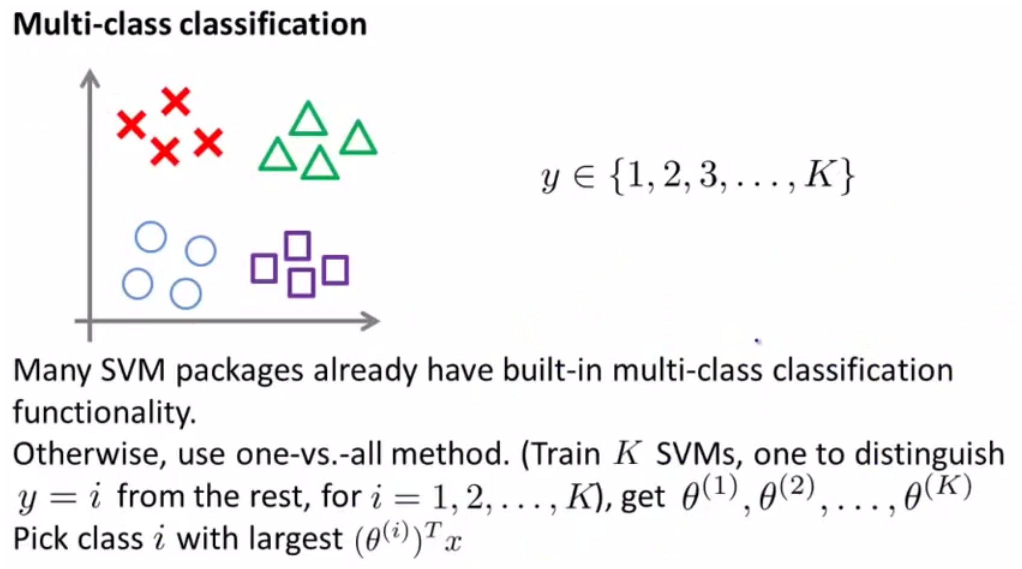
Take a housing prediction, for example, your data is about houses. If the first feature, x_1, the area of the house, is in the range of 1000's of ft^2, but your second feature, x_2, the number of bedrooms, is in the range of 1 to 5 bedrooms, then $(x_1 - l_1)^2$ is going to be huge – this could be like 1000^2, – whereas $(x_2 - l_2)^2$ is going to be much smaller – from 0 to 25. And if that's the case, then $||x - l||^2$ will be almost essentially dominated by the sizes of the houses and the number of bedrooms feature would be largely ignored in the hyphotesis.

As so, to avoid this problem and in order to make the SVM work well, do perform feature scaling. That will assure that the SVM gives comparable amount of attention to all of your different features, and not just to some or to only one.

And we stress that, when you try a SVM, chances are by far that you'll use one of the two most common kernels: the linear, or no kernel; or the Gaussian, that we just talked about.

And just one note of warning: **not all similarity functions `similarity(x,l)` you might come up with are valid kernels**. The Gaussian kernel, the linear kernel and other kernels that you sometimes will use, all of them need to satisfy a technical condition: it's called **Mercer's Theorem**.

And the reason of that constraint is because SVM algorithms or implementations have lots of clever numerical optimization tricks in order to solve for the parameters θ efficiently. Here we restrict our attention to kernels that satisfy this technical condition called Mercer's Theorem. And that makes sure that all SVM software packages can eventually use a large class of optimization techniques and get the training parameters θ very quickly.



Multi-class classification

$y \in \{1, 2, 3, \ldots, K\}$

Many SVM packages already have built-in multi-class classification functionality.
Otherwise, use one-vs.-all method. (Train K SVMs, one to distinguish $y = i$ from the rest, for $i = 1, 2, \ldots, K$), get $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(K)}$
Pick class i with largest $(\theta^{(i)})^T x$

Figure 7: Multiclass classification.

So, what most people end up doing is using either the linear or the Gaussian kernels, but there are a few other kernels that also satisfy Mercer's theorem and that you may run across other people using, although I personally end up using those other kernels very, very rarely, if at all.

One is the **polynomial kernel**. And for that the similarity between x and l has a lot of options to be defined. You can take $(x^T l)^2$ as it is one of the versions, it is one measure of how similar x and l are. If x and l are very close, then the inner product will tend to be large. This is a slightly unusual kernel that is not used that often, but you may run across some people using it.

That was one version of a polynomial kernel. Other versions are $(x^T l)^3$, or $(x^T l + 1)^3$, or $(x^T l + 5)^4$, and so the polynomial kernel actually has two parameters. One is the number, a, added to $x^T l$, and the other is the degree, d, of the polynomial. The general form of the polynomial kernel is thus

$$(x^T l + a)^d$$

So, the polynomial kernel almost always performs worse than the Gaussian kernel, and it is not used that much, but is just something that you may run across. Usually it is used only for data where x and l are strictly non negative, what ensures that the inner products $x^T l$ are never negative. And this captures the intuition that if x and l are very similar to each other, then the inner product between them will be large. They have some other properties as well, but people tend not to use them much.

And then, depending on what you're doing, there are other, sort of more esoteric, kernels that you may come across: there's a *string kernel*, sometimes used if your input data is text strings or other types of strings; there are the *chi-square kernel*, the *histogram intersection kernel*, and so on. These are sort of more esoteric kernels that you can use to measure similarity between different objects.

So, for example, if you're trying to do some sort of text classification problem, where the input x is a string. then maybe we want to find the similarity between two strings `sim(x,l)` using the string kernel, but I personally end up very rarely, if at all, using these more esoteric kernels: I think I might have used the chi-square kernel once in my life, the histogram kernel maybe once or twice in my life, and I've actually never used the string kernel myself.

But in case you've run across these in other applications, or if you've done a quick web search, you should have found definitions of these other kernels as well.

So, just two last details I want to talk about in this video.

One in multiclass classification (Fig. 7). So, you have 4 classes or, more generally, K classes, and you output some appropriate decision boundary between your multiple classes.

Most SVM packages already have built-in multiclass classification functionality. So, if you're using a package like that, you just use that functionality and that should work fine.

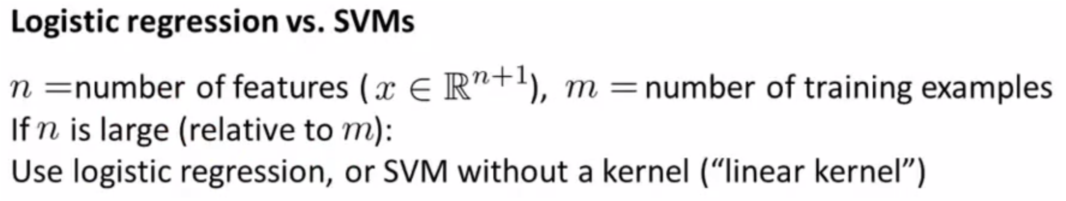
Otherwise, one way to do this is to use the one-versus-all method that we talked about when we were developing logistic regression. You train K instances of SVMs if you have K classes, each one to distinguish each of the classes from the rest. And this would give you K parameter vectors: $\theta^{(1)}$ which is trying to distinguish class $y = 1$ from all of the other classes; then you get the second vector, $\theta^{(2)}$, which is what you get when you have $y = 2$ as the positive class and all the others as negative classes; and so on, up to getting $\theta^{(K)}$, which is the parameter vector used for distinguishing the final class K from anything else.

This is exactly the same as the one-versus-all method we had for logistic regression, and we just predicted the class i with the largest

$$\left(\theta^{(i)}\right)^T x$$

For the more common cases, there is a good chance that whatever software package you use it already has built-in multiclass classification functionality, and so you don't need to worry about this result.

Finally, we developed SVMs starting off with LR and then modifying the cost function a little bit. The last thing I want to do in this video is just say a little bit about when you will use one of these two algorithms: logistic regression or the SVM.

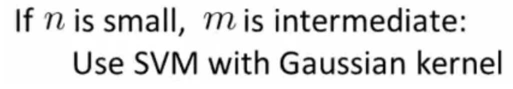


Logistic regression vs. SVMs

n =number of features ($x \in \mathbb{R}^{n+1}$), m =number of training examples
If n is large (relative to m):
Use logistic regression, or SVM without a kernel ("linear kernel")

So, n is the number of features and m is the number of training examples. When should we use one algorithm versus the other?

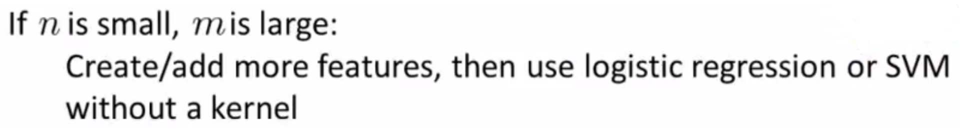
Well, suppose n is larger relative to your training set size, m. So, for example, suppose you have a text classification problem where the dimension of the feature vector is, maybe, 10000, and your training set size is maybe from 10 upto 1000. So, imagine a spam classification problem where you have 10000 features corresponding to 10000 words but you have, maybe, only from 10 upto 1000 training examples. So, if n is large relative to m, then what I would usually do is use LR or use SVM without a kernel, i.e. with a linear kernel. Because if you have so many features with a smaller training set a linear function will probably do fine, and you don't have really enough data to fit a complicated nonlinear function.



If n is small, m is intermediate:
Use SVM with Gaussian kernel

Now if is n is small and m is intermediate – what I mean by this is n is maybe anywhere from 1 upto 1000, and if the number of training examples is maybe anywhere from 10 upto 10000, maybe upto 50000 examples, but less than, say, a million – then often an SVM with a Gaussian kernel will work well.

We talked about this early, with the concrete example of a two dimensional training set. So, if $n = 2$, where you have a pretty large number of training examples, the Gaussian kernel will do a pretty good job separating positive and negative classes.


If n is small, m is large:
Create/add more features, then use logistic regression or SVM without a kernel

One third setting that's of interest is if n is small but m is large. So n again maybe in the range from 1 upto 1000, or could be larger, but m is, maybe 50000 and greater, upto millions. You have a very, very large training set size, right? So, if this is the case, then a SVM with a Gaussian Kernel will be somewhat slow to run: today's SVM packages, if you're using a Gaussian Kernel tend to struggle a bit. If you have, maybe, $m = 50000$ is okay, but if you have a million training examples, or even only 100000, with a massive value of m today's SVM packages are very, very good, but they can still struggle a little bit when you have a massive, massive training set size when using a Gaussian Kernel.

So, in that case what I would usually do is try to just manually create more features and then use logistic regression or an SVM without kernel – i.e., with the linear kernel.

You see LR and SVM without kernel paired together in several scenarios. There's a reason for that: logistic regression and SVM without kernel are really pretty similar algorithms and either LR or SVM without kernel will usually do pretty similar things and give pretty similar performance. Depending on implementation details, one may be more efficient than the other. But where one of these algorithms, LR or SVM without kernel, applies, the other one is likely to work pretty well as well.

But the power of the SVM is released when you use Gaussian kernels to learn complex nonlinear functions. That usually happens if n is small and m is intermediate. In this regime you have maybe up to 10000 examples, maybe up to 50000, and your number of features is reasonably large. That's a very common regime and that's a regime where a SVM with a Gaussian kernel will shine: you can do things that are much harder to do with logistic regression.

And where do neural networks fit in? Well, for all of these different regimes, a well designed NN is likely to work well as well. The one disadvantage, the one single reason that might sometimes exclude the NN is that, for some of these problems, the NN might be slower to train. If you have a very good SVM implementation package, that probably will run faster, quite a bit faster, than your NN.

And although we didn't show this earlier, it turns out that **the optimization problem in the SVM is a convex optimization problem** and so the good SVM optimization software packages will always find the global minimum, or something close to it. And so, for the SVM you don't need to worry about local optima. In practice, local optima aren't a huge problem for NNs, but this is one less thing to worry about if you're using an SVM. And depending on your problem, the neural network may be slower, especially in the regime where the SVM with the Gaussian kernel shines.

In case the guidelines I gave here seem a little bit vague, and if you're looking at some problems and the guidelines are a bit vague, and you're not entirely sure – should I use this algorithm or that algorithm? – that's actually okay.

When I face a machine learning problem, sometimes its actually just not clear whether what's the best algorithm to use. But, as you saw in earlier videos, really the algorithm does matter, but what often matters even more is things like how much data do you have, how skilled are you, how good are you at doing error analysis and debugging learning algorithms, at figuring out how to design new features and figuring out what other features to give your learning algorithms, and so on.

And often those things will matter more than if you are using LR or an SVM. But having said that, the SVM is still widely perceived as one of the most powerful learning algorithms, and it's a very effective way to learn complex nonlinear functions.

So, if you use SVMs, together with LR and NNs to speed-up learning algorithms, you're very well positioned to build state-of-the-art machine learning systems for a wide range of applications. The SVM is a very powerful tool to have in your arsenal, one that is used all over the place in Silicon Valley, in industry and in the Academia, to build high performance machine learning systems.

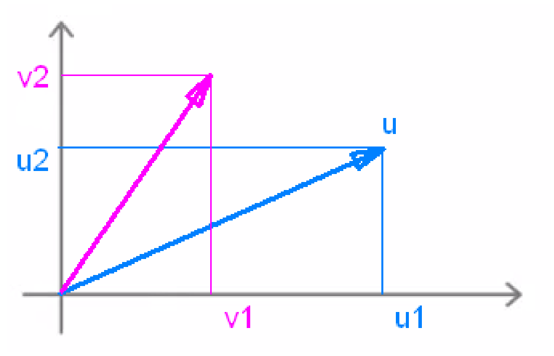


Figure 8: 2D vectors, v and u.

6 Mathematics Behind Large Margin Classification (Optional)

I'd like to tell you a bit about the mathematics behind large margin classification. This subject is optional, so please feel free to skip it. It may also give you better intuition about how the optimization problem of the SVM leads to large margin classifiers.

In order to get started, let's review a couple of properties of vector inner products. Let's say I have two two-dimensional (2D) vectors, u and v

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \qquad v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

Then let's see what their inner product $u.v = u^T v = v^T u$ looks like. I can plot the 2D vector u in a figure (blue arrow in Fig. 8) and show its coordinates, or components, u_1 and u_2.

The norm of the vector u is denoted by $||u||$. It is also called the Euclidean length, or simply lenght, of the vector u. And the Pythagoras' theorem says $||u|| = \sqrt{u_1^2 + u_2^2}$. Notice that $||u|| \in \mathbb{R}$ is a real number, is the length of this vector, of this arrow in Fig. 8.

The vector v is also in Fig. 8, drawn in magenta, as well as its components v_1 and v_2.

To compute the inner product between u and v we are going to use the orthogonal projection of v onto the vector u, the red line in the left of Fig. 9, and the length of that red line is p.

So, p is the length or magnitude of the projection of the vector v onto the vector u.

And it is possible to show that

$$u^T v = p \times ||u||$$

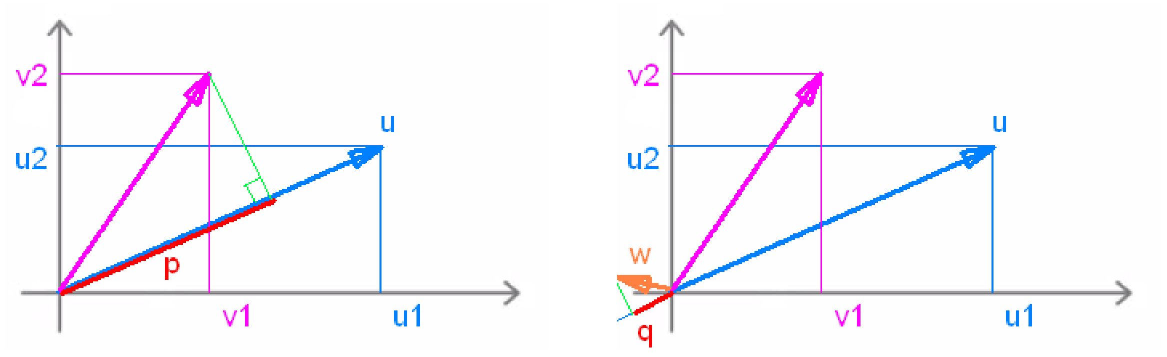


Figure 9: In the left, projection of v onto u shown in red. The value p is the lenght of the projection. In the right, q is the "negative" lenght of the projection of w on the vector u, because the angle between w and u is larger than 90 degrees.

So, this is one way to compute the inner product.

Another way is just use the formula

$$u^T v = u_1 v_1 + u_2 v_2$$

And, by the way, $u^T v = v^T u$. So if you were to do the same process in reverse, instead of projecting v onto u, you could project u onto v. And you should actually get the same number for the inner product, whatever that number is.

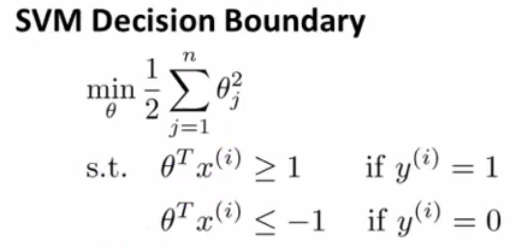
Note that $||u||$ and p are real numbers, and so $u^T v$ is the regular multiplication of two real numbers.

Just one last detail: p is actually a signed value, and it can either be positive or negative. So let me say what I mean by that. If u is the same vector as before and w is the orange vector in the right picture in Fig. 9, the angle between u and w is greater than 90 degrees. Then, if I project w onto u, what I get is the small red line with length q. And in this case, I will still have $u^T w = q \times ||u||$, except that in this example q will be negative.

In inner products of two vectors, if the angle between them is less than ninety degrees, such as in $p = u.v$, then p is a positive length as for that red line in the left Fig. 9, whereas if the angle is greater than 90 degrees, such as in $q = u.w$, then q will be the negative of the length of the little red line segment in the right in Fig. 9.

So that's how vector inner products work. We're going to use these properties of the vector inner product to try to understand the SVM optimization objective.

Here is the optimization objective for the SVM that we worked out earlier.



SVM Decision Boundary

$$\min_\theta \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

$$\text{s.t.} \quad \theta^T x^{(i)} \geq 1 \qquad \text{if } y^{(i)} = 1$$

$$\theta^T x^{(i)} \leq -1 \qquad \text{if } y^{(i)} = 0$$

I am going to make one simplification just to make the objective function easy to analyze, and what I'm going to do is ignore the intercept, that is, to make $\theta_0 = 0$. To make things easier to plot, I'm also going to set the number of features $n = 2$, so we have only two features, x_1 and x_2.

Now, let's look at the objective function of the SVM. With only two features it can be written as either of

$$\min_\theta \frac{1}{2} \left(\theta_1^2 + \theta_2^2\right)$$

$$\min_\theta \frac{1}{2} \left(\sqrt{\theta_1^2 + \theta_2^2}\right)^2$$

This odd $a = (\sqrt{a})^2$ equality is mathematically valid and is used for convenience. What you may notice then is that $\sqrt{\theta_1^2 + \theta_2^2} = ||\theta||$ where, obviously, $\theta = [\theta_0 \ \theta_1 \ \theta_2]^T$, and recall that I assume here that $\theta_0 = 0$.

So we can think just of $\theta = [\theta_1 \ \theta_2]^T$; the math works out either way, whether we include θ_0 in the vector θ or not.

And so, finally, this means that the optimization objective is

$$\min_\theta \frac{1}{2} ||\theta||^2$$

So all the SVM is doing is it's minimizing $||\theta||^2$.

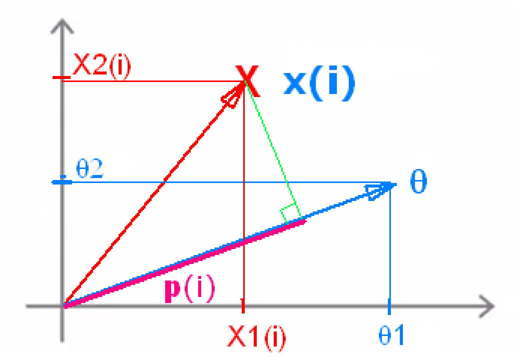


Figure 10: Meaning of $\theta^T x^{(i)}$.

Now let's look at the terms $\theta^T x^{(i)} \geq 1$ if $y = 1$ and $\theta^T x^{(i)} \leq -1$ if $y = 0$, and understand better what they're doing. Given the parameter vector θ and an example $x^{(i)}$, we conclude that $\theta^T x^{(i)}$ is their inner product.

So, let's say I look at just a single positive training example, $x^{(i)}$, drawn as the red cross in Fig. 10, and let's draw the corresponding (red) vector, and components $x_1^{(i)}$ and $x_2^{(i)}$. And now let's say we have a parameter vector θ plotted there as the blue vector, with components θ_1 and θ_2 as well.

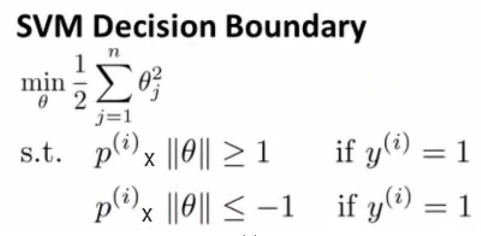
Using our earlier method to compute the inner product, we project $x^{(i)}$ onto θ. And then, the length $p^{(i)}$ of the magenta segment is a projection of the i-th training example onto the parameter vector θ.

And so what we have is that

$$\begin{aligned}\theta^T x^{(i)} &= \left(x^{(i)}\right)^T \theta = p^{(i)} \times \|\theta\| \\ &= \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)}\end{aligned}$$

So these are all equally valid ways of computing the inner product $\theta . x^{(i)}$.

Where does this leave us? What this means is that, in the constraints $\theta^T x^{(i)} \geq 1$ if $y = 1$, and $\theta^T x^{(i)} \leq -1$ if $y = 0$, the inner product $\theta^T x^{(i)}$ can be replaced by $p^{(i)} \times \|\theta\|$ (this is the product of two scalars). So, our optimization objective can be thus written as



SVM Decision Boundary

$$\begin{aligned}&\min_\theta \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 \\ \text{s.t.}\quad & p^{(i)} \times \|\theta\| \geq 1 \qquad \text{if } y^{(i)} = 1 \\ & p^{(i)} \times \|\theta\| \leq -1 \quad\ \text{if } y^{(i)} = 1\end{aligned}$$

where we have the simplification $\theta_0 = 0$ and $p^{(i)} = \frac{\theta^T x^{(i)}}{\|\theta\|}$ is the projection of the i-th training example $x^{(i)}$ onto the parameter vector θ.

And, just to remind you, recall that $(1/2)\sum_{j=1}^{n}\theta_j^2 = (1/2)\|\theta\|^2$.

So, let's consider the training example in Fig. 11 and continue to use the simplification $\theta_0 = 0$.

Let's see what decision boundary the SVM will choose. Let's say the SVM were to choose this green decision boundary in the left of Fig. 11. This is not a very good choice, because it has very small margins and comes very close to the training examples.

Let's see why the SVM will not do this. For this choice of parameters, it's possible to show that the vector θ is actually orthogonal to the decision boundary (blue vector in the figure).

The simplification $\theta_0 = 0$ just means that the decision boundary must pass through the origin, $(0, 0)$.

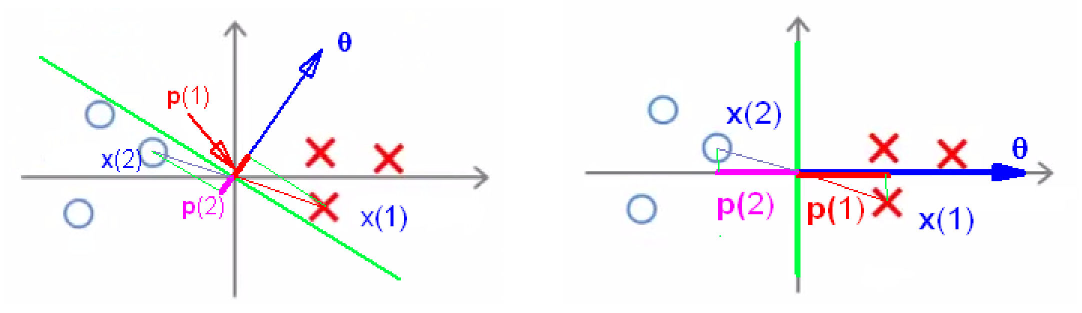


Figure 11: Two possible decision boundaries (green lines): the one shown in the right graph, which has the largest margins, is that produced by the SVM (read the text for getting the details).

So, now let's look at what this implies for the optimization objective. Let's consider the lower red cross to be the example $x^{(1)}$. The projection of this example $x^{(1)}$ (the thin red line) onto θ is the little red line segment called $p^{(1)}$. And that is going to be pretty small, right?

And, similarly, for the example $x^{(2)}$, the projection of it onto θ is the small magenta segment $p^{(2)} = \frac{\theta^T x^{(2)}}{||\theta||}$ which is getting pretty small also. $p^{(2)}$ will actually be a negative number; it is in the opposite direction of θ because the vector (thin gray line) corresponding to $x^{(2)}$ has a greater than 90 degree angle with θ.

And so, what we're finding is that these terms $p^{(i)}$ are going to be pretty small numbers. So, if we look at the optimization objective $p^{(1)} \times ||\theta|| \geq 1$ for the positive example, and $p^{(1)}$ is pretty small, that means that we need $||\theta||$ to be pretty large to satisfy the $p^{(1)} \times ||\theta|| \geq 1$ condition, right?

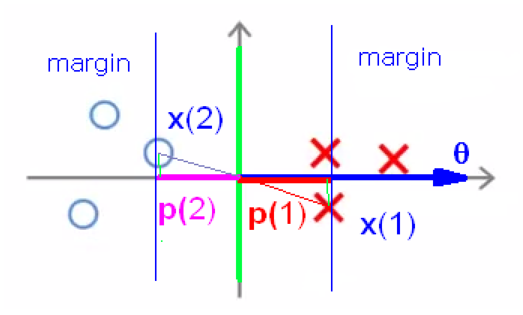
And similarly for our negative example. We need $p^{(2)} \times ||\theta|| \leq -1$, and we saw in this example already that $p^{(2)}$ is going to be a pretty small negative number, and so that means that we need again $||\theta||$ to be pretty large.

But in the optimization objective, $\min_\theta \frac{1}{2}||\theta||^2$, we are trying to find a setting of parameters such that $||\theta||$ is small, and so this green boundary in the left of Fig. 11 doesn't seem like to get such a good direction for the parameter vector θ to be in accordance with our optimization objective.

In contrast, just look at the decision boundary in the right of Fig. 11. The SVM chooses a vertical decision boundary (green), and the corresponding direction for the vector θ is approximately the positive horizontal axis (recall that θ is orthogonal to the boundary). And now the (red) projection of $x^{(1)}$ onto θ, which is $p^{(1)}$, and the (magenta) projection of $x^{(2)}$ onto θ, which is $p^{(2)}$, and note that $p^{(2)} < 0$ is a negative length, notice that now $p^{(1)}$ and $p^{(2)}$, the lengths of the projections, are much bigger than those shown in the left graph.

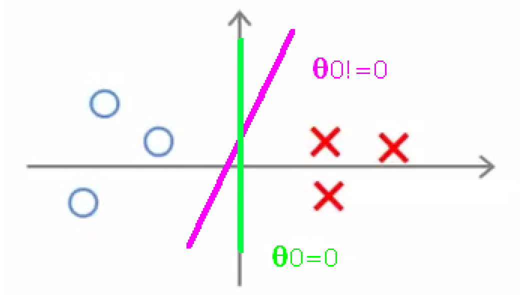
And so, if we still need to enforce the constraints $p^{(1)} \times ||\theta|| \geq 1$ for the positive example and $p^{(2)} \times ||\theta|| \leq -1$ for the negative example, note that $||\theta||$ now will be a smaller number because $p^{(1)}$ and $p^{(2)}$ are so much bigger now. And so, by choosing the vertical decision boundary shown on the right of Fig. 11 instead of that on the left of the same figure, the SVM can make $||\theta||$ much smaller, and therefore make $||\theta||^2$ smaller, what goes along with the minimization goal in our problem.

This is why the SVM would choose this hypothesis on the right of Fig. 11 instead. And this is how the SVM gives rise to the *large margin classification effect*. If you look at the green hypothesis in the right, the projections of the positive and the negative examples onto θ are large, and the only way for that to hold true is if surrounding the green line there's a large margin, there's a large gap that separates positive and negative examples from the hyphotesis, the green boundary.



The magnitude of these margins are exactly the values of $p^{(1)}$, $p^{(2)}$, $p^{(3)}$, and so on. And so by making the margins large, that's how the SVM can end up with a smaller value for $||\theta||$, which is what the objective function is trying to do. And this is why this support vector machine ends up being a large margin classifier, because it's trying to maximize the norm of the $p^{(i)}$ which are the distances from the training examples to the decision boundary.

Finally, we did this whole derivation using the simplification $\theta_0 = 0$. The effect of this enforcement is that the decision boundaries pass through the origin $(0, 0)$ of the graph (green boundary in the example below). If you allow θ_0 to be different from zero, then you can have decision boundaries that do not cross the origin (magenta boundary in the example below).



I'm not going to do the full derivation of that, but it turns out that the large margin proof works in pretty much exactly the same way when $\theta_0 \neq 0$. There's a generalization of the argument just presented that shows that, even when $\theta_0 \neq 0$, what the SVM is trying to do when you have this optimization objectives on $p^{(i)} \times ||\theta||$, and still in the case when C is very large, the SVM is still really trying to find the large margin separator or decision boundary that sits between the positive and negative examples.