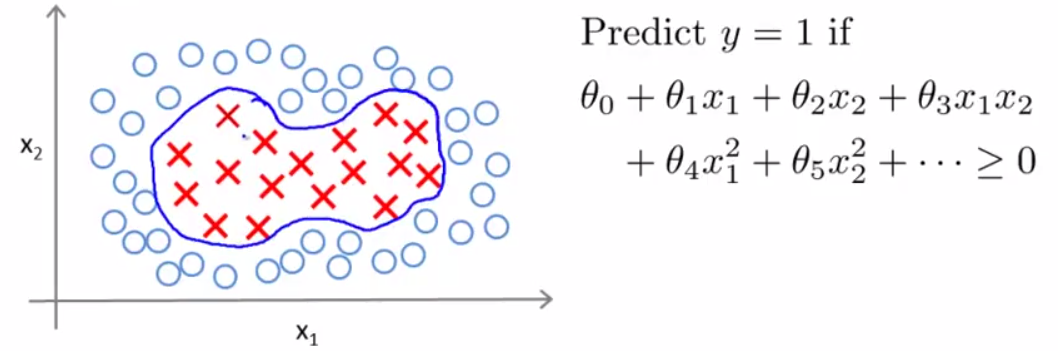
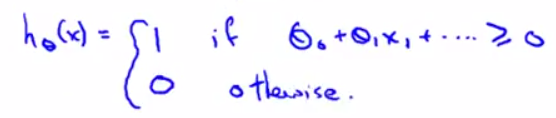
***Kernels***

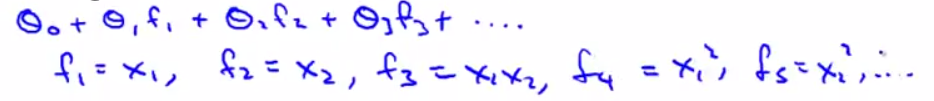
* Now we’d like to start adapting SVMs in order to develop complex nonlinear classifiers via the main technique for doing this, **kernels**.
* If you have a training set that looks like below+ you want to find a nonlinear decision boundary to distinguish positive + negative examples, 1 way to do so is via a set of complex polynomial features:



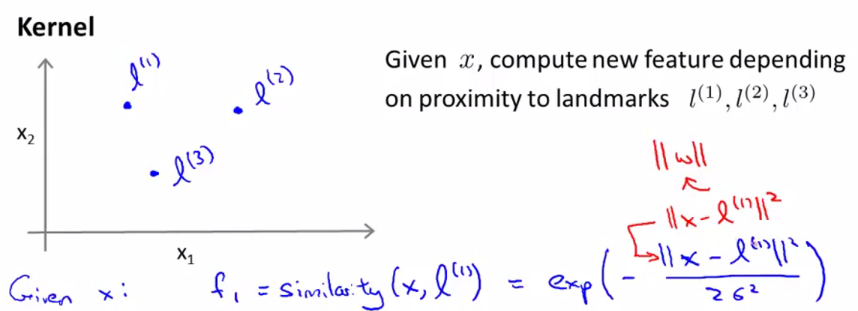
* You end up w/ an hӨ(x) that predicts 1 if all those polynomial features > 0 + predict 0 otherwise



* Another way of writing this w/ “f” instead of “x” for the new features (f4 = x1^2. f5 = x2^2, etc.)



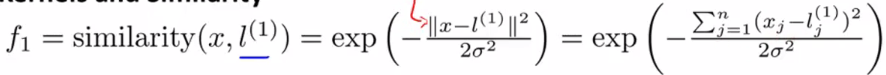
* The question is, *is there a different/better choice of features than high-order polynomials?*
* B/c using high order polynomials can become very computationally expensive
* Here’s a new idea for how to define new features f1, f2, f3 (for real problems we can define a much larger number of features, n, ignoring x0 (intercept)):
* Manually pick a few points = **landmarks**, l1-l3
* Define my *new features* as follows: Given an example X, f1 = some measure of the similarity between my training example X + my 1ST landmark l1



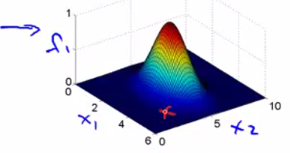
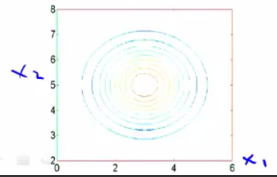
* Then we so the same thing to create f2 as the similarity between X and l2, and the same for f3
* This **similarity function** is be a **kernel function**
* This particular kernel/choice of similarity function is called a **Gaussian kernel**.
* We can have different similarity functions + these different similarity functions are called **kernels**



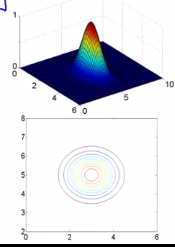
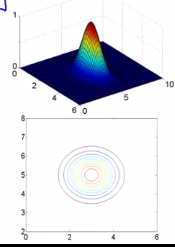
* Take l1, the similarity of the kernel between x + l1 is given by:



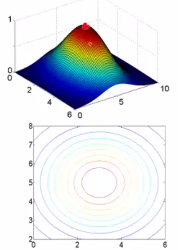
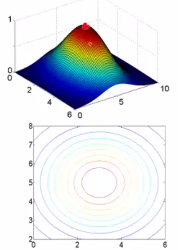
* where that final term on the right is the component-wise distance between the vector X + the vector l (ignoring the intercept term X0, which is always equal to 1)
* This is how you compute the kernel w/ similarity between X + a landmark.
* Suppose X is close to 1 of the landmarks, then this Euclidean distance formula in the numerator will be close to 0, so f1 would be a simple feature = approximately E^-0 = close to 1.
* If X is far from l1, feature f1 will be approximately E^-(some large number^2 / 2δ^2) = close to 0
* What these features do is measure how similar X is from 1 of the landmarks
* A feature f is going to be close to 1 when X is close to a landmark + and 0/close to 0 when X is far from a landmark.
* Each landmarks defines a new feature
* Given the training example X and the landmarks, we can compute new features
* Let's look at this similarity function
* Say I have 2 features X1 + X2, + my 1st landmark, l1 is at a location, {3, 5} + δ^2 = 1:

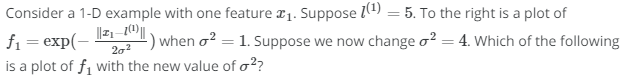
 

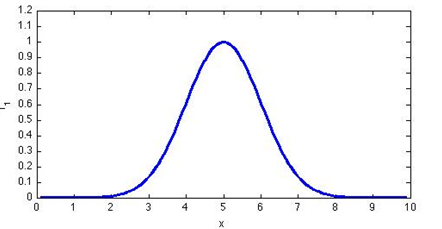
* The height above the surface is the corresponding value of f1, and on the right we have the contour plot of the figure on the left.
* When X is equal to {3, 5} exactly, f1 takes on the value 1, it’s maximum
* As X moves further away, then f1 takes on values closer to 0.
* Remember f1 measures how close X is to the l1
* **δ^2** is the parameter of the Gaussian kernel + as we vary it, we get slightly different effects.
* Set δ^2 = 0.5 + find the kernel looks similar, except the width of the bump becomes narrower + the contours shrink a bit too.

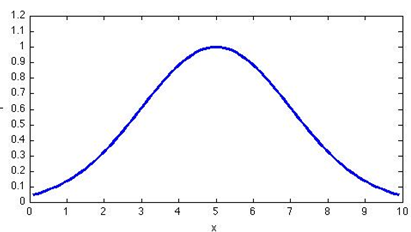
 

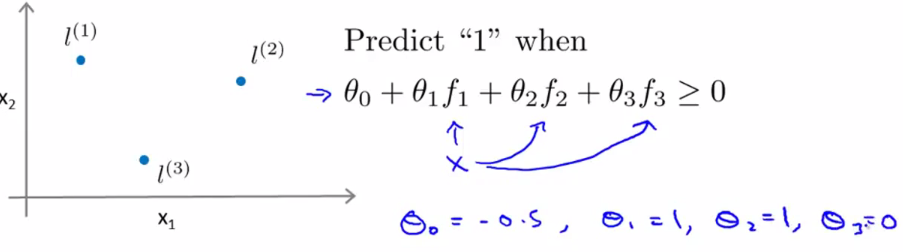
* As X moves away from l1, f1 falls to 0 much more
* If we has increase δ^2 = 3, as X moves away from l1, the value of the f1 falls away much more slowly

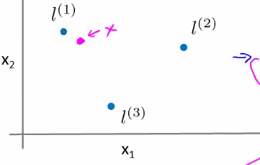
* 



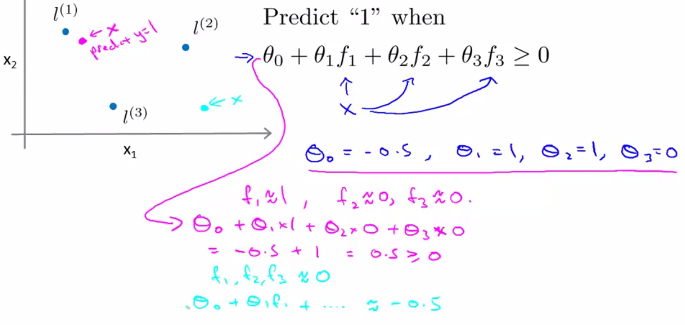
* 
* So, given this definition of the features, let's see what source of hypothesis we can learn.
* Given the training example X, we compute features + our hypothesis hϴ(x) is going to predict y = 1 when Ө0 + Ө1\*f1 + Ө2\*f2, + so on >= 0.



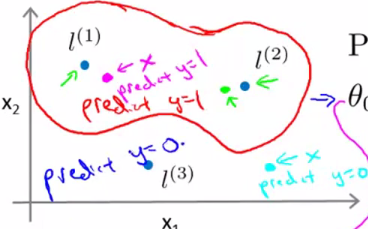
* For a particular example, let's say we've already found a learning algorithm we ended up w/ the above values of the parameters Ө
* If we have a training example X like below:



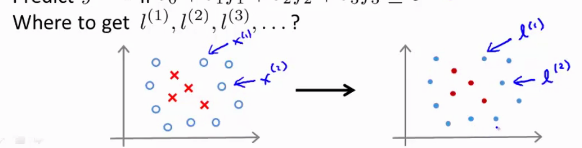
* B/c my training example X is close to l1, f1 is going to be close to 1, and f2 + f3 are close to 0 b/c X is far from l2 and l3
* Therefore we have -0.5 + 1\*1 + 1\*0 + 1\*0 = 0.5, which is >= 0 🡪 we're going to predict Y = 1
* For a new training example below, if you make a similar computation, f1, f2, + f3 are all going to be close to 0, so hϴ(x) = -0.5



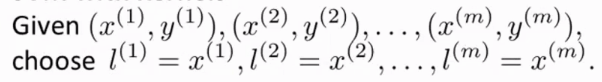
* This is less than 0, so at this point, we're going to predict Y = 0.
* If you do this for a range of different points, you should be able to convince yourself that w/ a training example close to l2, say, we'll also predict Y = 1.
* In fact, for points near l1 + l2 we end up predicting positive (Y = 1) + for points far away from l1 + l2, , we end up predicting the class Y = 0.
* The decision boundary of this hypothesis would end up looking something like below where inside the red decision boundary predicts Y = 1 + outside we predict Y = 0



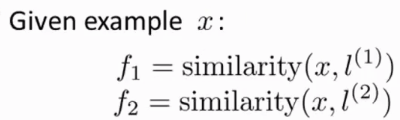
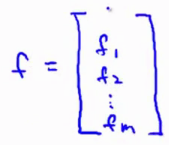
* This is how w/ this definition of landmarks + of the kernel function, we can learn pretty complex non-linear decision boundary
* This is part of the idea of kernels of + how we use them w/ an SVM to define extra features using landmarks + similarity functions to learn more complex nonlinear classifiers.
* But there are a couple of questions that we haven't answered yet.
* How do we get these landmarks? How do we choose these landmarks? What other similarity functions, if any, can we use other than the Gaussian kernel?
* Where do we get landmarks from?
* For more complex learning problems, maybe we want a lot more landmarks than just 3
* In practice, landmarks are chosen from some data set of positive + negative examples
* For every training example that we have, we put landmarks in exactly the same locations
* W/ 1 training example, x1, choose a 1st first landmark l1 to be at exactly the same location as x1
* We end up w/ m landmarks of l1-l(m) (1 landmark location per training example location)



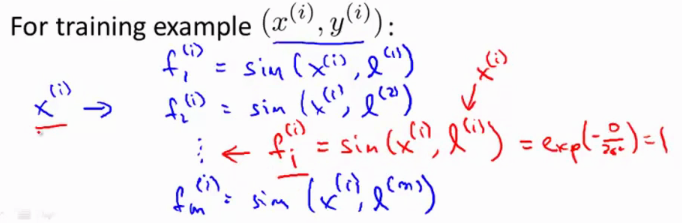
* This is nice b/c it’s basically saying my features, f(m), are going to measure how close an example is to 1 of the things I saw in my training set.
* Given m training examples, choose the location of landmarks to be exactly near the locations of the m training examples.



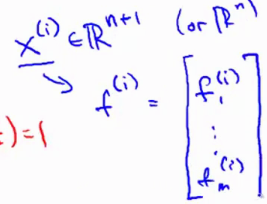
* When given example x (in any set), compute the features (where x(m) = l(m) + get a feature vector (ignoring f0, the intercept, which = 1)

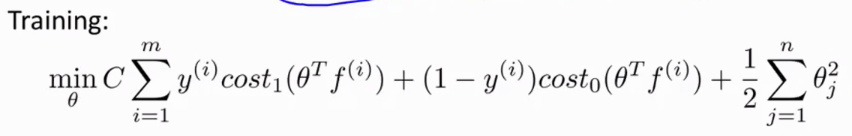
* For a training example x(i), y(i), we will map the given x(i) it to f1(i), the similarity between x(i) + l(1) down to f(m)(i) = similarity between x(i) + l(m).
* Somewhere in this list of values, at the ith component, we will have a feature f(i)(i), the similarity between x(i) + l(i), which are equal
* If using the Gaussian kernel, it’s E^(-0/2δ^2) = 1, AKA 1 of our features for this training example is going to be = 1.



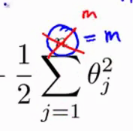
* Then take all these m features + group them into a feature vector
* So, now, instead of representing an example, using x(i), an R(n) or R(n+1) dimensional vector, we can represent a training example using a feature vector, where f0(i) = 1



* Given these kernels + similarity functions, here's how we use a SVM.
* If you already have a learned set of parameters ϴ, given a value of x, we want to make a prediction
* 1st, compute the features f, an R(m + 1)-dimensional feature vector (m training examples + thus m landmarks)
* Then take R(m + 1)-dimensional feature vector ϴ and predict 1 if ϴ (t)\*f >= 0
* ϴ(t)\*f = ϴ0\*f0 + ϴ1\*f1 + … + ϴm\*f(m)
* This is how you make a prediction if you *already have a setting for the parameters ϴ*.
* How do you *GET* the parameters ϴ? 🡺 *using the SVM learning algorithm*
* Specifically, would solve this minimization problem:



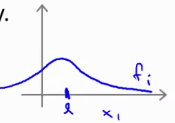
* Minimized parameters ϴ of: C times this cost function but now, instead of making predictions using ϴ(t)\*x(i), our original features, replace them w/ new features f(i) to make a prediction on the ith raining example
* By solving this minimization problem that you get the parameters for your SVM.
* 1 last detail: B/c for this optimization problem, we really have n = m features.

 `🡺 

* The effective number of features we have is a dimension of
* Just think of this as a sum from j = 1 through m where n = m
* We still do the sum from j = 1 b/c in regularization, we don’t regularize the parameter ϴ0
* The way SVM is implemented, this last regularization term is actually done a little bit differently than before.
* You don't really need to know about this last detail in order to use SVMs
* But in the way SVM is implemented, another way to write this term is as ϴ(t)ϴ *if we ignore the parameter ϴ0*



* What most SVM implementations do is actually replace this ϴ(t)\*ϴ w/ ϴ(t)\* some matrix inside, depending on the kernel you use
* This gives us a slightly different distance metric/a slightly different measure instead of minimizing the norm of ϴ squared ( ||ϴ||^2), but minimize something slightly similar to it.
* *It’s like a rescale version of the parameter vector ϴ that depends on the kernel* which allows a SVM software to run much more efficiently.
* W/ have a training set w/ 10k training examples, we end up w/ 10k landmarks so ϴ becomes 10k-dimensional
* When m becomes really, really big, solving for all these parameters can become expensive for SVM optimization software
* Thus solving the minimization problem w/ the modified last term optimizes something slightly different than just minimizing the norm of ϴ squared
* Think of this as an implementation-al detail that changes the objective a bit, but is done primarily for reasons of computational efficiency, so usually you don't have to worry about it
* We don't apply the kernel's idea to other algorithms b/c it turns out the computational tricks that apply for SVMs don't generalize well to other algorithms like logistic regression
* Using kernels w/ logistic regression is going too very slow due to these of computational tricks
* SVMs + kernels tend go particularly well together
* Logistic regression + kernels *can* work, but would run very slowly + won't be able to take advantage of advanced optimization techniques people have figured out for the particular case of SVM w/ a kernel
* But all this pertains only to how you actually implement software to minimize the cost function.
* You really don't need to know how to write software to minimize this cost function b/c you can find very good off the shelf software for doing so.
* 1 other thing that is worth knowing, when applying a SVM, how do you choose the parameters of the SVM considering the bias + variance trade-offs when using a SVM
* When using an SVM, 1 of the things you choose is the parameter C, which plays a role similar to 1/, where λ was the regularization parameter from logistic regression.
* If we have a *large value of C*, this corresponds to a *small value of λ* in logistic regression= not using much regularization
* In this scenario, you tend to have a hypothesis w/ *lower bias + higher variance.*
* Whereas w/ a *smaller value of C,* this corresponds to a *large value of λ* in logistic regression, which corresponds to a hypothesis w*/ higher bias + lower variance*.
* **Hypotheses w/ large C = higher variance = more prone to overfitting**
* **Hypothesis w/ small C = higher bias = more prone to underfitting.**
* The other parameter we need to choose is the parameter **δ^2** (appeared in the Gaussian kernel)
* If **δ^2** is large, the Gaussian kernel would tend to fall off relatively slowly as we change the input x + we’d get a smoother function = a hypothesis w/ higher bias + lower variance



* If **δ^2** was small, the Gaussian kernel will vary more abruptly + the features vary less smoothly (higher slopes/higher derivatives)
* Using this, you end up fitting hypotheses of lower bias + higher variance.

