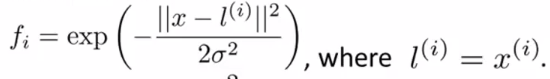
**SVMs in Practice**

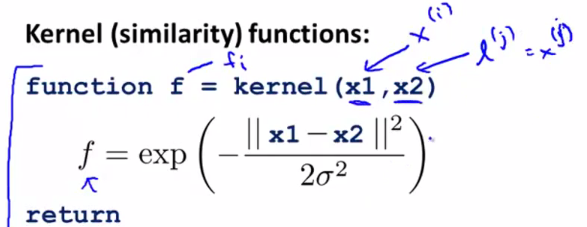
* The SVM algorithm poses a particular optimization problem, + we use some software to solve for SVM parameters Ө
* Even when using some software for SVM, , there are a few things *you* need to do
* 1st: Come up w/ some parameter C
* 2nd: Choose the kernel/similarity function you want to use.
* 1 choice might be to not use any kernel = a **linear kernel**.



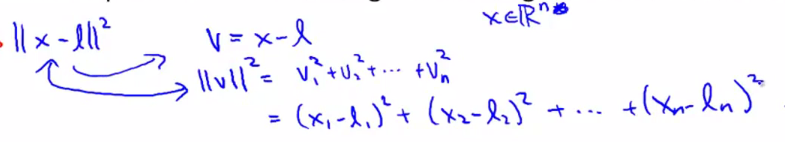
* “I used an SVM w/ a linear kernel” = used an SVM w/out a kernel + just used Ө(t)\*X = Ө0 + Ө1x1 + Ө2x2 … + ӨnXn, which predicts 1 if Ө(t)\*X > 0
* Think of “linear kernel” as a version of SVM that just gives you a standard linear classifier.
* Why would you want to do this?
* If you have a large number of features, n, + the number of training examples, m, is small (have a small training set), maybe you want to just fit a non-complicated linear function/decision boundary b/c you might not have enough data
* Also might risk overfitting if trying to fit a very complicated function in a very high-dimensional feature space w/ a small training set sample
* A 2nd choice for the kernel = **Gaussian kernel**



* W/ this, you need to choose the parameter δ^2
* If δ^2 is large, you have a high bias, low variance classifier, but if δ^2 is small, you have a high variance, low bias classifier.
* You’d choose a Gaussian kernel when n is small AND/OR m is large
* Concretely, if you decide to use a Gaussian kernel, depending on what SVM software package you use, it may ask you to implement a kernel function/the similarity function.
* Octave or MATLAB may ask you to provide a function to compute a particular feature of the kernel that takes as input a test example X + 1 of the landmarks (really 1 of the training examples) + computes the similarity function between them + returns a real number, f(i)



* So the kernel function takes as input X1, X2 + returns a real number + automatically generates all the features (automatically take X + map it to f1, f2, down to f(m)) using this function + trains the SVM from there.
* But sometimes *you* need to provide this function yourself.
* Some SVM implementations will also include the Gaussian + a few other kernels as well (Gaussian + linear kernels = 2 most popular kernels by far)
* 1 implementation note 🡺 If you have features of very different scales, it is important to perform feature scaling before using the Gaussian kernel
* If you imagine the computing the squared norm between X + l in a vector **V = X – l** 🡪 ||V|| = v1^2 + v2^2 + … + v(n)^2 = (x1 – l1)^2 + … + (x(n) – l(n))^2

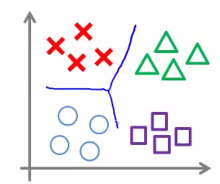


* If your features take on very different ranges of value (square feet of house vs. # of bedrooms), X1 - l1 is going to be huge, whereas X2 - l2 is going to be much smaller
* If that's the case, ||x – l||^2 will be dominated by the sizes of the houses + the number of bathrooms would be largely ignored.
* To avoid this in order to make the SVM work well, perform future scaling to ensure the SVM gives a comparable amount of attention to different features
* When you try SVMs, chances are by far the 2 most common kernels you use will be the linear (i.e. no) kernel or the Gaussian kernel
* **1 note of warning** 🡪 Not all similarity functions you might come up w/ are *valid kernels*
* The Gaussian, linear, + other kernels you sometimes will use all must satisfy a technical condition called **Mercer's Theorem**
* SVM algorithms/implementations of the SVM have lots of clever numerical optimization tricks in order to solve for the parameter's Ө efficiently
* In the original design of SVMs, there was a decision made to restrict attention only to kernels that satisfy Mercer's Theorem
* This makes sure that all SVM software packages can use the large class of optimizations + get the parameter Ө very quickly.
* What most people end up doing is using either the linear or Gaussian kernel, but there are a few other kernels that also satisfy Mercer's theorem you may run across other people using
* 1 is the **polynomial kernel.**
* similarity between X + l is defined as take (X(t)\*l)^2, or (X(t)\*l)^3, or (X(t)\*l + 1)^3, (X(t)\*l + 3)^4, etc. 🡺 **(X(t)\*l + c)^d**
* 2 parameters 🡪 # to add to X(t)\*l and the exponent (degree of the polynomial)
* If X + l are very close w/ each other, the inner product will tend to be large.
* This is a slightly unusual kernel that is not used that often, but
* The polynomial kernel usually performs worse than the Gaussian kernel + is not used that much, but you may run across it
* Usually it is used only for data where X + l are all strictly non negative, + which ensures that the inner products are never negative.
* This captures the intuition that X + l are very similar to each other, then maybe the inner product between them will be large.
* More esoteric kernels you can use to measure similarity between different objects as well:
* **String Kernel** 🡪 sometimes used if input data is text strings or other types of strings
* 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 2 strings using this kernel
* **Chi-Square Kernel**
* **Histogram Intersection Kernel**

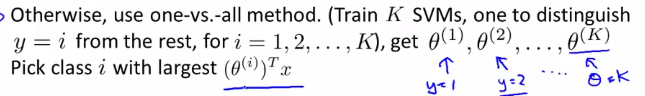




* 2 last details:
* 1) **Multiclass Classification** 🡪 y = {1, 2, 3, … k} = 4 classes



* Many SVM packages already have built-in multiclass classification functionality.
* Otherwise, 1 way to do this is to use the **one versus all method** (logistic regression)
* 1) Train k SVM's (for the k classes) w/ 1 SVM to distinguish each of the classes from the rest.
* This would give you k parameter vectors:
* Vector Ө1 🡪 tries to distinguish class y = 1 from all other classes
* Vector Ө2 🡪 result of y = 2 as the positive class + all others as the negative class
* And so on up to a parameter vector Өk, the parameter vector for distinguishing the final class k from anything else
* 2) We then predict the class i w/ the largest Ө(t)\*X.



* For common cases, whatever the software package, there’s a reasonable chance multiclass classification functionality is already built-in, + so you don't need to worry about this result
* Remember we developed SVMs starting off w/ logistic regression + then modifying the cost function a little bit
* Let's say n = number of features + m = number of training examples.
* *When should we use one algorithm versus the other?*
* If n is larger relative to your training set size (n >> m), such as a text classification problem where dimensions of the feature vector = 10k + your training set size is 10:
* What we usually do is use logistic regression *or* use SVM *w/out a kernel* (w/ a linear kernel)
* B/c w/ so many features + smaller training sets, a linear function will probably do fine
* You don't have really enough data to fit a very complicated nonlinear function.
* Now if is n is small + m is *intermediate* (n is anywhere from 1-1000 + m is anywhere from 10-10k):
* Often an SVM w/ a Gaussian kernel will work well
* Ex: 2D-training set (n = 2) + a pretty large number of training examples.



* A Gaussian kernel will do a pretty good job separating positive + negative classes.
* But if n is small but m is large (n << m):
* An SVM w/ Gaussian Kernel will be somewhat slow to run.
* Today's SVM packages, if using a Gaussian Kernel, are very good, but can still struggle a little bit w/ a massive, massive training set size
* Try to just manually create/add have more features + then use logistic regression or an SVM w/out a kernel (w/ a linear kernel)
* We kind of paired logistic regression and linear (no) kernels together in the above examples b/c they’re pretty similar algorithms that do pretty similar things + give pretty similar performance
* But depending on your implementation details, 1 may be more efficient than the other.
* Where 1 of these algorithms applies, the other is to likely to work pretty well
* But w/ the power of SVM, when you have maybe up to 10k-50k examples + the number of features is reasonably large, we have a scenario where a SVM w/ a Gaussian kernel will shine.
* It can do things that are much harder to do for logistic regression
* Finally, where do NNs fit in?
* For all of these problems, a well-designed NN is likely to work well as well.
* 1 disadvantage/reason to not use a NN is that, for some of these problems, the NN might be slow to train + a very good SVM implementation package could run quite a bit faster than a NN network.
* It turns out that the optimization problem the SVM has is a **convex optimization problem**
* The good SVM optimization software packages will *always* find the global minimum or something close to it
* i.e. For the SVM you don't need to worry about local optima.
* In practice, local optima aren't a huge problem for NNs, but it’s 1 less thing to worry about if using an SVM.
* If you're looking at some problems + you’re still not entirely sure which algorithm to use, that's actually okay.
* The algorithm does matter, but what often matters even more is things like how much data you have, how skilled are you, how good are you at doing error analysis + debugging learning algorithms, figuring out how to design new features + what other features to give a learning algorithms, + so on
* Often those things will matter more than whether you’re using logistic regression or an SVM.
* But having said that, the SVM is still widely perceived as 1 of the most powerful learning algorithms + are a very effective way to learn complex non-linear functions.