## Using an Support Vector Machine

In this lesson we're going to talk about what we actually need to do in order to run or to use an SVM.

- We saw previously that is not recommend writing or implementing our own software to solve for the parameters theta by ourselves, insted we can use:
  - A SVM software package (e.g. liblinear, libsvm, ...) to solve for parameters $\theta$.

Even though we shouldn't be writing our own SVM optimization software, there are a few things we need to do, we need to specify:

- Choice of parameter $C$
- Choice of kernel (similarity function):
  - E.g. No kernel ("linear kernel")
    - Predict "$y = 1$" if $\theta^T x \geq 0$ (where $\theta^T x = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n$)
    - This term linear kernel, we can think of this as the version of the SVM that just gives us a standard linear classifier.
    - When we would want to use a "linear kernel" (no kernel)?, if we have a large numbe of features $n$, and a small number of training examples $m$, maybe we want to just fit a linear decision boundary and not try to fit a very complicated nonlinear function, because we might not have enough data. And we might risk overfitting, if we're trying to fit a very complicated function.
  - Gaussian kernel:

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

Need to choose $\sigma^2$.

  - When would you chose a Gaussian kernel?
    - If $n$ is small and/or $m$ is large
  - If we're using a Gaussian kernel then we may need to implement the kernel function (that returns a real number)
    - Kernel Similarity function (in Octave):

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

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

```
return
```

    - This function is computing $f^{(i)}$ for one particular value of $i$, where $f$ is just a single real number
    - Note: Do perform **feature scaling** before using the Gaussian Kernel
      - If we don't, features with a large value will dominate the $f$ value
    - Some SVM packages will expect us to define kernel
  - Although, some SVM implementations include the Gaussian and a few others
    - Gaussian is probably most popular kernel

#### Other choices of kernel

When we try a support vector machines chances are by far the two most common kernels we use will be the linear kernel, meaning no kernel, or the Gaussian kernel that we saw previously.

Note: Not all similarity functions $\text{similarity}(x, l)$ make valid kernels. (Need to satisfy technical condition called "Mercer's Theorem" to make sure SVM packages' optimizations run correctly, and do not diverge).

- SVM use numerical optimization tricks
  - Mean certain optimizations can be made, but they must follow the Mercer's Theorem

**Many off-the-shelf kernels available:**

- Polynomial Kernel
  - We measure the similarity of $x$ and $l$ by doing one of:
    - $k(x, l) = (x^T l + 0)^2, (x^T l + 1)^3, (x^T l + 5)^4$
    - General form is $(x^T l + \text{constant})^{\text{degree}}$
    - If they're similar then the inner product tends to be large
    - Not used that often and use two parameters:
      - Degree of polynomial $d$ (degree)
      - Number we add to $l$ (constant)
    - This kernel usually performs worse than the Gaussian kernel
    - Polynomial kernel is used when $x$ and $l$ are both estrictly non-negative
- More esoteric: String kernel, chi-square kernel, histogram, intersection kernel, ...

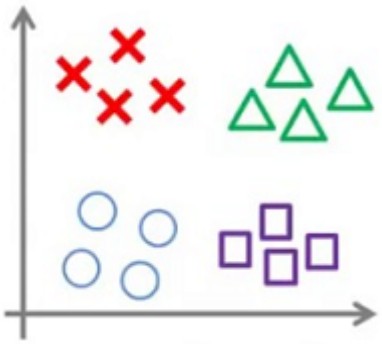
**Video Question:** Suppose you are trying to decide among a few different choices of kernel and are also choosing parameters such as $C$, $\sigma^2$, etc. How should you make the choice?

- Choose whatever performs best on the training data.

> Choose whatever performs best on the cross-validation data.

- Choose whatever performs best on the test data.
- Choose whatever gives the largest SVM margin.

**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 $\theta^{(1)}$, $\theta^{(2)}$, …, $\theta^{(k)}$. Pick class $i$ with larget $(\theta^{(i)})$^Tx

## Logistic regression vs. SVM

Finally, we developed support vector machines starting off with logistic regression and then modifying the cost function a little bit. So when should we use SVM and when is logistic regression more applicable?

We suppose that $n$ = number of features $(x \in \mathbb{R}^{n+1})$, $m$ = number of training examples

- If $n$ (features) is large (relative to $m$):
  - E.g. text classification problem (where $n \geq m$)
    - Feature vector dimension is 10,000
    - Training set is 10 to 1,000 examples
    - Then use logistic regression or SVM with a linear kernel

    - Use logistic regression, or SVM without a kernel ("linear kernel")
- If $n$ is small, $m$ is intermediate ($n$ = 1 to 1000, $m$ = 10 to 10000):
    - $m$ = 10 to 10000, maybe $m$ = 50000, but not $m$ = 1000000
    - Use SVM with Gaussian kernel
- If $n$ is small, $m$ is large:
    - $n$ = 1 to 1000, $m$ = 50000+
    - Create/add more features, then use logistic regression or SVM without a kernel (linear kernel)
    - SVM without a kernel because SVM will be slow to run with Gaussian kernel

**Logistic regression and SVM with a linear kernel are pretty similar** (both do similar things and get similar performance), but depending on our implementational details, one may be more efficient than the other.

- A lot of SVM's power is using diferent kernels to learn complex non-linear functions

For all these regimes a well designed Neural Network likely to work well for most of these settings, but may be slower to train (SVM well implemented would be faster).

- Support Vector Machine has a convex optimization problem - so we get a global minimum.
- It's not always clear how to chose an algorithm
    - Often more important to get enough data
    - Designing new features
    - Debugging the algorithm

**Support Vector Machine is widely perceived a very powerful learning algorithm**