# CSE 546 Machine Learning, Winter 2016
# Homework 3

Due: Tuesday, February 29, beginning of class

Please submit a PDF of your answers and your programming implementation to the class dropbox:
`https://catalyst.uw.edu/collectit/dropbox/summary/farhadi/37186`

## 1 Fitting an SVM classifier by hand [50 Points]

(Source: Murphy text, Exercise 14.1) Consider a dataset with 2 points in 1d:

$$(x_1 = 0, y_1 = -1) \text{ and } (x_2 = \sqrt{2}, y_2 = 1).$$

Consider mapping each point to 3d using the feature vector $\phi(x) = [1, \sqrt{2}x, x^2]^T$ (This is equivalent to using a second order polynomial kernel). The max margin classifier has the form

$$\hat{w}, \hat{w_0} = \arg\min ||w||^2 \quad s.t. \tag{1}$$
$$y_1(w^T\phi(x_1) + w_0) \geq 1 \tag{2}$$
$$y_2(w^T\phi(x_2) + w_0) \geq 1 \tag{3}$$

1. *(10 Points)* Write down a vector that is parallel to the optimal vector $\hat{w}$. Hint: Recall from Figure 14.12 (page 500 in the Murphy text) that $\hat{w}$ is perpendicular to the decision boundary between the two points in the 3d feature space.

2. *(10 Points)* What is the value of the margin that is achieved by this $\hat{w}$? Hint: Recall that the margin is the distance from each support vector to the decision boundary. Hint 2: Think about the geometry of the points in feature space, and the vector between them.

3. *(10 Points)* Solve for $\hat{w}$, using the fact the margin is equal to $1/||\hat{w}||$.

4. *(10 Points)* Solve for $\hat{w_0}$ using your value for $\hat{w}$ and Equations 1 to 3. Hint: The points will be on the decision boundary, so the inequalities will be tight. A "tight inequality" is an inequality that is as strict as possible. For this problem, this means that plugging in these points will push the left-hand side of Equations 2 and 3 as close to 1 as possible.

5. *(10 Points)* Write down the form of the discriminant function $f(x) = \hat{w_0} + \hat{w}^T\phi(x)$ as an explicit function of $x$. Plot the 2 jkkk points in the dataset, along with $f(x)$ in a 2d plot. You may generate this plot by hand, or using a computational tool like Python.

**Show your work.**

# 2 Kernels [50 Points]

In this section we will work with a regression model called a *normalized RBF network*. As usual, we have a dataset $(x_i, y_i)$ with $M$ distinct examples, where $x_i \in \mathbb{R}^N$ and $y_i \in \mathbb{R}$. We will be performing linear regression, and here we will use the RBF kernel to create our basis functions. Recall that the RBF kernel is

$$K(y, z) = \exp\left(\frac{-||y - z||_2^2}{2\sigma^2}\right)$$

where $\sigma$ is known as the "bandwidth" and determines the "width" of the kernel. As $\sigma \to 0$, $K(y, z)$ tends to 0 when $y \neq z$ and 1 when $y = z$. As $\sigma \to \infty$, $K(y, z)$ tends to the same value for all $y$, $z$. We will use the RBF kernel and our data points to define $M$ basis functions

$$\phi_i(x) = K(x, x_i)/Z(x)$$

$$Z(x) = \sum_{j=1}^{M} K(x, x_j)$$

In addition, we define a constant bias term $\phi_0(x) = 1$, and define a linear prediction function with weights $w_0, \ldots, w_M$

$$f(x) = \sum_{i=0}^{M} w_i\phi_i(x)$$

1. *(10 points)* Assume that our loss function is the sum of squared errors on the training data with an L2 penalty on the non-bias weights.

$$L(w) = \frac{1}{2}\left(\sum_{i=1}^{M}(y_i - f(x_i))^2 + \lambda\sum_{i=1}^{M} w_i^2\right)$$

   We will also define a helpful matrix

$$Q = \begin{bmatrix} \phi_0(x_1) & \phi_1(x_1) & \ldots & \phi_M(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_M) & \phi_1(x_M) & \ldots & \phi_M(x_M) \end{bmatrix}$$

   Using this matrix, give a closed form solution for the optimal weights $w^*$.

2. *(20 points)* Choosing the bandwidth is very important, as setting it too high or too low can lead to suboptimal results. Give the optimal weights $w^*$ that result as $\sigma \to \infty$.

You do not need to give a rigorous proof of optimality, but you should justify your answer reasonably.

(Hint 1) Consider what happens to $\phi_i(x_j)$ as $\sigma \to \infty$.
(Hint 2) Let's define $\bar{y} = \sum_i y_i/M$.

3. *(20 points)* Now let's assume that we have no bias term and no $L2$ penalty, so

$$f(x) = \sum_{i=1}^{M} w_i \phi_i(x)$$

$$L(w) = \frac{1}{2}\left(\sum_{i=1}^{M}(y_i - f(x_i))^2\right)$$

Give the optimal weights $w^*$ as $\sigma \to 0$. Again, you do not need a rigorous proof, but you should justify your answer.

4. *(EXTRA CREDIT +5 points)* Assume that we add back in the bias term and $L2$ penalty, and choose $\lambda$ to be a very small (but non-zero) value. What are the optimal weights $w^*$ as $\sigma \to 0$? Justify your answer.

# 3 Programming Question [100 Points]

In this problem, we seek to perform a digit recognition task, where we are given an image of a handwritten digit and wish to predict what number it represents. This is a special case of an important area of language processing known as Optical Character Recognition (OCR). We will be simplifying our goal to that of a binary classification between two relatively hard-to-distinguish numbers (specifically, predicting a '3' versus a '5'). To do this, you will implement a kernelized version of the Perceptron algorithm.

You can use whatever language you want but make sure it runs on the department's linux machines. Python is preferred.

## 3.1 Dataset

The digit images have been taken from the Kaggle competition linked to on the projects page, `http://www.kaggle.com/c/digit-recognizer/data`. This data was originally from the MNIST database of handwritten digits, but was converted into a easier-to-use file format.

The data has also undergone some preprocessing. It has been filtered to just those datapoints whose labels are 3 or 5, which have then been relabeled to 1 and -1 respectively. Then, 1000-point samples have been created, named *validation.csv* and *test.csv*. The first column of these files is the label of each point, followed by the grayscale value of each pixel.

## 3.2 Perceptron

In the basic Perceptron algorithm, we keep track of a weight vector $w$, and define our prediction to be $\hat{y}^{(t)} = sign(w \cdot x^{(t)})$. If we predict a point correctly, we make no update and continue running. Any time we make a mistake, our update step is

$$w^{(t+1)} \leftarrow w^{(t)} + y^{(t)} x^{(t)},$$

$$\text{so at time } t,\ w^{(t)} = \sum_{i \in M^{(t)}} y^i x^i$$

where $M^{(t)}$ is the set of mistakes made up to time $t$.

### 3.3 Kernels

To apply the kernel trick, we would like to replace $x$ and $w$ with $\Phi(x)$ and $\Phi(w)$, where $\Phi : X \to F$ is a mapping into some high- or infinite-dimensional space. For example, $\Phi$ could map to the set of all polynomials of degree exactly $d$. To do this, we try to find a function $k$ for this particular $\Phi$ that has the property $k(u, v) = \Phi(u) \cdot \Phi(v)$ for every $u$ and $v$. The trick, however, is that although this function lets us compute dot products easily, we must not actually deal with any $\Phi(x)$ directly. Because of this, the natural extension of storing our weight vector doesn't work:

$$\Phi(w^{(t)}) = \sum_{i \in M^{(t)}} y^i \Phi(x^i)$$

would require both computing the sum of $\Phi(x^i)$ explicitly and storing it as $\Phi(w)$. As stated above, $\Phi(w)$ could have millions (or in fact an infinite number) of terms, so this can quickly become impractical. Instead, we can rely on the fact that our prediction becomes

$$\hat{y}^{(t)} = sign(\Phi(w) \cdot \Phi(x)) = sign\left(\sum_{j \in M^{(t)}} y^j \Phi(x^j) \cdot \Phi(x^{(t)})\right) = sign\left(\sum_{j \in M^{(t)}} y^j k(x^j, x^{(t)})\right)$$

This means that it's possible for us to store the $(x^j, y^j)$ pairs of our mistakes $M^{(t)}$, and use these to compute our dot product $\Phi(w) \cdot \Phi(x)$.

The drawback of this is that we are now storing a list of all mistakes we ever make, which is quite a bit more overhead than simply $w$ in the case without kernels. This also means that if we make too many mistakes, performing the dot product can become quite slow. However, we are now able to build much more complex models, and changing between models is as easy as switching kernel functions.

### 3.4 Task

*Hint: It is probably a good idea to write your Perceptron implementation so that it can be passed a kernel function as an argument.*

1. *(40 Points)* First, get Perceptron working with the kernel $k_p^1(u, v) = u \cdot v + 1$.
   ($k_p^1$ is what the standard dot product would give us, if we had added a constant term $x_0 \equiv 1$.)
   Run Perceptron for a single pass over the validation set with this kernel, and plot the average loss $\bar{L}$ as a function of the number of steps $T$, where

$$\bar{L}(T) = \frac{1}{T} \sum_{j=1}^{T} \mathbb{1}(\hat{y}^{(t)} \neq y^{(t)}) \tag{4}$$

   where $\hat{y}^t$ is the label that Perceptron predicts for datapoint $t$ as it runs, and $\mathbb{1}$ is an indicator function, which is 1 if its condition is true and 0 otherwise. Record the average loss every 100 steps, e.g. [100, 200, 300, ...].

2. *(30 Points)* For a positive integer $d$, the polynomial kernel $k_p^d(u, v) = (u \cdot v + 1)^d$ maps $X$ into a feature space of all polynomials of degree up to $d$.

   For the set $d = [1, 3, 5, 7, 10, 15, 20]$, run Perceptron for a single pass over the validation set with $k_p^d$, and plot the average loss over the validation set $\bar{L}(1000)$ as a function of $d$. What value of $d$ produces the lowest loss?

3. *(30 Points)* For $\sigma > 0$, the Exponential kernel $k_E^\sigma(u, v) = e^{-\frac{\|u-v\|}{2\sigma^2}}$ is a map to *all* polynomials of $x$, where $\sigma$ is a tuning constant that roughly corresponds to the "window size" of the exponential. Tuning on the validation set has produced a value of $\sigma = 10$.

   For the $d$ you chose in the previous step, run Perceptron with both $k_p^d$ and $k_E^{10}$ for a single pass over the testing set. For each of these two kernels, plot the average loss $\bar{L}(T)$ as a function of the number of steps. As above, report the average loss for every 100 steps.