CSE446 Machine Learning, Winter 2017: Homework 3

Due: Monday, February 20th, by 9:30am

Instructions  There are 3 written questions on this assignment, plus a fourth coding question. Submit both your written answers (as a pdf) and your implementation to the Dropbox at https://catalyst.uw.edu/collectit/dropbox/summary/qaz2wsx3/39488 Please show your work for all questions.

You are required to submit the following files or documents for this homework:

• A PDF file containing answers to all written questions. You must include the plots and explanation for programming questions (if asked or required) in this PDF document only. This document should be of a reasonable size (no more than a few MB).

• A completed version of the Python template code for the programming assignment. Since you’ll just be submitting a single file, there’s no need to compress it or anything.

These are the only items you should submit. In particular, please do not include the data used for the assignment, or .png files with your plots.

Your written answers may be typed, or you may scan in hand-written answers. You’re welcome to type your solutions in LaTex if you know how. If you don’t know LaTex but want to type, there a number of markdown editors with real-time preview and equation editing. Here are two: https://stackedit.io/, http://marxi.co/. Writing your solutions by hand is also fine as long as they’re neat.

You are welcome to use any Python libraries for data munging, visualization, and numerical linear algebra. Examples includes Numpy, Pandas, and Matplotlib. You may NOT, however, use any Python machine learning libraries such as Scikit-Learn or TensorFlow. If in doubt, email the instructors.

Please note also that all references to Murphy refer to the fourth printing of the textbook. Page and section numbers may differ between editions.

1 Perceptrons [22 points]

Recall that a perceptron learns a linear classifier with weight vector w. It predicts

\[ \hat{y} = \text{sign}(w^T x_t) \]

(assuming here that \( \hat{y} \in \{+1, -1\} \). Also, note that we are not using a bias weight \( w_0 \), for simplicity). When the perceptron makes a mistake, it updates the weights using the formula

\[ w^{(t+1)} = w^{(t)} + y_t x_t \]

Imagine that we have \( x_t \in \mathbb{R}^2 \), and we encounter the following data points

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-3</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
1. (10 points) Starting with \( \mathbf{w} = [0 \ 0]^T \), use the perceptron algorithm to learn on the data points in the order from top to bottom. Show the perceptron’s linear decision boundary after observing each data point in the graphs below. Be sure to show which side is classified as positive.
2. (5 points) Does our learned perceptron maximize the margin between the training data and the decision boundary? If not, draw the maximum-margin decision boundary on the graph below.

3. (7 points) Assume that we continue to collect data and train the perceptron. If all data we see (including the points we just trained on) are linearly separable with margin $\gamma = 0.5$ and have maximum norm $||x_t|| \leq 5$, what is the maximum number of mistakes we can make on future data?
2 SVMs: Hinge loss and mistake bounds [24 Points]

Suppose we build a predictive model \( \hat{y} = f_w(x) = \text{sgn}(w^T x) \), where \( \text{sgn}(z) = 1 \) if \( z \) is positive and -1 otherwise. Here we are dealing with binary predictions where each label is in \( \{-1, 1\} \). Let us say we make a mistake on \( (x, y) \) with \( w \) if \( y \neq \text{sgn}(w^T x) \). The number of mistakes with \( w \) on a dataset is the classification loss.

The SVM loss function can be viewed as a relaxation to the classification loss. The hinge loss on a pair \( (x, y) \) is defined as:

\[
\ell((x, y), w) = \max \left[ 0, 1 - y\langle w, x \rangle \right],
\]

where \( x \in \mathbb{R}^D \) and \( y \in \{-1, 1\} \). (Recall that the hinge loss we examined for SVM is definitely slightly differently than the hinge loss for the perceptron, but both are considered “hinge loss”.) The SVM attempts to minimize:

\[
\frac{1}{N} \sum_{i=1}^{N} \ell((x, y), w) + \lambda ||w||^2,
\]

where \( \lambda \) is a regularizer.

The hinge loss provides a way of dealing with datasets that are not separable. Let us go through the argument as to why we view this as a relaxation of finding the max margin classifier.

1. (7 Points) Show that the function \( \ell((x, y), w) = \max \left[ 0, 1 - y\langle w, x \rangle \right] \) is convex (as a function of \( w \)). You may write a rigorous mathematical proof if you like, but a well-annotated picture with a clear explanation is also just fine.

2. (7 Points) (Margin mistakes) Suppose that for some \( w \) we have a correct prediction of \( y_i \) with \( x_i \), i.e. \( y_i = \text{sgn}(w^T x_i) \). What range of values can the hinge loss, \( \ell((x, y), w) \), take on this correctly classified example? Points that are classified correctly and which have non-zero hinge loss are referred to as margin mistakes.

3. (10 Points) (Mistake bound) Let \( M(w) \) be the number of mistakes made by \( w \) on our dataset (in terms of the classification loss). Show that:

\[
\frac{1}{N} M(w) \leq \frac{1}{N} \sum_{i=1}^{N} \max \left[ 0, 1 - y_i \langle w, x_i \rangle \right].
\]

In other words, the average hinge loss on our dataset is an upper bound on the average number of mistakes we make on our dataset.
3 Kernel functions and linear separability [14 Points]

(Source: Norvig and Russell, Exercise 18.16.) The power of kernel methods stems from the observation that data which are not linearly separable in their input feature space may in fact be separable in a higher-dimensional space, which is a function of the input features. For example, the data shown in Figure 1 are not linearly separable in the feature space $(x[1], x[2])$, but they are separable in the transformed space where we transform each point $x = (x[1], x[2])$ according to $\Phi(x) = (x[1]^2, x[2]^2, \sqrt{2}x[1]x[2])$. Further, we can observe that for two data points $x$ and $y$, we have that
\[
\]
where $K$ is a kernel function. Kernel functions are described in greater detail in Problem 4.3.

In this problem, we will generalize the situation shown in Figure 1.

1. (6 Points) Consider data as in Figure 1 but now separated by a circular decision boundary of arbitrary center and radius. Such a decision boundary can be expressed as $(x[1] - a)^2 + (x[2] - b)^2 - r^2 = 0$. Show that a linear decision boundary can always be found if the data are projected in the four-dimensional feature space $(x[1], x[2], x[1]^2, x[2]^2)$.

2. (8 Points) For data separated by an ellipse of the form $c(x[1] - a)^2 + d(x[2] - b)^2 - 1 = 0$, show that a linear decision boundary can be found if the data are projected into the five-dimensional feature space $(x[1], x[2], x[1]^2, x[2]^2, x[1]x[2])$.

Figure 1: In (a), a two-dimensional training set with positive examples as black circles and negative as white circles. The true decision boundary, $x[1]^2 + x[2]^2 \leq 1$, is also shown. In (b), the data have been mapped into the three-dimensional input space $(x[1]^2, x[2]^2, \sqrt{2}x[1]x[2])$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions.

A notation detail: in case the notation $x \cdot y$ is new to you, it’s fine to think of it as the same as $x^T y$. Unless you really like math, stop reading here. For math people, we use the “dot” notation here and in Problem 4 because it turns out that inner products can be defined on spaces more general than $\mathbb{R}^n$; in particular, they can be defined over infinite-dimensional spaces such as Hilbert space. In such a space, the notation $x^T y$ doesn’t have an obvious semantics — what does it mean to transpose an infinite-dimensional object? However, the inner product is still well-defined. Kernel methods thus allow you to manipulate infinite-dimensional objects with finite memory, by working only with similarities between object pairs and not representing the objects explicitly. Did that just totally blow your mind? If this sort of things interests you, search online or in the Murphy textbook for the phrase “reproducing kernel Hilbert space”, or email the course staff. On the other hand, if this discussion sounds like total nonsense, forget you ever read this.
4 Programming Question [40 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.

4.1 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](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.

4.2 Perceptron

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

\[
\begin{align*}
w(t+1) & \leftarrow w(t) + y_t x_t, \\
w(t) & = \sum_{i \in M(t)} y_i x_i
\end{align*}
\]

(5)

so at time \(t\),

\[
w(t) = \sum_{i \in M(t)} y_i x_i
\]

(6)

where \(M(t)\) is the set of mistakes made up to time \(t\).

4.3 Kernels

To apply the kernel trick, we would like to replace \(x\) and \(w\) with \(\Phi(x)\) and \(\Phi(w)\), where \(\Phi : X \rightarrow F\) 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:

\[
\Phi(w(t)) = \sum_{i \in M(t)} y_i \Phi(x_i)
\]

(7)

would require both computing the sum of \(\Phi(x_i)\) explicitly and storing it as \(\Phi(w(t))\). As stated above, \(\Phi(w(t))\) 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

\[
\hat{y}_t = \text{sgn} \left( \Phi(w(t)) \cdot \Phi(x_t) \right) = \text{sgn} \left( \sum_{i \in M(t)} y_i \Phi(x_i) \cdot \Phi(x_t) \right) = \text{sgn} \left( \sum_{i \in M(t)} y_i K(x_i, x_t) \right)
\]

(8)

\footnote{A notation point: at the \(t\)th time step, we predict \(y\) for the \(t\)th point in our data set and call this prediction \(\hat{y}_t\). We make the prediction using the current weight vector \(w(t)\), and the value of \(x\) for the \(t\)th point \(x_t\). The index \(t\) appears as a subscript on \(y\) and \(x\) because it is serving to index into the training data. It appears as a superscript on \(w\) to indicate that \(w\) is being updated on every iteration of the algorithm.}
This means that it’s possible for us to store the \((x_i, y_i)\) pairs of our mistakes \(M(t)\), and use these to compute our dot product \(\Phi(w(t)) \cdot \Phi(x_i)\).

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.

### 4.4 Task

For this part of the assignment, you will fill in the blanks in some Python code to fit a Perceptron. Download the code file `perceptron_template.py`, as well as the data files `test.csv` and `validation.csv` from the course’s homework website. For the program to run correctly, you should put the data in the same directory as the code. Some plots will also be generated in this same directory. However, make sure not to submit the data and the plots along with your code. This eats up space on the TA’s hard drives.

1. **(20 Points)** We will first get the perceptron algorithm working with the linear kernel

   \[ K^1_\text{p}(u, v) = u \cdot v + 1. \]  

   (The kernel \(K^1_\text{p}\) is what the standard dot product would give us, if we had added a constant term \(x_0 = 0\).)

   - First, fill in the function `linear_kernel` in `perceptron_template.py` such that it implements (9).

   - Then, fill in the function `compute_y_hat`. This function takes an input vector \(x_t\), as well as a vector \(y_{\text{mistake}}\) whose \(i^{th}\) element is the output for the \(i^{th}\) data point that the model has predicted wrong so far, a matrix \(X_{\text{mistake}}\) whose \(i^{th}\) row is the input for the \(i^{th}\) data point that the model has gotten wrong so far, and a kernel function. This function computes \(\hat{y}_t\) as specified in (8).

   - Finally, in the function `fit_perceptron`, define variables \(x_t\), \(y_t\), \(X_{\text{mistake}}\), and \(y_{\text{mistake}}\) that should be fed to `compute_y_hat` to make a prediction for the \(t^{th}\) data point. As you can see from the code, if the model predicts incorrectly on the \(t^{th}\) point, it records this point as a mistake.

   - When this is all done, enter `python perceptron_template.py linear` at the command line. The program will run the perceptron algorithm for one pass over the validation data set, using a linear kernel. It will output a plot `linear-kernel.png` showing the average loss

   \[ \bar{L}(T) = \frac{1}{T} \sum_{j=1}^{T} 1(\hat{y}_t \neq y_t), \]  

   where \(\hat{y}_t\) is the label that Perceptron predicts for data point \(t\) as it runs, and \(1\) is an indicator function, which is 1 if its condition is true and 0 otherwise. The plot will show this average loss every 100 iterations, e.g. \([100, 200, 300, \ldots]\).

2. **(10 Points)** For a positive integer \(d\), the polynomial kernel

   \[ K^d_\text{p}(u, v) = (u \cdot v + 1)^d \]  

   maps \(X\) into a feature space of all polynomials of degree up to \(d\). We will run the perceptron algorithm for the kernel exponents \(d \in [1, 3, 5, 7, 10, 15, 20]\).

   - Fill in the function `make_polynomial_kernel`. This is a higher-order function which takes a degree \(d\) and returns a kernel function which computes \(K^d_\text{p}(u, v)\) for input vectors \(u, v\).
Now enter `python perceptron_template.py poly` at the command line. The program will run the perceptron algorithm for all the previously specified values of $d$ (for a single pass over the validation data set as before), and will save a plot `polynomial-kernel.png`. This plot will show the average loss for $T = 1000$ (that is, after all data points have been seen), for each kernel degree $d$.

- Note the value of $d$ that gave the lowest average loss over the validation set.

3. (10 Points) For $\sigma > 0$, the Exponential kernel

$$K_\sigma(x, y) = \exp\left[-\|x - y\|^2 / 2\sigma^2\right]$$

(12)

is a map to *all* polynomials of $x$, where $\sigma$ is a tuning constant that roughly corresponds to the “window size” of the exponential. For this problem, we’ll use $\sigma = 10$. In general, you have to experiment to find a good value of $\sigma$; this will depend on your data set.

We will compare the performance of an exponential kernel to the best-performing polynomial kernel from item 2 of this problem.

- Fill in the function `exponential_kernel`.

- In the function `run_poly_expon`, fill in the `best_degree` variable with whichever value of $d$ gave the best performance for the perceptron with a polynomial kernel.

- Enter `python perceptron_template.py expon` at the command line. The program will run the perceptron algorithm on the test set, using both a degree-$d$ polynomial kernel (for the $d$ you have selected) and an exponential kernel with $\sigma = 10$. It will output a plot of the average loss every 100 steps.

To wrap up, write a few sentences summarizing what you’ve observed. Which degree polynomial performed best? Which was better: the polynomial kernel or the exponential? Do these results match your expectations?