CIS 520, Machine Learning, Fall 2015: Assignment 6B
Due: Friday, Nov 6th, 2015. 11:59pm
[46 points]

Instructions. This is a MATLAB programming assignment. This assignment consists of multiple parts. Portions of each part will be graded automatically, and you can submit your code to be automatically checked for correctness to receive feedback ahead of time.

We are providing you with codebase / templates / dataset that you will require for this assignment. Download the file `hw6_kit.zip` from Canvas **before** beginning the assignment. **Please read through the documentation provided in ALL Matlab files before starting the assignment.** The instructions for submitting your homeworks and receiving automatic feedback are online on the wiki:

http://alliance.seas.upenn.edu/~cis520/wiki/index.php?n=Resources.HomeworkSubmission

If you are not familiar with Matlab or how Matlab functions work, you can refer to Matlab online documentation for help:

http://www.mathworks.com/help/matlab/

In addition, please use built-in Matlab functions rather than external library functions. Without proper reference to external library, auto-grader may fail even if your code runs perfectly on your local machine.

Collaboration. For this programming assignment, you must work individually. **The LOWEST grade will be recorded for multiple submissions.** Be sure to include your pennkey and name in the *Group.txt* file. Failure to do so will result in a failure in the grading process. **We will be using automatic checking software to detect blatant copying of other student's assignments, so, please, don't do it.**

1 Kernels for SVM [16 points]

Description. In this problem you'll be comparing three commonly used kernels. You will not be responsible for programming how to solve the SVM optimization problem given the kernel; `libsvm` will do that for you. We provide the function `kernel_libsvm` that will handle training and testing using libsvm using a kernel function that you will provide.

For SVMs, we will study a classic natural language processing task: newsgroup post classification. A newsgroup post is an old-fashioned internet forum posting consisting of raw text. The data we have given you contains posts from two newsgroups, one concerning Mac computers and the other concerning Windows PCs. The task is to classify which newsgroup a given post belongs on only from the text content of the posting.

The data is given to you in the following format. For each example, you have a Matlab struct with three fields: `raw`, `counts`, and `label`. The field `raw` is a cell array containing each line of the file and is not used, but is just for fun so you can read the postings and get a sense of the data. The field `counts` is a 2-column matrix: the first column is the index of a given word that occurs in the document, and the second column is the number of times that word appeared. The *index* of the word looks up into the `vocab` cell array, which is a list of all approximately 49,000 words that appear in the corpus.

Your task. Your task is to convert the count data into a form that can be used by SVM. We will use one feature per word, where the value of the feature is the number of times that word appears in the given example. Unfortunately, because there are more than 49,000 features, our data matrix X would be huge (around 500 MB in memory). However, observe that only a small subset of the features are ever active for a given example. Therefore we will use a *sparse matrix representation* (Matlab has a built in data structure for this) that only requires as much memory as there are non-zero examples.

1. **[4 points]** Your first task is to fill in the function `make_ sparse.m`, which builds a sparse X matrix from the data structures we gave you. More details on how to create a sparse matrix in Matlab are given in the `make_sparse.m`. Once the sparse matrix is created, we can use it just like a regular matrix, but it will be far more effcient.

2. Your next task is to write functions that compute three types of kernels to be plugged into `libsvm`. The arguments are *two sparse data matrices* $X_1 \in \mathbb{R}^{n_1 \times m}$ and $X_2 \in \mathbb{R}^{n_2 \times m}$, and some parameters when needed. The functions will return a Kernel matrix $K \in \mathbb{R}^{n_2 \times n_1}$ of similarities between samples of X_2 and X_1. Note the following:

   - $(K)_{ij} = k(\mathbf{x_{i,2}}, \mathbf{x_{j,1}})$ where $\mathbf{x_{i,2}}$ is the i^{th} sample of X_2, $\mathbf{x_{j,1}}$ is the j^{th} sample of X_1.
   - At training time, you will call the kernel functions with X_1 and X_2 both equal to the training data matrix X_{train}, and the output K_{train} will be a square matrix.
   - At testing time, you will call the function with $X_1 = X_{\text{train}}$ and $X_2 = X_{\text{test}}$ and the output K_{test} will not be square: $K_{\text{test}} \in \mathbb{R}^{\#\text{test} \times \#\text{train}}$ (the i_{th} *row* is $\mathbf{k}(x_{i,\text{test}})$).

   Keeping the above in mind, you need to fill in code in the following three files:

   - **[2 points]** `kernel_poly.m` In this file, compute the *polynomial* kernel matrix K such that:

     $$k(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b})^d$$

     where d, the degree, is a parameter of the function. Note that the case $d = 1$ corresponds to a basic linear kernel (inner product of the samples).
   - **[4 points]** `kernel_gaussian.m` In this file, compute the *Gaussian* kernel matrix K such that:

     $$k(\mathbf{a}, \mathbf{b}) = \exp\left(-\frac{\|\mathbf{a} - \mathbf{b}\|_2^2}{2\sigma^2}\right)$$

     where σ, the width, is a parameter of the function.
   - **[4 points]** `kernel_intersection.m` In this file, compute the *histogram intersection* kernel matrix K such that:

     $$k(\mathbf{a}, \mathbf{b}) = \sum_{j=1}^{m} \min(\mathbf{a}_j, \mathbf{b}_j)$$

     where j indexes the word bins (m is the number of features, i.e. the width of X_1 and X_2) This kernel does not have any parameters. Intuitively, since the features you will use are histograms (word frequencies), two documents that have similar types of words will have peaks in the same histogram bins and their intersection kernel score should be high.

3. **[2 points]** Finally, fill in the code in `generate_svm_plots.m` to evaluate and compare the kernel methods using `kernel_libsvm`. Make sure to read `kernel_libsvm` before proceeding. At the end of the plot generation code, you will generate a bar graph showing the errors of each kernel. Which kernel works best? Propose one reason why that kernel might work better than the others.

2 Programming Perceptrons [30 points]

Description. SVMs are incredibly powerful, but solving the quadratic program globally for a large dataset can be difficult to implement. As you saw in the previous problem, the matrices involved can get very large and memory intensive. Sometimes it can be useful to take a simpler approach: Perceptrons. Unlike SVMs, perceptrons are trained on each example one at a time, making the optimization problem much simpler to solve.

In this problem, you will be training perceptrons on the exact same dataset used with SVMs so you can directly compare the results. Since the perceptron algorithm is much simpler than the SVM optimization, you will be responsible for writing the optimization routine itself.

Your task. Turning back to our old friend, breast cancer data set (already partitioned this time), you will implement an averaged perceptron algorithm with two updating strategies. You will also compare your results to the linear SVM you implemented earlier.

1. **[16 points]** Your first task is to finish the function `averaged_perceptron_train.m`, which implements the overall structure of the averaged perceptron algorithm and returns a Dx1 averaged hyperplane vector $\vec{w}$ obtained, where D is the number of features. Note that this weight vector now includes the bias term w_0 so to compute $\vec{w} \cdot \vec{x}_i$ so we need to pad $\vec{x}_i$ with a constant feature (column of 1's). This has been done for you in `generate_perceptron_plots.m`.

   Instead of computing the update step within this function, you will pass a function handler as an argument to `averaged_perceptron_train.m`. In your inner loop, you should have something to the effect of

   $$\vec{w} = \vec{w} + update\_step\_function(\vec{x}_i, y_i, \vec{w}) * \vec{x}_i$$

   You will also return the training error obtained for every averaged model $\vec{w}_ave$ along the way. The function `perceptron_error.m` is provided to efficiently calculate the training error at each step. You are not required to use this function, but it's recommended.

   The function `averaged_perceptron_train.m` will stop after passing the whole training set `numPasses` times, which is the last argument.

   The best part about an averaged perceptron is you can maintain a running accumulator of the weights used at each step and divide by the number of iterations at the end of training. **Do NOT keep a record of every weight vector you obtain and then use mean(). That will require a LOT of memory.**

2. Your next task is to write functions that compute two types of update steps to be plugged into `averaged_perceptron_train.m`. The arguments are the padded 1xD feature vector of the current example $\vec{x}$, the example's true label y, the current model $\vec{w}$.The functions will return a scalar describing the magnitude of update step

   $$\vec{w} = \vec{w} + update\_step(\vec{x}_i, y_i, \vec{w}) * \vec{x}_i$$

   You need to fill in code in the following two files:

   - **[2 points]** `update_constant.m` In this file, compute the *constant learning rate* update such that:

     $$update\_constant(x, y, w, \eta) = \eta * (y - sign(\vec{w} \cdot \vec{x}))$$

   - **[4 points]** `update_passive_aggressive.m` In this file, compute the *passive aggressive learning rate* update such that:

     $$update\_passive\_aggressive(x, y, w) = \eta * y$$

     where

     $$\eta = L/|x|^2$$

     and

     $$L = \begin{cases} 0 & : y * \vec{w} \cdot \vec{x} >= 1 \\ 1 - y * \vec{w} \cdot \vec{x} & : otherwise \end{cases}$$

- **[8 points]** Finally, use code in `generate_perceptron_plots.m` to evaluate and compare the learning methods. Details of the plots to generate are included in the file comments. Include your answers to the following questions in the provided field.
  (a) Which updating strategy converges more quickly?
  (b) Which updating strategy has lowest training error?
  (c) Which updating strategy has lowest test error?
  (d) How does the best test error compare to linear SVM on the same dataset?