CS 229, Spring 2016
Problem Set #2: Naive Bayes, SVMs, and Theory

Due Wednesday, May 4 at 11:00 pm on Gradescope.

Notes: (1) These questions require thought, but do not require long answers. Please be as concise as possible. (2) If you have a question about this homework, we encourage you to post your question on our Piazza forum, at `https://piazza.com/stanford/spring2016/cs229`. (3) If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on Handout #1 (available from the course website) before starting work. (4) For problems that require programming, please include in your submission a printout of your code (with comments) and any figures that you are asked to plot.

If you are scanning your document by cellphone, please check the Piazza forum for recommended cellphone scanning apps and best practices.

1. **[15 points] Constructing kernels**

   In class, we saw that by choosing a kernel $K(x, z) = \phi(x)^T \phi(z)$, we can implicitly map data to a high dimensional space, and have the SVM algorithm work in that space. One way to generate kernels is to explicitly define the mapping ϕ to a higher dimensional space, and then work out the corresponding K.

   However in this question we are interested in direct construction of kernels. I.e., suppose we have a function $K(x, z)$ that we think gives an appropriate similarity measure for our learning problem, and we are considering plugging K into the SVM as the kernel function. However for $K(x, z)$ to be a valid kernel, it must correspond to an inner product in some higher dimensional space resulting from some feature mapping ϕ. Mercer's theorem tells us that $K(x, z)$ is a (Mercer) kernel if and only if for any finite set $\{x^{(1)}, \ldots, x^{(m)}\}$, the matrix K is symmetric and positive semidefinite, where the square matrix $K \in \mathbb{R}^{m \times m}$ is given by $K_{ij} = K(x^{(i)}, x^{(j)})$.

   Now here comes the question: Let K_1, K_2 be kernels over $\mathbb{R}^n \times \mathbb{R}^n$, let $a \in \mathbb{R}^+$ be a positive real number, let $f : \mathbb{R}^n \mapsto \mathbb{R}$ be a real-valued function, let $\phi : \mathbb{R}^n \to \mathbb{R}^d$ be a function mapping from $\mathbb{R}^n$ to $\mathbb{R}^d$, let K_3 be a kernel over $\mathbb{R}^d \times \mathbb{R}^d$, and let $p(x)$ a polynomial over x with *positive* coefficients.

   For each of the functions K below, state whether it is necessarily a kernel. If you think it is, prove it; if you think it isn't, give a counter-example.

   (a) [1 points] $K(x, z) = K_1(x, z) + K_2(x, z)$

   (b) [1 points] $K(x, z) = K_1(x, z) - K_2(x, z)$

   (c) [1 points] $K(x, z) = aK_1(x, z)$

   (d) [1 points] $K(x, z) = -aK_1(x, z)$

   (e) [5 points] $K(x, z) = K_1(x, z)K_2(x, z)$

   (f) [2 points] $K(x, z) = f(x)f(z)$

   (g) [2 points] $K(x, z) = K_3(\phi(x), \phi(z))$

   (h) [2 points] $K(x, z) = p(K_1(x, z))$

[Hint: For part (e), the answer is that the K there *is* indeed a kernel. You still have to prove it, though. (This one may be harder than the rest.) This result may also be useful for another part of the problem.]

Answer: All 8 cases of proposed kernels K are trivially symmetric because K_1, K_2, K_3 are symmetric; and because the product of 2 real numbers is commutative (for (1f)). Thanks to Mercer's theorem, it is sufficient to prove the corresponding properties for positive semidefinite matrices. To differentiate between matrix and kernel function, we'll use G_i to denote a kernel matrix (Gram matrix) corresponding to a kernel function K_i.

(a) Kernel. The sum of 2 positive semidefinite matrices is a positive semidefinite matrix: $\forall z\ z^T G_1 z \geq 0, z^T G_2 z \geq 0$ since K_1, K_2 are kernels. This implies $\forall z\ z^T G z = z^T G_1 z + z^T G_2 z \geq 0$.

(b) Not a kernel. Counterexample: let $K_2 = 2K_1$ (we are using (1c) here to claim K_2 is a kernel). Then we have $\forall z\ z^T G z = z^T (G_1 - 2G_1) z = -z^T G_1 z \leq 0$.

(c) Kernel. $\forall z\ z^T G_1 z \geq 0$, which implies $\forall z\ a z^T G_1 z \geq 0$.

(d) Not a kernel. Counterexample: $a = 1$. Then we have $\forall z\ -z^T G_1 z \leq 0$.

(e) Kernel. K_1 is a kernel, thus $\exists \phi^{(1)}\ K_1(x,z) = \phi^{(1)}(x)^T \phi^{(1)}(z) = \sum_i \phi_i^{(1)}(x)\phi_i^{(1)}(z)$. Similarly, K_2 is a kernel, thus $\exists \phi^{(2)}\ K_2(x,z) = \phi^{(2)}(x)^T \phi^{(2)}(z) = \sum_j \phi_j^{(2)}(x)\phi_j^{(2)}(z)$.

$$
\begin{aligned}
K(x,z) &= K_1(x,z) K_2(x,z) && (1)\\
&= \sum_i \phi_i^{(1)}(x)\phi_i^{(1)}(z) \sum_i \phi_i^{(2)}(x)\phi_i^{(2)}(z) && (2)\\
&= \sum_i \sum_j \phi_i^{(1)}(x)\phi_i^{(1)}(z)\phi_j^{(2)}(x)\phi_j^{(2)}(z) && (3)\\
&= \sum_i \sum_j (\phi_i^{(1)}(x)\phi_j^{(2)}(x))(\phi_i^{(1)}(z)\phi_j^{(2)}(z)) && (4)\\
&= \sum_{(i,j)} \psi_{i,j}(x)\psi_{i,j}(z) && (5)
\end{aligned}
$$

Where the last equality holds because that's how we define ψ. We see K can be written in the form $K(x,z) = \psi(x)^T \psi(z)$ so it is a kernel.

Here is an alternate super-slick linear-algebraic proof. If G is the Gram matrix for the product $K_1 \times K_2$, then G is a principal submatrix of the Kronecker product $G_1 \otimes G_2$, where G_i is the Gram matrix for K_i. As the Kronecker product is positive semi-definite, so are its principal submatrices.

(f) Kernel. Just let $\psi(x) = f(x)$, and since $f(x)$ is a scalar, we have $K(x,z) = \phi(x)^T \phi(z)$ and we are done.

(g) Kernel. Since K_3 is a kernel, the matrix G_3 obtained for *any* finite set $\{x^{(1)}, \ldots, x^{(m)}\}$ is positive semidefinite, and so it is also positive semidefinite for the sets $\{\phi(x^{(1)}), \ldots, \phi(x^{(m)})\}$.

(h) Kernel. By combining (1a) sum, (1c) scalar product, (1e) powers, (1f) constant term, we see that any polynomial of a kernel K_1 will again be a kernel.

2. **[10 points] Kernelizing the Perceptron**

Let there be a binary classification problem with $y \in \{-1, 1\}$. The perceptron uses hypotheses of the form $h_\theta(x) = g(\theta^T x)$, where $g(z) = \text{sign}(z) = 1$ if $z \geq 0$, -1 otherwise.

In this problem we will consider a stochastic gradient descent-like implementation of the perceptron algorithm where each update to the parameters θ is made using only one training example. However, unlike stochastic gradient descent, the perceptron algorithm will only make one pass through the entire training set. The update rule for this version of the perceptron algorithm is given by

$$\theta^{(i+1)} := \begin{cases} \theta^{(i)} + \alpha y^{(i+1)} x^{(i+1)} & \text{if } h_{\theta^{(i)}}(x^{(i+1)}) y^{(i+1)} < 0 \\ \theta^{(i)} & \text{otherwise,} \end{cases}$$

where $\theta^{(i)}$ is the value of the parameters after the algorithm has seen the first i training examples. Prior to seeing any training examples, $\theta^{(0)}$ is initialized to $\vec{0}$.

Let K be a Mercer kernel corresponding to some very high-dimensional feature mapping ϕ. Suppose ϕ is so high-dimensional (say, ∞-dimensional) that it's infeasible to ever represent $\phi(x)$ explicitly. Describe how you would apply the "kernel trick" to the perceptron to make it work in the high-dimensional feature space ϕ, but without ever explicitly computing $\phi(x)$. [Note: You don't have to worry about the intercept term. If you like, think of ϕ as having the property that $\phi_0(x) = 1$ so that this is taken care of.] Your description should specify

(a) How you will (implicitly) represent the high-dimensional parameter vector $\theta^{(i)}$, including how the initial value $\theta^{(0)} = \vec{0}$ is represented (note that $\theta^{(i)}$ is now a vector whose dimension is the same as the feature vectors $\phi(x)$);

(b) How you will efficiently make a prediction on a new input $x^{(i+1)}$. I.e., how you will compute $h_{\theta^{(i)}}(x^{(i+1)}) = g({\theta^{(i)}}^T \phi(x^{(i+1)}))$, using your representation of $\theta^{(i)}$; and

(c) How you will modify the update rule given above to perform an update to θ on a new training example $(x^{(i+1)}, y^{(i+1)})$; i.e., using the update rule corresponding to the feature mapping ϕ:

$$\theta^{(i+1)} := \theta^{(i)} + \alpha \mathbf{1}\{{\theta^{(i)}}^T \phi(x^{(i+1)}) y^{(i+1)} < 0\} y^{(i+1)} \phi(x^{(i+1)}).$$

[Hint: our discussion of the representer theorem may be useful.]

Answer:

In the high-dimensional space we update θ as follows:

$$\theta := \theta + \alpha(y^{(i)} - h_\theta(\phi(x^{(i)})))\phi(x^{(i)})$$

So (assuming we initialize $\theta^{(0)} = \vec{0}$) θ will always be a linear combination of the $\phi(x^{(i)})$, i.e., $\exists \beta_l$ such that $\theta^{(i)} = \sum_{l=1}^{i} \beta_l \phi(x^{(l)})$ after having incorporated i training points. Thus $\theta^{(i)}$ can be compactly represented by the coefficients β_l of this linear combination, i.e., i real numbers after having incorporated i training points $x^{(i)}$. The initial value $\theta^{(0)}$ simply corresponds to the case where the summation has no terms (i.e., an empty list of coefficients β_l).

We do not work explicitly in the high-dimensional space, but use the fact that $g({\theta^{(i)}}^T \phi(x^{(i+1)})) = g(\sum_{l=1}^{i} \beta_l \cdot \phi(x^{(l)})^T \phi(x^{i+1})) = g(\sum_{l=1}^{i} \beta_l K(x^{(l)}, x^{(i+1)}))$, which can be computed efficiently.

We can efficiently update θ. We just need to compute $\beta_i = \alpha(y^{(i)} - g({\theta^{(i-1)}}^T \phi(x^{(i)})))$ at iteration i. This can be computed efficiently, if we compute ${\theta^{(i-1)}}^T \phi(x^{(i)})$ efficiently as described above.

In an alternative approach, one can observe that, unless a sample $\phi(x^{(i)})$ is misclassified, $y^{(i)} - h_{\theta^{(i)}}(\phi(x^{(i)}))$ will be zero; otherwise, it will be ± 1 (or ± 2, if the convention $y, h \in \{-1, 1\}$ is taken). The vector θ, then, can be represented as the sum $\sum_{\{i:y^{(i)} \neq h_{\theta^{(i)}}(\phi(x^{(i)}))\}} \alpha(2y^{(i)} - 1)\phi(x^{(i)})$ under the $y, h \in \{0, 1\}$ convention, and containing $(2y^{(i)})$ under the other convention. This can then be expressed as $\theta^{(i)} = \sum_{i \in \text{Misclassified}} \beta_i \phi(x^{(i)})$ to be in more obvious congruence with the above. The efficient representation can now be said to be a list which stores only those indices that were misclassified, as the β_is can be recomputed from the $y^{(i)}$s and α on demand. The derivation for (b) is then only cosmetically different, and in (c) the update rule is to add $(i + 1)$ to the list if $\phi(x^{(i+1)})$ is misclassified.

3. **[30 points] Spam classification**

   In this problem, we will use the naive Bayes algorithm and an SVM to build a spam classifier.

   In recent years, spam on electronic newsgroups has been an increasing problem. Here, we'll build a classifier to distinguish between "real" newsgroup messages, and spam messages. For this experiment, we obtained a set of spam emails, and a set of genuine newsgroup messages.[1] Using only the subject line and body of each message, we'll learn to distinguish between the spam and non-spam.

   All the files for the problem are in `http://cs229.stanford.edu/materials/spam_data.tgz`. **Note: Please do not circulate this data outside this class.** In order to get the text emails into a form usable by naive Bayes, we've already done some preprocessing on the messages. You can look at two sample spam emails in the files `spam_sample_original*`, and their preprocessed forms in the files `spam_sample_preprocessed*`. The first line in the preprocessed format is just the label and is not part of the message. The preprocessing ensures that only the message body and subject remain in the dataset; email addresses (EMAILADDR), web addresses (HTTPADDR), currency (DOLLAR) and numbers (NUMBER) were also replaced by the special tokens to allow them to be considered properly in the classification process. (In this problem, we'll going to call the features "tokens" rather than "words," since some of the features will correspond to special values like EMAILADDR. You don't have to worry about the distinction.) The files `news_sample_original` and `news_sample_preprocessed` also give an example of a non-spam mail.

   The work to extract feature vectors out of the documents has also been done for you, so you can just load in the design matrices (called document-word matrices in text classification) containing all the data. In a document-word matrix, the i^{th} row represents the i^{th} document/email, and the j^{th} column represents the j^{th} distinct token. Thus, the (i, j)-entry of this matrix represents the number of occurrences of the j^{th} token in the i^{th} document.

   For this problem, we've chosen as our set of tokens considered (that is, as our vocabulary) only the medium frequency tokens. The intuition is that tokens that occur too often or too rarely do not have much classification value. (Examples tokens that occur very often are words like "the," "and," and "of," which occur in so many emails and are sufficiently content-free that they aren't worth modeling.) Also, words were stemmed using a standard stemming algorithm; basically, this means that "price," "prices" and "priced" have all been replaced with "price," so that they can be treated as the same word. For a list of the tokens used, see the file `TOKENS_LIST`.

[1] Thanks to Christian Shelton for providing the spam email. The non-spam messages are from the 20 newsgroups data at `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/news20.html`.

Since the document-word matrix is extremely sparse (has lots of zero entries), we have stored it in our own efficient format to save space. You don't have to worry about this format.[2] The file `readMatrix.m` provides the `readMatrix` function that reads in the document-word matrix and the correct class labels for the various documents. Code in `nb_train.m` and `nb_test.m` shows how `readMatrix` should be called. The documentation at the top of these two files will tell you all you need to know about the setup.

(a) [11 points] Implement a naive Bayes classifier for spam classification, using the multinomial event model and Laplace smoothing.

You should use the code outline provided in `nb_train.m` to train your parameters, and then use these parameters to classify the test set data by filling in the code in `nb_test.m`. You may assume that any parameters computed in `nb_train.m` are in memory when `nb_test.m` is executed, and do not need to be recomputed (i.e., that `nb_test.m` is executed immediately after `nb_train.m`) [3].

Train your parameters using the document-word matrix in `MATRIX.TRAIN`, and then report the test set error on `MATRIX.TEST`.

Remark. If you implement naive Bayes the straightforward way, you'll find that the computed $p(x|y) = \prod_i p(x_i|y)$ often equals zero. This is because $p(x|y)$, which is the product of many numbers less than one, is a very small number. The standard computer representation of real numbers cannot handle numbers that are too small, and instead rounds them off to zero. (This is called "underflow.") You'll have to find a way to compute naive Bayes' predicted class labels without explicitly representing very small numbers such as $p(x|y)$. [Hint: Think about using logarithms.]

(b) [3 points] Intuitively, some tokens may be particularly indicative of an email being in a particular class. We can try to get an informal sense of how indicative token i is for the SPAM class by looking at:

$$\log \frac{p(x_j = i|y = 1)}{p(x_j = i|y = 0)} = \log \left(\frac{P(\text{token } i|\text{email is SPAM})}{P(\text{token } i|\text{email is NOTSPAM})} \right).$$

Using the parameters fit in part (a), find the 5 tokens that are most indicative of the SPAM class (i.e., have the highest positive value on the measure above). The numbered list of tokens in the file `TOKENS_LIST` should be useful for identifying the words/tokens.

(c) [3 points] Repeat part (a), but with training sets of size ranging from 50, 100, 200, ..., up to 1400, by using the files `MATRIX.TRAIN.*`. Plot the test error each time (use `MATRIX.TEST` as the test data) to obtain a learning curve (test set error vs. training set size). You may need to change the call to `readMatrix` in `nb_train.m` to read the correct file each time. Which training-set size gives the best test set error?

(d) [11 points] Train an SVM on this dataset using stochastic gradient descent and the radial basis function (also known as the Gaussian) kernel, which sets

$$K(x, z) = \exp \left(-\frac{1}{2\tau^2} \|x - z\|_2^2 \right).$$

[2] Unless you're not using Matlab/Octave, in which case feel free to ask us about it. We have provided Julia code to read the file in `MatrixReading.jl`.

[3] Matlab note: If a .m file doesn't begin with a function declaration, the file is a script. Variables in a script are put into the global namespace, unlike with functions.

In this case, recall that (as proved in class) the objective with kernel matrix $K = [K^{(1)} \ \cdots \ K^{(m)}] \in \mathbb{R}^{m \times m}$ is given by

$$J(\alpha) = \frac{1}{m} \sum_{i=1}^{m} \left[1 - y^{(i)} {K^{(i)}}^T \alpha\right]_+ + \frac{\lambda}{2} \alpha^T K \alpha$$

where $[t]_+ = \max\{t, 0\}$ is the positive part function. In this case, the gradient (actually, this is known as a *subgradient*) of the individual loss terms is

$$\nabla_\alpha \left[1 - y^{(i)} K^{(i)} \alpha\right]_+ = \begin{cases} -y^{(i)} K^{(i)} & \text{if } y^{(i)} {K^{(i)}}^T \alpha < 1 \\ 0 & \text{otherwise.} \end{cases}$$

In your SVM training, you should perform stochastic gradient descent, where in each iteration you choose an index $i \in \{1, \ldots, m\}$ uniformly at random, for a total of $40 \cdot m$ steps, where m is the training set size, and your kernel should use $\tau = 8$ and regularization multiplier $\lambda = \frac{1}{64m}$. For this part of the problem, you should also replace each training or test point $x^{(i)}$ with a zero-one vector $z^{(i)}$, where $z_j^{(i)} = 1$ if $x_j^{(i)} > 0$ and $z_j^{(i)} = 0$ if $x_j^{(i)} = 0$. Initialize your SGD procedure at $\alpha = 0$.

The output of your training code, which you should implement in `svm_test.m`, should be the α vector that is the *average* of all the α vectors that your iteration updates. At iteration t of stochastic gradient descent you should use stepsize $1/\sqrt{t}$.

Similar to part (c), train an SVM with training set sizes 50, 100, 200, ..., 1400, by using the file `MATRIX.TRAIN.50` and so on. Plot the test error each time, using `MATRIX.TEST` as the test data.

(A few hints for more efficient Matlab code: you should try to vectorize creation of the Kernel matrix, and you should call the method `full` to make the matrix non-sparse, which will make the method faster. In addition, the training data uses labels in $\{0, 1\}$, so you should change the output of the `readMatrix` method to have labels $y \in \{-1, 1\}$.)

(e) [2 points] How do naive Bayes and Support Vector Machines compare (in terms of generalization error) as a function of the training set size?

Answer:

(a) The test error when training on the full training set was 1.63%. If you got a different error (or if you got the words `website` and `lowest` for part b), you most probably implemented the wrong Naive Bayes model.

(b) The five most indicative words for the spam class were: `httpaddr`, `spam`, `unsubscrib`, `ebai` and `valet`.

(c) The test set error for different training set set sizes was:

i. Training set size 50: Test set error = 3.87%
ii. Training set size 100: Test set error = 2.62%
iii. Training set size 200: Test set error = 2.62%
iv. Training set size 400: Test set error = 1.87%
v. Training set size 800: Test set error = 1.75%
vi. Training set size 1400: Test set error = 1.63%

vii. Full training set: Test set error = 1.63%

(d) The test set error from the SVM for different training set sizes, averaged over 10 randomizations of the training data order, was:

i. Training set size 50: Test set error = 2.26%
ii. Training set size 100: Test set error = 1.47%
iii. Training set size 200: Test set error = .26%
iv. Training set size 400: Test set error = .14%
v. Training set size 800: Test set error = 0%
vi. Training set size 1400: Test set error = 0%

(e) The deduction that can be drawn is that for this dataset, Naive Bayes is simply not as good as the Kernelized SVM.

The Matlab code for the problem:

```
% ------------------------------------------------------------------------ %
% ---------------               Outer  loop                --------------- %
% ------------------------------------------------------------------------ %

training_set_size_list = [50, 100, 200, 400, 800, 1400];

%% SVM training %%

test_errors_svm = zeros(length(training_set_size_list), 1);
total_svms_to_avg = 10;
[M, tokenlist, category] = readMatrix('MATRIX.TEST');
Xtest = M;
ytest = (2 * category - 1)';

for train_ind = 1:length(training_set_size_list)
  for iter = 1:total_svms_to_avg
    num_train = training_set_size_list(train_ind);
    svm_train;
    svm_test;
    test_errors_svm(train_ind) = test_errors_svm(train_ind) + test_error;
  end
end

test_errors_svm = test_errors_svm / total_svms_to_avg;

%% Naive Bayes training %%

test_errors_nb = zeros(length(training_set_size_list), 1);
for train_ind = 1:length(training_set_size_list)
  num_train = training_set_size_list(train_ind);
  nb_train;
  nb_test;
  test_errors_nb(train_ind) = error;
```

```
end

figure;
semilogx(training_set_size_list, test_errors_svm, 'bs-', 'linewidth', 2);
hold on;
semilogx(training_set_size_list, test_errors_nb, 'ko-', 'linewidth', 2);
legend('SVM error', 'NB error');
set(gca, 'fontsize', 18);
axis([min(training_set_size_list), max(training_set_size_list), 0, .04]);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% svm_train.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[sparseTrainMatrix, tokenlist, trainCategory] = ...
    readMatrix(sprintf('MATRIX.TRAIN.%d', num_train));
m_train = size(Xtrain, 1);
ytrain = (2 * trainCategory - 1)';
Xtrain = 1.0 * (Xtrain > 0);

squared_X_train = sum(Xtrain.^2, 2);
gram_train = Xtrain * Xtrain';
tau = 8;

% Get full training matrix for kernels using vectorized code.
Ktrain = full(exp(-(repmat(squared_X_train, 1, m_train) ...
                    + repmat(squared_X_train', m_train, 1) ...
                    - 2 * gram_train) / (2 * tau^2)));

lambda = 1 / (64 * m_train);
num_outer_loops = 40;
alpha = zeros(m_train, 1);

avg_alpha = zeros(m_train, 1);
Imat = eye(m_train);

count = 0;
for ii = 1:(num_outer_loops * m_train)
  count = count + 1;
  ind = ceil(rand * m_train);
  margin = ytrain(ind) * Ktrain(ind, :) * alpha;
  g = -(margin < 1) * ytrain(ind) * Ktrain(:, ind) + ...
      m_train * lambda * (Ktrain(:, ind) * alpha(ind));
  % g(ind) = g(ind) + m_train * lambda * Ktrain(ind,:) * alpha;
  alpha = alpha - g / sqrt(count);
  avg_alpha = avg_alpha + alpha;
end
avg_alpha = avg_alpha / (num_outer_loops * m_train);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% svm_test.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Construct test and train matrices
Xtest = 1.0 * (Xtest > 0);
squared_X_test = sum(Xtest.^2, 2);
m_test = size(Xtest, 1);
gram_test = Xtest * Xtrain';
Ktest = full(exp(-(repmat(squared_X_test, 1, m_train) ...
                   + repmat(squared_X_train', m_test, 1) ...
                   - 2 * gram_test) / (2 * tau^2)));

% preds = Ktest * alpha;

% fprintf(1, 'Test error rate for final alpha: %1.4f\n', ...
%         sum(preds .* ytest <= 0) / length(ytest));

preds = Ktest * avg_alpha;
fprintf(1, 'Test error rate for average alpha: %1.4f\n', ...
        sum(preds .* ytest <= 0) / length(ytest));
test_error = sum(preds .* ytest <= 0) / length(ytest);

[spmatrix, tokenlist, category] = readMatrix('MATRIX.TEST');

testMatrix = full(spmatrix);
numTestDocs = size(testMatrix, 1);
numTokens = size(testMatrix, 2);

% ...
output = zeros(numTestDocs, 1);

%---------------
% YOUR CODE HERE

for k=1:numTestDocs,
  [i,j,v] = find(testMatrix(k,:));
  neg_posterior = sum(v .* neg_log_phi(j)) + neg_log_prior;
  pos_posterior = sum(v .* pos_log_phi(j)) + pos_log_prior;

  if (neg_posterior > pos_posterior)
    output(k) = 0;
  else
    output(k) = 1;
  end
end

%---------------
```

```
y = full(category);
y = y(:);
% Compute the error on the test set
error = sum(y ~= output) / numTestDocs;

%Print out the classification error on the test set
fprintf(1, 'Test error: %1.4f\n', error);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% nb_train.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


[spmatrix, tokenlist, trainCategory] = ...
    readMatrix(sprintf('MATRIX.TRAIN.%d', num_train));

trainMatrix = full(spmatrix);
numTrainDocs = size(trainMatrix, 1);
numTokens = size(trainMatrix, 2);

% ...
% YOUR CODE HERE

V = size(trainMatrix, 2);
neg = trainMatrix(find(trainCategory == 0), :);
pos = trainMatrix(find(trainCategory == 1), :);

neg_words = sum(sum(neg));
pos_words = sum(sum(pos));

neg_log_prior = log(size(neg,1) / numTrainDocs);
pos_log_prior = log(size(pos,1) / numTrainDocs);

for k=1:V,
  neg_log_phi(k) = log((sum(neg(:,k)) + 1) / (neg_words + V));
  pos_log_phi(k) = log((sum(pos(:,k)) + 1) / (pos_words + V));
end
```

4. **[20 points] Properties of VC dimension**

   In this problem, we investigate a few properties of the Vapnik-Chervonenkis dimension, mostly relating to how VC(H) increases as the set H increases. For each part of this problem, you should state whether the given statement is true, and justify your answer with either a formal proof or a counter-example.

   (a) Let two hypothesis classes H_1 and H_2 satisfy $H_1 \subseteq H_2$. Prove or disprove: $\text{VC}(H_1) \leq \text{VC}(H_2)$.

(b) Let $H_1 = H_2 \cup \{h_1, \ldots, h_k\}$. (I.e., H_1 is the union of H_2 and some set of k additional hypotheses.) Prove or disprove: $\text{VC}(H_1) \leq \text{VC}(H_2) + k$. [Hint: You might want to start by considering the case of $k = 1$.]

(c) Let $H_1 = H_2 \cup H_3$. Prove or disprove: $\text{VC}(H_1) \leq \text{VC}(H_2) + \text{VC}(H_3)$.

Answer:

(a) True. Suppose that $VC(H_1) = d$. Then there exists a set of d points that is shattered by H_1 (i.e., for each possible labeling of the d points, there exists a hypothesis $h \in H_1$ which realizes that labeling). Now, since H_2 contains all hypotheses in H_1, then H_2 shatters the same set, and thus we have $VC(H_2) \geq d = VC(H_1)$.

(b) True. If we can prove the result for $k = 1$, then the result stated in the problem set follows immediately by applying the same logic inductively, one hypothesis at a time. So, let us prove that if $H_1 = H_2 \cup \{h\}$, then $VC(H_1) \leq VC(H_2) + 1$. Suppose that $VC(H_1) = d$, and let S_1 be a set of d points that is shattered by H_1. Now, pick an arbitrary $x \in S_1$. Since H_1 shatters S_1, there must be some $\bar{h} \in H_1$ such that h and $\bar{h}$ agree on labelings for all points in S_1 except x. This means that $H' := H_1 \setminus \{h\}$ achieves all possible labelings on $S' := S_1 \setminus \{x\}$ (i.e. H' shatters S'), so $VC(H') \geq |S'| = d - 1$. But $H' \subseteq H_2$, so from part (a), $VC(H') \leq VC(H_2)$. It follows that $VC(H_2) \geq d - 1$, or equivalently, $VC(H_1) \leq VC(H_2) + 1$, as desired.

For this problem, there were a number of possible correct proof methods; generally, to get full credit, you needed to argue formally that there exists no set of $(VC(H_2) + 2)$ points shattered by H_1, or equivalently, that there always exists a set of $(VC(H_1) - 1)$ points shattered by H_2. Here are a couple of the more common errors:

- Some submitted solutions stated that adding a single hypothesis to H_2 increases the VC dimension by at most one, since the new hypothesis can only realize a single labeling. While this statement is vaguely true, it is neither sufficiently precise, nor is its correctness immediately obvious.
- Some solutions made arguments relating to the cardinality of the sets H_1 and H_2. However, generally when we speak about VC dimension, the sets H_1 and H_2 often have infinite cardinality (e.g., the set of all linear classifiers in $\mathbb{R}^2$).

(c) False. Counterexample: let $H_1 = \{h_1\}, H_2 = \{h_2\}$, and $\forall x,\ h_1(x) = 0, h_2(x) = 1$. Then we have $VC(H_1) = VC(H_2) = 0$, but $VC(H_1 \cup H_2) = 1$.

5. **[20 points] Training and testing on different distributions**

In the discussion in class about learning theory, a key assumption was that we trained and tested our learning algorithms on the same distribution $\mathcal{D}$. In this problem, we'll investigate one special case of training and testing on different distributions. Specifically, we will consider what happens when the training labels are *noisy*, but the test labels are not.

Consider a binary classification problem with labels $y \in \{0, 1\}$, and let $\mathcal{D}$ be a distribution over (x, y), that we'll think of as the original, "clean" or "uncorrupted" distribution. Define $\mathcal{D}_\tau$ to be a "corrupted" distribution over (x, y) which is the same as $\mathcal{D}$, except that the labels y have some probability $0 \leq \tau < 0.5$ of being flipped. Thus, to sample from $\mathcal{D}_\tau$, we would first sample (x, y) from $\mathcal{D}$, and then with probability τ (independently of the observed x and y) replace y with $1 - y$. Note that $\mathcal{D}_0 = \mathcal{D}$.

The distribution $\mathcal{D}_\tau$ models a setting in which an unreliable human (or other source) is labeling your training data for you, and on each example he/she has a probability τ of mislabeling it. Even though our training data is corrupted, we are still interested in evaluating our hypotheses with respect to the original, uncorrupted distribution $\mathcal{D}$.

We define the generalization error *with respect to* $\mathcal{D}_\tau$ to be

$$\varepsilon_\tau(h) = P_{(x,y)\sim\mathcal{D}_\tau}[h(x) \neq y].$$

Note that $\varepsilon_0(h)$ is the generalization error with respect to the "clean" distribution; it is with respect to ε_0 that we wish to evaluate our hypotheses.

(a) For any hypothesis h, the quantity $\varepsilon_0(h)$ can be calculated as a function of $\varepsilon_\tau(h)$ and τ. Write down a formula for $\varepsilon_0(h)$ in terms of $\varepsilon_\tau(h)$ and τ, and justify your answer.

(b) Let $|H|$ be finite, and suppose our training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$ is obtained by drawing m examples IID from the corrupted distribution $\mathcal{D}_\tau$. Suppose we pick $h \in H$ using empirical risk minimization: $\hat{h} = \arg\min_{h\in H} \hat{\varepsilon}_S(h)$. Also, let $h^* = \arg\min_{h\in H} \varepsilon_0(h)$.

Let any $\delta, \gamma > 0$ be given. Prove that for

$$\varepsilon_0(\hat{h}) \leq \varepsilon_0(h^*) + 2\gamma$$

to hold with probability $1 - \delta$, it suffices that

$$m \geq \frac{1}{2(1-2\tau)^2\gamma^2} \log \frac{2|H|}{\delta}.$$

Remark. This result suggests that, roughly, m examples that have been corrupted at noise level τ are worth about as much as $(1-2\tau)^2 m$ uncorrupted training examples. This is a useful rule-of-thumb to know if you ever need to decide whether/how much to pay for a more reliable source of training data. (If you've taken a class in information theory, you may also have heard that $(1-\mathcal{H}(\tau))m$ is a good estimate of the information in the m corrupted examples, where $\mathcal{H}(\tau) = -(\tau \log_2 \tau + (1-\tau)\log_2(1-\tau))$ is the "binary entropy" function. And indeed, the functions $(1-2\tau)^2$ and $1-\mathcal{H}(\tau)$ are quite close to each other.)

(c) Comment **briefly** on what happens as τ approaches 0.5.

Answer:

(a) We compute ε_τ as a function of ε_0 and then invert the obtained expression. An error occurs on the corrupted distribution, if and only if, an error occurred for the original distribution and the point that was not corrupted, or no error occurred for the original distribution but the point was corrupted. So we have

$$\varepsilon_\tau = \varepsilon_0(1-\tau) + (1-\varepsilon_0)\tau$$

Solving for ε_0 gives

$$\varepsilon_0 = \frac{\varepsilon_\tau - \tau}{1-2\tau}$$

(b) We will need to apply the following (in the right order):

$$\forall h \in H,\ |\varepsilon_\tau(h) - \hat{\varepsilon}_\tau(h)| \leq \bar{\gamma} \quad \text{w.p.}(1-\delta), \quad \delta = 2K\exp(-2\bar{\gamma}^2 m) \tag{6}$$

$$\varepsilon_\tau = (1-2\tau)\varepsilon + \tau, \quad \varepsilon_0 = \frac{\varepsilon_\tau - \tau}{1-2\tau} \tag{7}$$

$$\forall h \in H,\ \hat{\varepsilon}_\tau(\hat{h}) \leq \hat{\varepsilon}_\tau(h), \quad \text{in particular for } h^* \tag{8}$$

Here is the derivation:

$$\varepsilon_0(\hat{h}) = \frac{\varepsilon_\tau(\hat{h}) - \tau}{1-2\tau} \tag{9}$$

$$\leq \frac{\hat{\varepsilon}_\tau(\hat{h}) + \bar{\gamma} - \tau}{1-2\tau} \quad \text{w.p.}(1-\delta) \tag{10}$$

$$\leq \frac{\hat{\varepsilon}_\tau(h^*) + \bar{\gamma} - \tau}{1-2\tau} \quad \text{w.p.}(1-\delta) \tag{11}$$

$$\leq \frac{\varepsilon_\tau(h^*) + 2\bar{\gamma} - \tau}{1-2\tau} \quad \text{w.p.}(1-\delta) \tag{12}$$

$$= \frac{(1-2\tau)\varepsilon_0(h^*) + \tau + 2\bar{\gamma} - \tau}{1-2\tau} \quad \text{w.p.}(1-\delta) \tag{13}$$

$$= \varepsilon_0(h^*) + \frac{2\bar{\gamma}}{1-2\tau} \quad \text{w.p.}(1-\delta) \tag{14}$$

$$= \varepsilon_0(h^*) + 2\gamma \quad \text{w.p.}(1-\delta) \tag{15}$$

Where we used in the following order: (7)(6)(8)(6)(7), and the last 2 steps are algebraic simplifications, and defining γ as a function of $\bar{\gamma}$. Now we can fill out $\bar{\gamma} = \gamma(1-2\tau)$ into δ of (6), solve for m and we are done.

Note: one could shorten the above derivation and go straight from (9) to (12) by using that result from class.

(c) The closer τ is to 0.5, the more samples are needed to get the same generalization error bound. For τ approaching 0.5, the training data becomes more and more random; having no information at all about the underlying distribution for $\tau = 0.5$.

6. **[19 points] Boosting and high energy physics**

In class, we discussed boosting algorithms and decision stumps. In this problem, we explore applications of these ideas to detect particle emissions in a high-energy particle accelerator. In high energy physics, such as at the Large Hadron Collider (LHC), one accelerates small particles to relativistic speeds and smashes them into one another, tracking the emitted particles. The goal in these problems is to detect the emission of certain interesting particles based on other observed particles and energies.[4] In this problem, we explore the application of boosting to a high energy physics problem, where we use decision stumps applied to 18 low- and high-level physics-based features. All data for the problem is available at `http://cs229.stanford.edu/materials/boost_data.tgz`.

[4]For more, see the following paper: Baldi, Sadowski, Whiteson. Searching for Exotic Particles in High-Energy Physics with Deep Learning. *Nature Communications* 5, Article 4308. `http://arxiv.org/abs/1402.4735`.

For the first part of the problem, we explore how decision stumps based on thresholding can provide a weak-learning guarantee. In particular, we show that for real-valued attributes x, there is an edge $\gamma > 0$ that decision stumps guarantee. Recall that thresholding-based decision stumps are functions indexed by a threshold s and sign $+/-$, such that

$$\phi_{s,+}(x) = \begin{cases} 1 & \text{if } x \geq s \\ -1 & \text{if } x < s \end{cases} \quad \text{and} \quad \phi_{s,-}(x) = \begin{cases} -1 & \text{if } x \geq s \\ 1 & \text{if } x < s. \end{cases}$$

That is, $\phi_{s,+}(x) = -\phi_{s,-}(x)$. We assume for simplicity in the theoretical parts of this exercise that our input attribute vectors $x \in \mathbb{R}$, that is, they are one-dimensional. Now, we would like guarantee that there is some $\gamma > 0$ and a threshold s such that, for *any* distribution p on the training set $\{x^{(i)}, y^{(i)}\}_{i=1}^m$ (where $y^{(i)} \in \{-1, +1\}$ and $x^{(i)} \in \mathbb{R}$, and we recall that p is a distribution on the training set if $\sum_{i=1}^m p_i = 1$ and $p_i \geq 0$ for each i) we have

$$\sum_{i=1}^m p_i \mathbf{1}\left\{y^{(i)} \neq \phi_{s,+}(x^{(i)})\right\} \leq \frac{1}{2} - \gamma \quad \text{or} \quad \sum_{i=1}^m p_i \mathbf{1}\left\{y^{(i)} \neq \phi_{s,-}(x^{(i)})\right\} \leq \frac{1}{2} - \gamma.$$

For simplicity, we assume that all of the $x^{(i)}$ are *distinct*, so that none of them are equal. We also assume (without loss of generality, but this makes the problem notationally simpler) that

$$x^{(1)} > x^{(2)} > \cdots > x^{(m)}.$$

(a) [3 points] Show that for each threshold s, there is some $m_0(s) \in \{0, 1, \ldots, m\}$ such that

$$\sum_{i=1}^m p_i \mathbf{1}\left\{\phi_{s,+}(x^{(i)}) \neq y^{(i)}\right\} = \frac{1}{2} - \frac{1}{2}\left(\sum_{i=1}^{m_0(s)} y^{(i)} p_i - \sum_{i=m_0(s)+1}^{m} y^{(i)} p_i\right)$$

and

$$\sum_{i=1}^m p_i \mathbf{1}\left\{\phi_{s,-}(x^{(i)}) \neq y^{(i)}\right\} = \frac{1}{2} - \frac{1}{2}\left(\sum_{i=m_0(s)+1}^{m} y^{(i)} p_i - \sum_{i=1}^{m_0(s)} y^{(i)} p_i\right)$$

Treat sums over empty sets of indices as zero, so that $\sum_{i=1}^0 a_i = 0$ for any a_i, and similarly $\sum_{i=m+1}^m a_i = 0$.

Answer:

We perform several algebraic steps. Let $\operatorname{sgn}(t) = 1$ if $t \geq 0$, and $\operatorname{sgn}(t) = -1$ otherwise. Then $\mathbf{1}\{\phi_{s,+}(x) \neq y\} = \mathbf{1}\{\operatorname{sgn}(x - s) \neq y\} = \mathbf{1}\{y \operatorname{sgn}(x - s) \leq 0\}$. Thus we have

$$\begin{aligned}\sum_{i=1}^m p_i \mathbf{1}\left\{\phi_{s,+}(x^{(i)}) \neq y^{(i)}\right\} &= \sum_{i=1}^m p_i \mathbf{1}\left\{y^{(i)} \cdot \operatorname{sgn}(x^{(i)} - s) \leq 0\right\} \\ &= \sum_{i:x^{(i)} \geq s} p_i \mathbf{1}\left\{y^{(i)} = -1\right\} + \sum_{i:x^{(i)} < s} p_i \mathbf{1}\left\{y^{(i)} = 1\right\}.\end{aligned}$$

Thus, if we let $m_0(s)$ be the index in $\{1, \ldots, m\}$ such that $x^{(i)} \geq s$ for $i \leq m_0(s)$ and $x^{(i)} < s$ for $i > m_0(s)$, which we know must exist because $x^{(1)} > x^{(2)} > \cdots$, we have

$$\sum_{i=1}^m p_i \mathbf{1}\left\{\phi_{s,+}(x^{(i)}) \neq y^{(i)}\right\} = \sum_{i=1}^{m_0(s)} p_i \mathbf{1}\left\{y^{(i)} = -1\right\} + \sum_{i=m_0(s)+1}^{m} p_i \mathbf{1}\left\{y^{(i)} = 1\right\}.$$

Now we make the key observation: we have

$$\mathbf{1}\{y=-1\} = \frac{1-y}{2} \text{ and } \mathbf{1}\{y=1\} = \frac{1+y}{2},$$

as $y \in \{-1,1\}$. Consquently,

$$\begin{aligned}
\sum_{i=1}^{m} p_i \mathbf{1}\left\{\phi_{s,+}(x^{(i)}) \neq y^{(i)}\right\} &= \sum_{i=1}^{m_0(s)} p_i \mathbf{1}\left\{y^{(i)} = -1\right\} + \sum_{i=m_0(s)+1}^{m} p_i \mathbf{1}\left\{y^{(i)} = 1\right\} \\
&= \sum_{i=1}^{m_0(s)} p_i \frac{1-y^{(i)}}{2} + \sum_{i=m_0(s)+1}^{m} p_i \frac{1+y^{(i)}}{2} \\
&= \frac{1}{2}\sum_{i=1}^{m} p_i - \frac{1}{2}\sum_{i=1}^{m_0(s)} p_i y^{(i)} + \frac{1}{2}\sum_{i=m_0(s)+1}^{m} p_i y^{(i)} \\
&= \frac{1}{2} - \frac{1}{2}\left(\sum_{i=1}^{m_0(s)} p_i y^{(i)} - \sum_{i=m_0(s)+1}^{m} p_i y^{(i)}\right).
\end{aligned}$$

The last equality follows because $\sum_{i=1}^{m} p_i = 1$. The case for $\phi_{s,-}$ is symmetric to this one, so we omit the argument.

(b) [3 points] Define, for each $m_0 \in \{0, 1, \ldots, m\}$,

$$f(m_0) = \sum_{i=1}^{m_0} y^{(i)} p_i - \sum_{i=m_0+1}^{m} y^{(i)} p_i.$$

Show that there exists *some* $\gamma > 0$, which may depend on the training set size m (but should not depend on p), such that for any set of probabilities p on the training set, where $p_i \geq 0$ and $\sum_{i=1}^{m} p_i = 1$, we can find m_0 with

$$|f(m_0)| \geq 2\gamma.$$

What is your γ?

(*Hint:* Consider the difference $f(m_0) - f(m_0+1)$.)

Answer: We have for $m_0 \in \{1, \ldots, m\}$ that

$$f(m_0) - f(m_0-1) = \sum_{i=1}^{m_0} y^{(i)} p_i - \sum_{i=m_0+1}^{m} y^{(i)} p_i - \sum_{i=1}^{m_0-1} y^{(i)} p_i + \sum_{i=m_0}^{m} y^{(i)} p_i = 2y^{(m_0)} p_{m_0}.$$

In particular, we have $|f(m_0) - f(m_0-1)| = 2|y^{(m_0)}| p_{m_0} = 2p_{m_0}$ for all $m_0 \in \{1, \ldots, m\}$. Because $\sum_{i=1}^{m} p_i = 1$, there must be at least 1 index m_0 with $p_{m_0} \geq \frac{1}{m}$. Thus we have for some index m_0 that $|f(m_0) - f(m_0-1)| \geq \frac{2}{m}$, and so it must be the case that at least one of

$$|f(m_0)| \geq \frac{1}{m} \text{ or } |f(m_0-1)| \geq \frac{1}{m}$$

holds. We have $\gamma = \frac{1}{2m}$.

(c) [2 points] Based on your answer to part (6b), what edge can thresholded decision stumps guarantee on any training set $\{x^{(i)}, y^{(i)}\}_{i=1}^m$, where the raw attributes $x^{(i)} \in \mathbb{R}$ are all distinct? Recall that the edge of a weak classifier $\phi : \mathbb{R} \to \{-1, 1\}$ is the constant $\gamma \in [0, \frac{1}{2}]$ such that

$$\sum_{i=1}^m p_i \mathbf{1}\left\{\phi(x^{(i)}) \neq y^{(i)}\right\} \leq \frac{1}{2} - \gamma.$$

Can you give an upper bound on the number of thresholded decision stumps required to achieve zero error on a given training set?

Answer: Based on our answer to the first part of the question, the thresholded decision stumps are guaranteed to have edge at least $\gamma = \frac{1}{2m}$ over random guessing.

Boosting takes $\frac{\log m}{2\gamma^2}$ iterations to achieve zero error, as shown in class, so with decision stumps we will achieve zero error in at most $2m^2 \log m$ iterations of boosting. Each iteration of boosting introduces a single new weak classifier/hypothesis, so at most $2m^2 \log m$ thresholded decision stumps are necessary.

(d) [11 points] Now you will implement boosting on data developed from a physics-based simulation of a high-energy particle accelerator. We provide two datasets, `boosting-train.csv` and `boosting-test.csv`, which consist of training data and test data for a binary classification problem on which you will apply boosting techniques. (For those not using `Matlab`, the files are comma-separated files, the first column of which consists of binary ± 1-labels $y^{(i)}$, the remaining 18 columns are the raw attribtues.) The file `load_data.m`, which we provide, loads the datasets into memory, storing training data and labels in appropriate vectors and matrices, and then performs boosting using *your* implemented code, and plots the results.

i. [5 points] Implement a method that finds the optimal thresholded decision stump for a training set $\{x^{(i)}, y^{(i)}\}_{i=1}^m$ and distribution $p \in \mathbb{R}_+^m$ on the training set. In particular, fill out the code in the method `find_best_threshold.m`. Include your code in your solution.

ii. [2 points] Implement boosted decision stumps by filling out the code in the method `stump_booster.m`. Your code should implement the weight updating at each iteration $t = 1, 2, \ldots$ to find the optimal value θ_t given the feature index and threshold. Include your code in your solution.

iii. [2 points] Implement *random* boosting, where at each step the choice of decision stump is made completely randomly. In particular, at iteration t random boosting chooses a random index $j \in \{1, 2, \ldots, m\}$, then chooses a random threshold s from among the data values $\{x^{(i)}{}_j\}_{i=1}^m$, and then chooses the tth weight θ_t optimally for this (random) classifier $\phi_{s,+}(x) = \text{sign}(x_j - s)$. Implement this by filling out the code in `random_booster.m`.

iv. [2 points] Run the method `load_data.m` with your implemented boosting methods. Include the plots this method displays, which show the training and test error for boosting at each iteration $t = 1, 2, \ldots$. Which method is better?

Answer: Random decision stumps require about 200 iterations to get to error .23 or so, while regular boosting (with greedy decision stumps) requires about 15 iterations to get this error, and attains error rate about .2 after 200 or so iterations. See Fig. 1.

[A few notes: we do not expect boosting to get classification accuracy better than approximately 80% for this problem.]

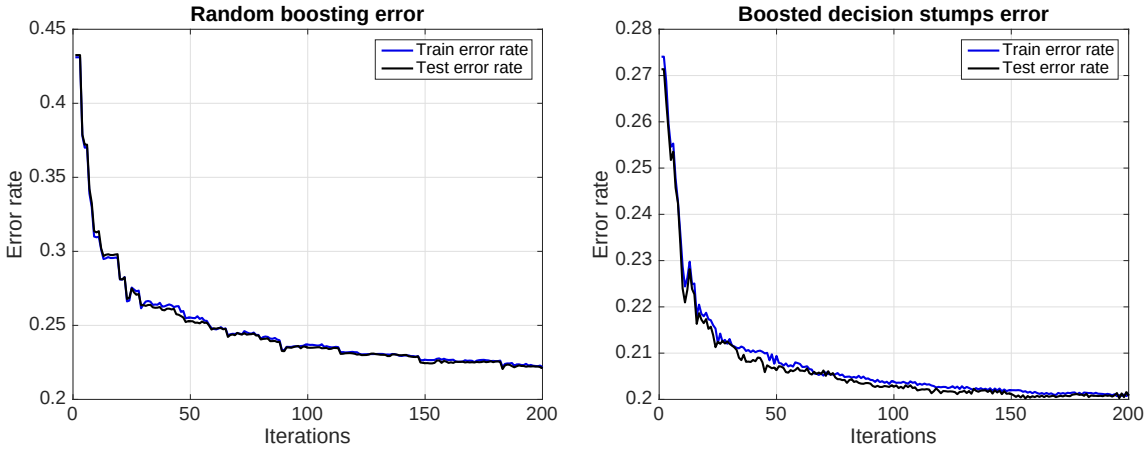

Figure 1: Boosting error for random selection of decision stumps and the greedy selection made by boosting.

Answer: Here is code for each of the three coding parts.

```
function [ind, thresh] = find_best_threshold(X, y, p_dist)
% FIND_BEST_THRESHOLD Finds the best threshold for the given data
%
% [ind, thresh] = find_best_threshold(X, y, p_dist) returns a threshold
%   thresh and index ind that gives the best thresholded classifier for the
%   weights p_dist on the training data. That is, the returned index ind
%   and threshold thresh minimize
%
%    sum_{i = 1}^m p(i) * 1{sign(X(i, ind) - thresh) ~= y(i)}
%
%   OR
%
%    sum_{i = 1}^m p(i) * 1{sign(thresh - X(i, ind)) ~= y(i)}.
%
%   We must check both signed directions, as it is possible that the best
%   decision stump (coordinate threshold classifier) is of the form
%   sign(threshold - x_j) rather than sign(x_j - threshold).
%
%   The data matrix X is of size m-by-n, where m is the training set size
%   and n is the dimension.
%
%   The solution version uses efficient sorting and data structures to perform
%   this calculation in time O(n m log(m)), where the size of the data matrix
%   X is m-by-n.

[mm, nn] = size(X);
best_err = inf;
ind = 1;
```

```
thresh = 0;

for jj = 1:nn
  [x_sort, inds] = sort(X(:, jj), 1, 'descend');
  p_sort = p_dist(inds);
  y_sort = y(inds);
  % We let the thresholds be s_0, s_1, ..., s_{m-1}, where s_k is between
  % x_sort(k-1) and x_sort(k) (so that s_0 > x_sort(1)). Then the empirical
  % error associated with threshold s_k is exactly
  %
  % err(k) = sum_{l = k + 1}^m p_sort(l) * 1(y_sort(l) == 1)
  %          + sum_{l = 1}^k p_sort(l) * 1(y_sort(l) == -1),
  %
  % because this is where the thresholds fall. Then we can sequentially
  % compute
  %
  % err(l) = err(l - 1) - p_sort(l) y_sort(l),
  %
  % where err(0) = p_sort' * (y_sort == 1).
  %
  % The code below actually performs this calculation with indices shifted by
  % one due to Matlab indexing.
  s = x_sort(1) + 1;
  possible_thresholds = x_sort;
  possible_thresholds = (x_sort + circshift(x_sort, 1)) / 2;
  possible_thresholds(1) = x_sort(1) + 1;
  increments = circshift(p_sort .* y_sort, 1);
  increments(1) = 0;
  emp_errs = ones(mm, 1) * (p_sort' * (y_sort == 1));
  emp_errs = emp_errs - cumsum(increments);
  [best_low, thresh_ind] = min(emp_errs);
  [best_high, thresh_high] = max(emp_errs);
  best_high = 1 - best_high;
  best_err_j = min(best_high, best_low);
  if (best_high < best_low)
    thresh_ind = thresh_high;
  end
  if (best_err_j < best_err)
    ind = jj;
    thresh = possible_thresholds(thresh_ind);
    best_err = best_err_j;
  end
end

function [theta, feature_inds, thresholds] = stump_booster(X, y, T)
% STUMP_BOOSTER Uses boosted decision stumps to train a classifier
%
% [theta, feature_inds, thresholds] = stump_booster(X, y, T)
%  performs T rounds of boosted decision stumps to classify the data X,
```

```
%  which is an m-by-n matrix of m training examples in dimension n,
%  to match y.
%
%  The returned parameters are theta, the parameter vector in T dimensions,
%  the feature_inds, which are indices of the features (a T dimensional
%  vector taking values in {1, 2, ..., n}), and thresholds, which are
%  real-valued thresholds. The resulting classifier may be computed on an
%  n-dimensional training example by
%
%   theta' * sign(x(feature_inds) - thresholds).
%
%  The resulting predictions may be computed simultaneously on an
%  n-dimensional dataset, represented as an m-by-n matrix X, by
%
%  sign(X(:, feature_inds) - repmat(thresholds', m, 1)) * theta.
%
%  This is an m-vector of the predicted margins.

[mm, nn] = size(X);
p_dist = ones(mm, 1);
p_dist = p_dist / sum(p_dist);

theta = [];
feature_inds = [];
thresholds = [];

for iter = 1:T
  [ind, thresh] = find_best_threshold(X, y, p_dist);
  Wplus = p_dist' * (sign(X(:, ind) - thresh) == y);
  Wminus = p_dist' * (sign(X(:, ind) - thresh) ~= y);
  theta = [theta; .5 * log(Wplus / Wminus)];
  feature_inds = [feature_inds; ind];
  thresholds = [thresholds; thresh];
  p_dist = exp(-y .* (...
    sign(X(:, feature_inds) - repmat(thresholds', mm, 1)) * theta));
  fprintf(1, 'Iter %d, empirical risk = %1.4f, empirical error = %1.4f\n', ...
          iter, sum(p_dist), sum(p_dist >= 1));
  p_dist = p_dist / sum(p_dist);
end

function [theta, feature_inds, thresholds] = random_booster(X, y, T)
% RANDOM_BOOSTER Uses random thresholds and indices to train a classifier
%
% [theta, feature_inds, thresholds] = random_booster(X, y, T)
%  performs T rounds of boosted decision stumps to classify the data X,
%  which is an m-by-n matrix of m training examples in dimension n.
%
%  The returned parameters are theta, the parameter vector in T dimensions,
%  the feature_inds, which are indices of the features (a T dimensional vector
```

```
%  taking values in {1, 2, ..., n}), and thresholds, which are real-valued
%  thresholds. The resulting classifier may be computed on an n-dimensional
%
%   theta' * sgn(x(feature_inds) - thresholds).

[mm, nn] = size(X);
p_dist = ones(mm, 1);
p_dist = p_dist / sum(p_dist);

theta = [];
feature_inds = [];
thresholds = [];

for iter = 1:T
  ind = ceil(rand * nn);
  thresh = X(ceil(rand * mm), ind) + 1e-8 * randn;
  Wplus = p_dist' * (sign(X(:, ind) - thresh) == y);
  Wminus = p_dist' * (sign(X(:, ind) - thresh) ~= y);
  theta = [theta; .5 * log(Wplus / Wminus)];
  feature_inds = [feature_inds; ind];
  thresholds = [thresholds; thresh];
  p_dist = exp(-y .* (...
    sign(X(:, feature_inds) - repmat(thresholds', mm, 1)) * theta));
  fprintf(1, 'Iter %d, empirical risk = %1.4f, empirical error = %1.4f\n', ...
          iter, sum(p_dist), sum(p_dist >= 1));
  p_dist = p_dist / sum(p_dist);
end

function s = sgn(v)
s = 2 * (v >= 0) - 1;
```