CS 229, Autumn 2017
Problem Set #2: Supervised Learning II

Due Wednesday, Nov 1 at 11:59 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/fall2017/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] Logistic Regression: Training stability**

   In this problem, we will be delving deeper into the workings of logistic regression. The goal of this problem is to help you develop your skills debugging machine learning algorithms (which can be very different from debugging software in general).

   We have provided a implementation of logistic regression at `http://cs229.stanford.edu/ps/ps2/lr_debug.py`, and two labeled datasets `http://cs229.stanford.edu/ps/ps2/data_a.txt`, and `http://cs229.stanford.edu/ps/ps2/data_b.txt` (datasets A and B). Please do not modify the code for the logistic regression training algorithm for this problem. First, run the given logistic regression code to train two different models on A and B.

   (a) [2 points] What is the most notable difference in training the logistic regression model on datasets A and B?

   Answer: The same code converges when trained on A, but does not converge when trained on B.

   (b) [5 points] Investigate why the training procedure behaves unexpectedly on dataset B, but not on A. Provide hard evidence (in the form of math, code, plots, etc.) to corroborate your hypothesis for the misbehavior. Remember, you should address why your explanation does *not* apply to A.

   Hint: The issue is not a numerical rounding or over/underflow error.

   Answer: The reason is dataset B is linearly separable while A is not. Logistic regression without regularization (as implemented in the code) will not converge to a single parameter if the data is perfectly separable (similar to X^TX being singular in linear regression). Since data is perfectly separable, any θ that separates the data can be increased in magnitude to further decrease the loss, without change in the decision boundary. If left to run, $\|\theta\|_2$ will eventually go to infinity.

   (c) [5 points] For each of these possible modifications, state whether or not it would lead to the provided training algorithm converging on datasets such as B. Justify your answers.

      i. Using a different constant learning rate. **Answer:** No.

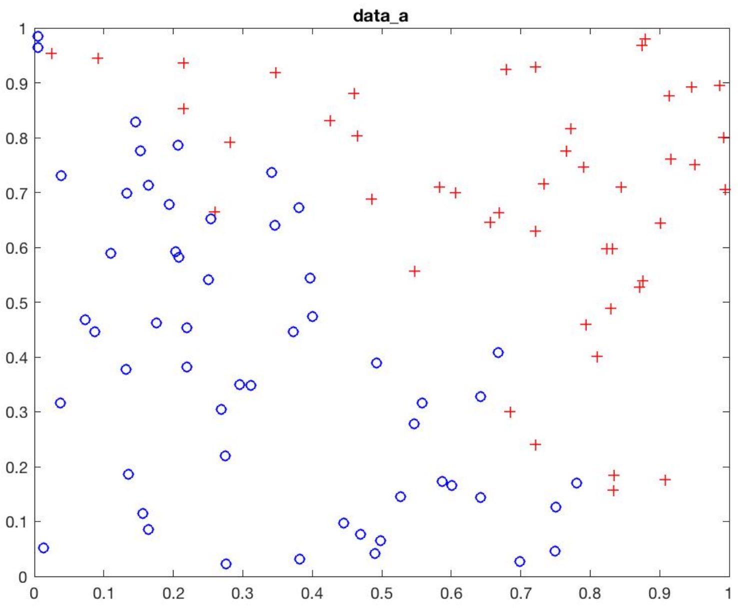

Figure 1: Plot for dataset a

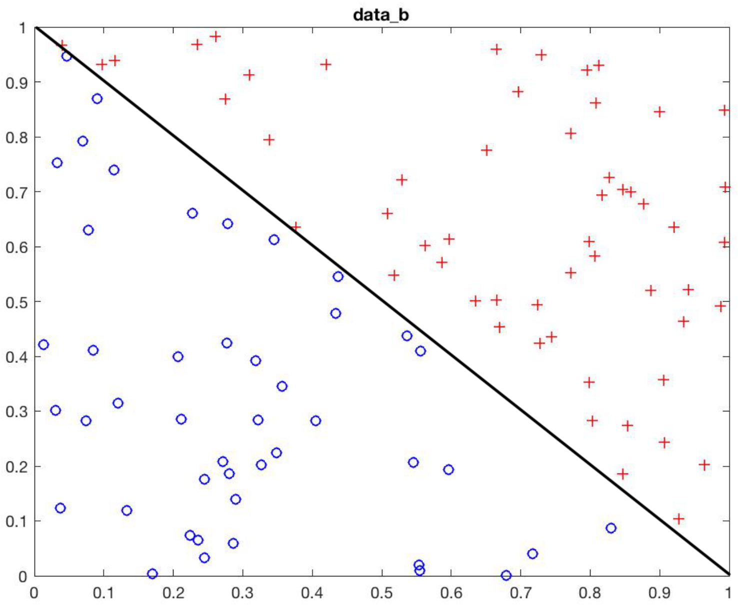

Figure 2: Plot for dataset b

ii. Decreasing the learning rate over time (e.g. scaling the initial learning rate by $1/t^2$, where t is the number of gradient descent iterations thus far). **Answer:** No / May help, eventually updates will become arbitrarily small. Still, the norm of theta can end up being very big. You may also end up underfitting if you aggressively decrease the learning rate.

iii. Adding a regularization term $\|\theta\|_2^2$ to the loss function. **Answer:** Yes. This penalizes large norm of theta and makes the training converge on all kinds of data.

iv. Linear scaling of the input features. **Answer:** No. This will not affect whether the examples are linearly separable.

v. Adding zero-mean Gaussian noise to the training data or labels. **Answer:** No / May help, if the noise make data linearly inseparable.

(d) [3 points] Are support vector machines, which use the hinge loss, vulnerable to datasets like B? Why or why not? Give an informal justification.

Answer: SVMs are NOT vulnerable. They explicitly handle linearly separable data in their objective. By maximizing the margin, their behavior is well defined in both linearly separable datasets and non-separable datasets (with or without regularization).

Hint: Think geometrically (What does minimizing the logistic regression loss do geometrically? What effect does that have on the parameters θ?)

2. **[15 points] Model Calibration**

In this question we will try to understand the output $h_\theta(x)$ of the hypothesis function of a logistic regression model, in particular why we might treat the output as a probability (besides the fact that the sigmoid function ensures $h_\theta(x)$ always lies in the interval $(0, 1)$).

When the probabilities outputted by a model match empirical observation, the model is said to be *well-calibrated* (or reliable). For example, if we consider a set of examples $x^{(i)}$ for which $h_\theta(x^{(i)}) \approx 0.7$, around 70% of those examples should have positive labels. In a well-calibrated model, this property will hold true at every probability value.

Logistic regression tends to output well-calibrated probabilities (this is often not true with other classifiers such as Naive Bayes, or SVMs). We will dig a little deeper in order to understand why this is the case, and find that the structure of the loss function explains this property.

Suppose we have a training set $\{x^{(i)}, y^{(i)}\}_{i=1}^m$ with $x^{(i)} \in \mathbb{R}^{n+1}$ and $y^{(i)} \in \{0, 1\}$. Assume we have an intercept term $x_0^{(i)} = 1$ for all i. Let $\theta \in \mathbb{R}^{n+1}$ be the maximum likelihood parameters learned after training a logistic regression model. In order for the model to be considered well-calibrated, given any range of probabilities (a, b) such that $0 \le a < b \le 1$, and training examples $x^{(i)}$ where the model outputs $h_\theta(x^{(i)})$ fall in the range (a, b), the fraction of positives in that set of examples should be equal to the average of the model outputs for those examples. That is, the following property must hold:

$$\frac{\sum_{i\in I_{a,b}} P\left(y^{(i)} = 1 | x^{(i)}; \theta\right)}{|\{i \in I_{a,b}\}|} = \frac{\sum_{i\in I_{a,b}} \mathbf{1}\{y^{(i)} = 1\}}{|\{i \in I_{a,b}\}|}$$

where $P(y = 1|x; \theta) = h_\theta(x) = 1/(1 + \exp(-\theta^\top x))$, $I_{a,b} = \{i | i \in \{1, ..., m\}, h_\theta(x^{(i)}) \in (a, b)\}$ is an index set of all training examples $x^{(i)}$ where $h_\theta(x^{(i)}) \in (a, b)$, and $|S|$ denotes the size of the set S.

(a) [12 points] Show that the above property holds true for the described logistic regression model over the range $(a, b) = (0, 1)$.
Hint: Use the fact that we include a bias term.
Answer: Consider the log-likelihood of logistic regression model:

$$\ell(\theta) = \sum_{i=1}^{m} y^{(i)} \log\left(h_\theta(x^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - h_\theta(x^{(i)})\right)$$

The condition for maximum-likelihood is that the gradient of $\ell(\theta)$ with respect to θ_j will be 0 for all $j \in \{0, ..., n\}$. i.e,

$$\begin{aligned}
0 &= \frac{\partial \ell(\theta)}{\partial \theta_j} \\
&= \sum_{i=1}^{m} y^{(i)} \frac{1}{h_\theta(x^{(i)})} h_\theta(x^{(i)}) \left(1 - h_\theta(x^{(i)})\right) x_j^{(i)} + (1 - y^{(i)}) \frac{-1}{1 - h_\theta(x^{(i)})} h_\theta(x^{(i)}) \left(1 - h_\theta(x^{(i)})\right) x_j^{(i)} \\
&= \sum_{i=1}^{m} y^{(i)} (1 - h_\theta(x^{(i)})) x_j^{(i)} - (1 - y^{(i)}) h_\theta(x^{(i)}) x_j^{(i)} \\
&= \sum_{i=1}^{m} x_j^{(i)} \left(y^{(i)} - y^{(i)} h_\theta(x^{(i)}) - h_\theta(x^{(i)}) + y^{(i)} h_\theta(x^{(i)})\right) \\
&= \sum_{i=1}^{m} x_j^{(i)} \left(y^{(i)} - h_\theta(x^{(i)})\right)
\end{aligned}$$

This holds specifically for $j = 0$, i.e $x_j = x_0 = 1$, giving us

$$\begin{aligned}
0 &= \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) \\
&= \left(\sum_{i=1}^{m} y^{(i)}\right) - \left(\sum_{i=1}^{m} h_\theta(x^{(i)})\right)
\end{aligned}$$

which gives us as desired

$$\sum_{i=1}^{m} P(y^{(i)} | x^{(i)}; \theta) = \sum_{i=1}^{m} \mathbf{1}\{y^{(i)} = 1\}. \ \blacksquare$$

(b) [3 points] If we have a binary classification model that is perfectly calibrated—that is, the property we just proved holds for any $(a, b) \subset [0, 1]$—does this necessarily imply that the model achieves perfect accuracy? Is the converse necessarily true? Justify your answers.
Answer: Even if a model is perfectly calibrated, this does *not* mean it correctly classify all the training examples. If the dataset has an equal number of positive and negative examples, a model that simply outputs 0.5 probability for all examples is still well-calibrated over $(0, 1)$, as useless as it may be. Likewise, a model that outputs 0.01 for all negative examples and 0.51 for all positive examples is poorly calibrated, but perfectly accurate. Thus accuracy and calibration can sometimes be unrelated.

(c) [2 points] [**Extra Credit**] Discuss what effect including L_2 regularization in the logistic regression objective has on model calibration.

Answer: The standard practice is to not regularize the bias/intercept term. In this case regularization has no effect on calibration. If we regularize the bias term, then the sign of the bias term (after convergence) will decide whether we under-estimate or over-estimate the probabilities (positive bias = under-estimate and vice versa).

Remark: We considered the range $(a, b) = (0, 1)$. This is the only range for which logisitic regression is guaranteed to be calibrated on the training set. When the GLM modeling assumptions hold, all ranges $(a, b) \subset [0, 1]$ are well calibrated. In addition, when the training and test set are from the same distribution and when the model has not overfit or underfit, logistic regression tends to be well-calibrated on unseen test data as well. This makes logistic regression a very popular model in practice, especially when we are interested in the level of uncertainty in the model output.

3. **[15 points] Bayesian Logistic Regression and weight decay**

Consider using a logistic regression model $h_\theta(x) = g(\theta^T x)$ where g is the sigmoid function, and let a training set $\{(x^{(i)}, y^{(i)}); i = 1, \dots, m\}$ be given as usual. The maximum likelihood estimate of the parameters θ is given by

$$\theta_{\text{ML}} = \arg\max_\theta \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}; \theta).$$

If we wanted to regularize logistic regression, then we might put a Bayesian prior on the parameters. Suppose we chose the prior $\theta \sim \mathcal{N}(0, \tau^2 I)$ (here, $\tau > 0$, and I is the $n+1$-by-$n+1$ identity matrix), and then found the MAP estimate of θ as:

$$\theta_{\text{MAP}} = \arg\max_\theta p(\theta) \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}, \theta)$$

Prove that

$$||\theta_{\text{MAP}}||_2 \le ||\theta_{\text{ML}}||_2$$

[Hint: Consider using a proof by contradiction.]

Remark. For this reason, this form of regularization is sometimes also called **weight decay**, since it encourages the weights (meaning parameters) to take on generally smaller values.

Answer: Assume that

$$||\theta_{\text{MAP}}||_2 > ||\theta_{\text{ML}}||_2$$

Then, we have that

$$\begin{aligned}
p(\theta_{\text{MAP}}) &= \frac{1}{(2\pi)^{\frac{n+1}{2}} |\tau^2 I|^{\frac{1}{2}}} e^{-\frac{1}{2\tau^2}(||\theta_{\text{MAP}}||_2)^2} \\
&< \frac{1}{(2\pi)^{\frac{n+1}{2}} |\tau^2 I|^{\frac{1}{2}}} e^{-\frac{1}{2\tau^2}(||\theta_{\text{ML}}||_2)^2} \\
&= p(\theta_{\text{ML}})
\end{aligned}$$

This yields

$$\begin{aligned}
p(\theta_{\mathrm{MAP}})\prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta_{\mathrm{MAP}}) &< p(\theta_{\mathrm{ML}})\prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta_{\mathrm{MAP}}) \\
&\leq p(\theta_{\mathrm{ML}})\prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta_{\mathrm{ML}})
\end{aligned}$$

where the last inequality holds since θ_{ML} was chosen to maximize $\prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta)$. However, this result gives us a contradiction, since θ_{MAP} was chosen to maximize $\prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta)p(\theta)$

4. **[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 square matrix $K \in \mathbb{R}^{m\times m}$ whose entries are given by $K_{ij} = K(x^{(i)},x^{(j)})$ is symmetric and positive semidefinite. You can find more details about Mercer's theorem in the notes, though the description above is sufficient for this problem.

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

(g) [3 points] $K(x,z) = K_3(\phi(x),\phi(z))$

(h) [3 points] **[Extra Credit]** $K(x,z) = p(K_1(x,z))$

[Hint: For part (e), the answer is that K *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.

5. **[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:

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

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 $\theta^{(i)} = \sum_{i \in \mathsf{Misclassified}} \beta_i \phi(x^{(i)})$ 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.

6. **[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/ps/ps2/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`.

   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.

   For MATLAB: 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

[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 .

`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.

For Python: the file `nb.py` provides the `readMatrix` function and starter code.

(a) [15 points] Implement a naive Bayes classifier for spam classification, using the multinomial event model and Laplace smoothing (refer to class notes on Naive Bayes for details on Laplace smoothing).

For MATLAB: 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`) [2].

For Python: You can use the code outline provieded in `nb.py` to train and test your model.

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) [5 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 variable `tokenlist` should be useful for identifying the words/tokens.

(c) [5 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) [3 points] Train an SVM on this dataset using the provided implementations, available for download from `http://cs229.stanford.edu/ps/ps2/`. This implements an SVM using an RBF (Gaussian) kernel. Implementations for both MATLAB and Python are provided.

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.

[2]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.

(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 was:

  i. Training set size 50: Test set error = 2.26% (2.0% python)
  ii. Training set size 100: Test set error = 1.47% (1.38% python)
  iii. Training set size 200: Test set error = .26% (1.63% python)
  iv. Training set size 400: Test set error = .14% (0.25% python)
  v. Training set size 800: Test set error = 0% (0.13% python)
  vi. Training set size 1400: Test set error = 0% (0% python)
  vii. Full training set: Test set error = 0% (0% python)

(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:

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

[spmatrix, tokenlist, trainCategory] = readMatrix('MATRIX.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


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% nb_test.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[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

%---------------

% Compute the error on the test set
error=0;
for i=1:numTestDocs
  if (category(i) ~= output(i))
```

```
    error=error+1;
  end
end

%Print out the classification error on the test set
error/numTestDocs



######################################
## nb.py
######################################

import numpy as np

def readMatrix(file):
    fd = open(file, 'r')
    hdr = fd.readline()
    rows, cols = [int(s) for s in fd.readline().strip().split()]
    tokens = fd.readline().strip().split()
    matrix = np.zeros((rows, cols))
    Y = []
    for i, line in enumerate(fd):
        nums = [int(x) for x in line.strip().split()]
        Y.append(nums[0])
        kv = np.array(nums[1:])
        k = np.cumsum(kv[:-1:2])
        v = kv[1::2]
        matrix[i, k] = v
    return matrix, tokens, np.array(Y)

def nb_train(matrix, category):
    state = {}
    N = matrix.shape[1]
    ###################
    phi = 1. * sum(category) / len(category)
    state['logphi_0'] = np.log(1.-phi)
    state['logphi_1'] = np.log(phi)
    theta_0 = (matrix[category == 0]).sum(axis=0) + 1
    theta_1 = (matrix[category == 1]).sum(axis=0) + 1
    theta_0 /= theta_0.sum()
    theta_1 /= theta_1.sum()
    state['logtheta_0'] = np.log(theta_0)
    state['logtheta_1'] = np.log(theta_1)
    ###################
    return state

def nb_test(matrix, state):
    output = np.zeros(matrix.shape[0])
```

```
        ###################
        logphi_0 = state['logphi_0']
        logphi_1 = state['logphi_1']
        logtheta_0 = state['logtheta_0']
        logtheta_1 = state['logtheta_1']
        logprobs_0 = (matrix * logtheta_0).sum(axis=1) + logphi_0
        logprobs_1 = (matrix * logtheta_1).sum(axis=1) + logphi_1

        output = (logprobs_1 > logprobs_0).astype(int)
        ###################
        return output

    def evaluate(output, label):
        error = (output != label).sum() * 1. / len(output)
        print 'Error: %1.4f' % error

    def main():
        trainMatrix, tokenlist, trainCategory = readMatrix('MATRIX.TRAIN')
        testMatrix, tokenlist, testCategory = readMatrix('MATRIX.TEST')

        state = nb_train(trainMatrix, trainCategory)

        for i in np.argsort(state['logtheta_0'] - state['logtheta_1'])[:5]:
            print tokenlist[i]

        output = nb_test(testMatrix, state)

        evaluate(output, testCategory)
        return

    if __name__ == '__main__':
        main()
```