CS 229, Autumn 2017
Problem Set #0: Linear Algebra and Multivariable Calculus

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/autumn2017/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) This specific homework is *not graded*, but we encourage you to solve each of the problems to brush up on your linear algebra. Some of them may even be useful for subsequent problem sets. It also serves as your introduction to using Gradescope for submissions.

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

1. **[0 points] Gradients and Hessians**

   Recall that a matrix $A \in \mathbb{R}^{n\times n}$ is *symmetric* if $A^T = A$, that is, $A_{ij} = A_{ji}$ for all i, j. Also recall the gradient $\nabla f(x)$ of a function $f : \mathbb{R}^n \to \mathbb{R}$, which is the n-vector of partial derivatives

$$\nabla f(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix} \quad \text{where} \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}.$$

   The hessian $\nabla^2 f(x)$ of a function $f : \mathbb{R}^n \to \mathbb{R}$ is the $n \times n$ symmetric matrix of twice partial derivatives,

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} f(x) & \frac{\partial^2}{\partial x_1 \partial x_2} f(x) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} f(x) \\ \frac{\partial^2}{\partial x_2 \partial x_1} f(x) & \frac{\partial^2}{\partial x_2^2} f(x) & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n} f(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} f(x) & \frac{\partial^2}{\partial x_n \partial x_2} f(x) & \cdots & \frac{\partial^2}{\partial x_n^2} f(x) \end{bmatrix}.$$

   (a) Let $f(x) = \frac{1}{2}x^T Ax + b^T x$, where A is a symmetric matrix and $b \in \mathbb{R}^n$ is a vector. What is $\nabla f(x)$?

   (b) Let $f(x) = g(h(x))$, where $g : \mathbb{R} \to \mathbb{R}$ is differentiable and $h : \mathbb{R}^n \to \mathbb{R}$ is differentiable. What is $\nabla f(x)$?

   (c) Let $f(x) = \frac{1}{2}x^T Ax + b^T x$, where A is symmetric and $b \in \mathbb{R}^n$ is a vector. What is $\nabla^2 f(x)$?

   (d) Let $f(x) = g(a^T x)$, where $g : \mathbb{R} \to \mathbb{R}$ is continuously differentiable and $a \in \mathbb{R}^n$ is a vector. What are $\nabla f(x)$ and $\nabla^2 f(x)$? (*Hint:* your expression for $\nabla^2 f(x)$ may have as few as 11 symbols, including $'$ and parentheses.)

2. **[0 points] Positive definite matrices**

   A matrix $A \in \mathbb{R}^{n\times n}$ is *positive semi-definite* (PSD), denoted $A \succeq 0$, if $A = A^T$ and $x^T Ax \geq 0$ for all $x \in \mathbb{R}^n$. A matrix A is *positive definite*, denoted $A \succ 0$, if $A = A^T$ and $x^T Ax > 0$ for all $x \neq 0$, that is, all non-zero vectors x. The simplest example of a positive definite matrix is the identity I (the diagonal matrix with 1s on the diagonal and 0s elsewhere), which satisfies $x^T I x = \|x\|_2^2 = \sum_{i=1}^n x_i^2$.

(a) Let $z \in \mathbb{R}^n$ be an n-vector. Show that $A = zz^T$ is positive semidefinite.

(b) Let $z \in \mathbb{R}^n$ be a *non-zero* n-vector. Let $A = zz^T$. What is the null-space of A? What is the rank of A?

(c) Let $A \in \mathbb{R}^{n\times n}$ be positive semidefinite and $B \in \mathbb{R}^{m\times n}$ be arbitrary, where $m, n \in \mathbb{N}$. Is BAB^T PSD? If so, prove it. If not, give a counterexample with explicit A, B.

3. **[0 points] Eigenvectors, eigenvalues, and the spectral theorem**

The eigenvalues of an $n \times n$ matrix $A \in \mathbb{R}^{n\times n}$ are the roots of the characteristic polynomial $p_A(\lambda) = \det(\lambda I - A)$, which may (in general) be complex. They are also defined as the the values $\lambda \in \mathbb{C}$ for which there exists a vector $x \in \mathbb{C}^n$ such that $Ax = \lambda x$. We call such a pair (x, λ) an *eigenvector, eigenvalue* pair. In this question, we use the notation $\mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ to denote the diagonal matrix with diagonal entries $\lambda_1, \ldots, \lambda_n$, that is,

$$\mathrm{diag}(\lambda_1, \ldots, \lambda_n) = \begin{bmatrix} \lambda_1 & 0 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ 0 & 0 & \lambda_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_n \end{bmatrix}.$$

(a) Suppose that the matrix $A \in \mathbb{R}^{n\times n}$ is diagonalizable, that is, $A = T\Lambda T^{-1}$ for an invertible matrix $T \in \mathbb{R}^{n\times n}$, where $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ is diagonal. Use the notation $t^{(i)}$ for the columns of T, so that $T = [t^{(1)} \ \cdots \ t^{(n)}]$, where $t^{(i)} \in \mathbb{R}^n$. Show that $At^{(i)} = \lambda_i t^{(i)}$, so that the eigenvalues/eigenvector pairs of A are $(t^{(i)}, \lambda_i)$.

A matrix $U \in \mathbb{R}^{n\times n}$ is orthogonal if $U^T U = I$. The spectral theorem, perhaps one of the most important theorems in linear algebra, states that if $A \in \mathbb{R}^{n\times n}$ is symetric, that is, $A = A^T$, then A is *diagonalizable by a real orthogonal matrix.* That is, there are a diagonal matrix $\Lambda \in \mathbb{R}^{n\times n}$ and orthogonal matrix $U \in \mathbb{R}^{n\times n}$ such that $U^T A U = \Lambda$, or, equivalently,

$$A = U\Lambda U^T.$$

Let $\lambda_i = \lambda_i(A)$ denote the ith eigenvalue of A.

(b) Let A be symmetric. Show that if $U = [u^{(1)} \ \cdots \ u^{(n)}]$ is orthogonal, where $u^{(i)} \in \mathbb{R}^n$ and $A = U\Lambda U^T$, then $u^{(i)}$ is an eigenvector of A and $Au^{(i)} = \lambda_i u^{(i)}$, where $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$.

(c) Show that if A is PSD, then $\lambda_i(A) \geq 0$ for each i.

CS 229, Fall 2017
Problem Set #1: Supervised Learning

Due Wednesday, Oct 18 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 `http://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 copy of your code (with comments) and any figures that you are asked to plot. If typing your solutions, include your code as text in your PDF. Do not submit extra files. (5) To account for late days, the due date listed on Gradescope is Oct 21 at 11:59 pm. If you submit after Oct 18, you will begin consuming your late days. If you wish to submit on time, submit before Oct 18 at 11:59 pm.

All students must submit an electronic PDF version. We highly recommend typesetting your solutions via latex. If you are scanning your document by cell phone, please check the Piazza forum for recommended scanning apps and best practices.

1. **[25 points] Logistic regression**

(a) [10 points] Consider the average empirical loss (the risk) for logistic regression:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \log(1 + e^{-y^{(i)}\theta^T x^{(i)}}) = -\frac{1}{m}\sum_{i=1}^{m} \log(h_\theta(y^{(i)}x^{(i)}))$$

where $y^{(i)} \in \{-1, 1\}$, $h_\theta(x) = g(\theta^T x)$ and $g(z) = 1/(1 + e^{-z})$. Find the Hessian H of this function, and show that for any vector z, it holds true that

$$z^T H z \geq 0.$$

Hint: Be careful that the range for label values, $\{-1, 1\}$, is different than the range used in lecture notes, which is $\{0, 1\}$. Please read the supplementary notes if you are having trouble. You might want to start by showing the fact that $\sum_i \sum_j z_i x_i x_j z_j = (x^T z)^2 \geq 0$.
Remark: This is one of the standard ways of showing that the matrix H is positive semi-definite, written "$H \succeq 0$." This implies that J is convex, and has no local minima other than the global one.[1] If you have some other way of showing $H \succeq 0$, you're also welcome to use your method instead of the one above.

(b) [10 points] We have provided two data files:

- `http://cs229.stanford.edu/ps/ps1/logistic_x.txt`
- `http://cs229.stanford.edu/ps/ps1/logistic_y.txt`

These files contain the inputs ($x^{(i)} \in \mathbb{R}^2$) and outputs ($y^{(i)} \in \{-1, 1\}$), respectively for a binary classification problem, with one training example per row. Implement[2] Newton's method for optimizing $J(\theta)$, and apply it to fit a logistic regression model to the data. Initialize Newton's method with $\theta = \vec{0}$ (the vector of all zeros). What are the coefficients θ resulting from your fit? (Remember to include the intercept term.)

[1] If you haven't seen this result before, please feel encouraged to ask us about it during office hours.
[2] Write your own version, and do not call a built-in library function.

(c) [5 points] Plot the training data (your axes should be x_1 and x_2, corresponding to the two coordinates of the inputs, and you should use a different symbol for each point plotted to indicate whether that example had label 1 or -1). Also plot on the same figure the decision boundary fit by logistic regression. (This should be a straight line showing the boundary separating the region where $h_\theta(x) > 0.5$ from where $h_\theta(x) \leq 0.5$.)

2. **[15 points] Poisson regression and the exponential family**

(a) [5 points] Consider the Poisson distribution parameterized by λ:

$$p(y;\lambda) = \frac{e^{-\lambda}\lambda^y}{y!}.$$

Show that the Poisson distribution is in the exponential family, and clearly state what are $b(y)$, η, $T(y)$, and $a(\eta)$.

(b) [3 points] Consider performing regression using a GLM model with a Poisson response variable. What is the canonical response function for the family? (You may use the fact that a Poisson random variable with parameter λ has mean λ.)

(c) [7 points] For a training set $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$, let the log-likelihood of an example be $\log p(y^{(i)}|x^{(i)};\theta)$. By taking the derivative of the log-likelihood with respect to θ_j, derive the stochastic gradient ascent rule for learning using a GLM model with Poisson responses y and the canonical response function.

(d) **[3 extra credit points]** Consider using GLM with a response variable from any member of the exponential family in which $T(y) = y$, and the canonical response function $h(x)$ for the family. Show that stochastic gradient ascent on the log-likelihood $\log p(\vec{y}|X;\theta)$ results in the update rule $\theta_i := \theta_i - \alpha(h(x) - y)x_i$.

3. **[15 points] Gaussian discriminant analysis**

Suppose we are given a dataset $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$ consisting of m independent examples, where $x^{(i)} \in \mathbb{R}^n$ are n-dimensional vectors, and $y^{(i)} \in \{-1, 1\}$. We will model the joint distribution of (x, y) according to:

$$\begin{aligned}
p(y) &= \begin{cases} \phi & \text{if } y = 1 \\ 1-\phi & \text{if } y = -1 \end{cases} \\
p(x|y=-1) &= \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_{-1})^T\Sigma^{-1}(x-\mu_{-1})\right) \\
p(x|y=1) &= \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_{1})^T\Sigma^{-1}(x-\mu_{1})\right)
\end{aligned}$$

Here, the parameters of our model are ϕ, Σ, μ_{-1} and μ_1. (Note that while there're two different mean vectors μ_{-1} and μ_1, there's only one covariance matrix Σ.)

(a) [5 points] Suppose we have already fit ϕ, Σ, μ_{-1} and μ_1, and now want to make a prediction at some new query point x. Show that the posterior distribution of the label at x takes the form of a logistic function, and can be written

$$p(y \mid x; \phi, \Sigma, \mu_{-1}, \mu_1) = \frac{1}{1 + \exp(-y(\theta^T x + \theta_0))},$$

where $\theta \in \mathbb{R}^n$ and the bias term $\theta_0 \in \mathbb{R}$ are some appropriate functions of $\phi, \Sigma, \mu_{-1}, \mu_1$. (Note: the term θ_0 corresponds to introducing an extra coordinate $x_0^{(i)} = 1$, as we did in class.)

(b) [10 points] For this part of the problem only, you may assume n (the dimension of x) is 1, so that $\Sigma = [\sigma^2]$ is just a real number, and likewise the determinant of Σ is given by $|\Sigma| = \sigma^2$. Given the dataset, we claim that the maximum likelihood estimates of the parameters are given by

$$\begin{aligned}
\phi &= \frac{1}{m}\sum_{i=1}^{m} 1\{y^{(i)} = 1\} \\
\mu_{-1} &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = -1\}x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = -1\}} \\
\mu_1 &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} \\
\Sigma &= \frac{1}{m}\sum_{i=1}^{m} (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T
\end{aligned}$$

The log-likelihood of the data is

$$\begin{aligned}
\ell(\phi, \mu_{-1}, \mu_1, \Sigma) &= \log \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}; \phi, \mu_{-1}, \mu_1, \Sigma) \\
&= \log \prod_{i=1}^{m} p(x^{(i)}|y^{(i)}; \mu_{-1}, \mu_1, \Sigma)p(y^{(i)}; \phi).
\end{aligned}$$

By maximizing ℓ with respect to the four parameters, prove that the maximum likelihood estimates of ϕ, μ_{-1}, μ_1, and Σ are indeed as given in the formulas above. (You may assume that there is at least one positive and one negative example, so that the denominators in the definitions of μ_{-1} and μ_1 above are non-zero.)

(c) [**3 extra credit points**] Without assuming that $n = 1$, show that the maximum likelihood estimates of ϕ, μ_{-1}, μ_1, and Σ are as given in the formulas in part (b). [Note: If you're fairly sure that you have the answer to this part right, you don't have to do part (b), since that's just a special case.]

4. **[10 points] Linear invariance of optimization algorithms**

Consider using an iterative optimization algorithm (such as Newton's method, or gradient descent) to minimize some continuously differentiable function $f(x)$. Suppose we initialize the algorithm at $x^{(0)} = \vec{0}$. When the algorithm is run, it will produce a value of $x \in \mathbb{R}^n$ for each iteration: $x^{(1)}, x^{(2)}, \ldots$.

Now, let some non-singular square matrix $A \in \mathbb{R}^{n\times n}$ be given, and define a new function $g(z) = f(Az)$. Consider using the same iterative optimization algorithm to optimize g (with initialization $z^{(0)} = \vec{0}$). If the values $z^{(1)}, z^{(2)}, \ldots$ produced by this method necessarily satisfy $z^{(i)} = A^{-1}x^{(i)}$ for all i, we say this optimization algorithm is **invariant to linear reparameterizations**.

(a) [7 points] Show that Newton's method (applied to find the minimum of a function) is invariant to linear reparameterizations. Note that since $z^{(0)} = \vec{0} = A^{-1}x^{(0)}$, it is sufficient to show that if Newton's method applied to $f(x)$ updates $x^{(i)}$ to $x^{(i+1)}$, then Newton's method applied to $g(z)$ will update $z^{(i)} = A^{-1}x^{(i)}$ to $z^{(i+1)} = A^{-1}x^{(i+1)}$.[3]

(b) [3 points] Is gradient descent invariant to linear reparameterizations? Justify your answer.

5. **[35 points] Regression for denoising quasar spectra**[4]

Introduction. In this problem, we will apply a supervised learning technique to estimate the light spectrum of *quasars*. Quasars are luminous distant galactic nuclei that are so bright, their light overwhelms that of stars in their galaxies. Understanding properties of the spectrum of light emitted by a quasar is useful for a number of tasks: first, a number of quasar properties can be estimated from the spectra, and second, properties of the regions of the universe through which the light passes can also be evaluated (for example, we can estimate the density of neutral and ionized particles in the universe, which helps cosmologists understand the evolution and fundamental laws governing its structure). The *light spectrum* is a curve that relates the light's intensity (formally, lumens per square meter), or *luminous flux*, to its wavelength. Figure 1 shows an example of a quasar light spectrum, where the wavelengths are measured in Angstroms (Å), where 1Å= 10^{-10} meters.

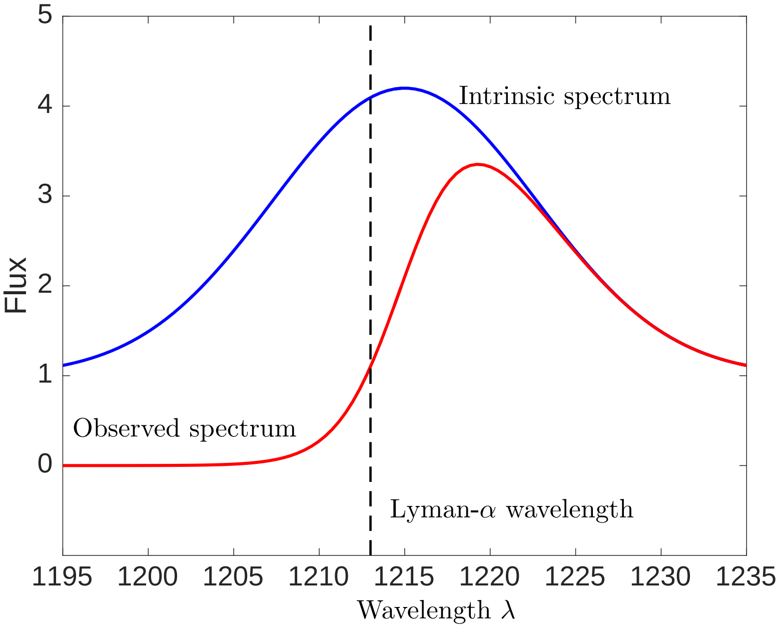


Figure 1: Light spectrum of a quasar. The blue line shows the intrinsic (i.e. original) flux spectrum emitted by the quasar. The red line denotes the observed spectrum here on Earth. To the left of the Lyman-α line, the observed flux is damped and the intrinsic (unabsorbed) flux continuum is not clearly recognizable (red line). To the right of the Lyman-α line, the observed flux approximates the intrinsic spectrum.

The Lyman-α wavelength is a wavelength beyond which intervening particles at most negligibly interfere with light emitted from the quasar. (Interference generally occurs when a photon is

[3] Note that for this problem, you must explicitly prove any matrix calculus identities that you wish to use that are not given in the lecture notes.

[4] Ciollaro, Mattia, et al. "Functional regression for quasar spectra." arXiv:1404.3168 (2014).

absorbed by a neutral hydrogen atom, which only occurs for certain wavelengths of light.) For wavelengths greater than this Lyman-α wavelength, the observed light spectrum f_{obs} can be modeled as a smooth spectrum f plus noise:

$$f_{\text{obs}}(\lambda) = f(\lambda) + \text{noise}(\lambda)$$

For wavelengths below the Lyman-α wavelength, a region of the spectrum known as the Lyman-α forest, intervening matter causes attenuation of the observed signal. As light emitted by the quasar travels through regions of the universe richer in neutral hydrogen, some of it is absorbed, which we model as

$$f_{\text{obs}}(\lambda) = \text{absorption}(\lambda) \cdot f(\lambda) + \text{noise}(\lambda)$$

Astrophysicists and cosmologists wish to understand the absorption function, which gives information about the Lyman-α forest, and hence the distribution of neutral hydrogen in otherwise unreachable regions of the universe. This gives clues toward the formation and evolution of the universe. Thus, it is our goal to estimate the spectrum f of an observed quasar.

Getting the data. We will be using data generated from the Hubble Space Telescope Faint Object Spectrograph (HST-FOS), Spectra of Active Galactic Nuclei and Quasars.[5] We have provided two comma-separated data files located at:

- Training set: `http://cs229.stanford.edu/ps/ps1/quasar_train.csv`
- Test set: `http://cs229.stanford.edu/ps/ps1/quasar_test.csv`

Each file contains a single header row containing 450 numbers corresponding integral wavelengths in the interval $[1150, 1600]$ Å. The remaining lines contain relative flux measurements for each wavelength. Specifically, `quasar_train.csv` contains 200 examples and `quasar_test.csv` contains 50 examples. You may use the helper file `load_quasar_data.m` to load the data in Matlab: `http://cs229.stanford.edu/ps/ps1/load_quasar_data.m`

(a) [10 points] Locally weighted linear regression

Consider a linear regression problem in which we want to "weight" different training examples differently. Specifically, suppose we want to minimize

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m} w^{(i)} \left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

In class, we worked out what happens for the case where all the weights (the $w^{(i)}$'s) are the same. In this problem, we will generalize some of those ideas to the weighted setting.

i. [2 points] Show that $J(\theta)$ can also be written

$$J(\theta) = (X\theta - \vec{y})^T W (X\theta - \vec{y})$$

for an appropriate diagonal matrix W, and where X and $\vec{y}$ are as defined in class. State clearly what W is.

ii. [4 points] If all the $w^{(i)}$'s equal 1, then we saw in class that the normal equation is

$$X^T X\theta = X^T \vec{y},$$

[5] `https://hea-www.harvard.edu/FOSAGN/`

and that the value of θ that minimizes $J(\theta)$ is given by $(X^TX)^{-1}X^T\vec{y}$. By finding the derivative $\nabla_\theta J(\theta)$ and setting that to zero, generalize the normal equation to this weighted setting, and give the new value of θ that minimizes $J(\theta)$ in closed form as a function of X, W and $\vec{y}$.

iii. [4 points] Suppose we have a training set $\{(x^{(i)}, y^{(i)}); i = 1\ldots, m\}$ of m independent examples, but in which the $y^{(i)}$'s were observed with differing variances. Specifically, suppose that

$$p(y^{(i)}|x^{(i)};\theta) = \frac{1}{\sqrt{2\pi}\sigma^{(i)}} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2(\sigma^{(i)})^2}\right)$$

I.e., $y^{(i)}$ has mean $\theta^T x^{(i)}$ and variance $(\sigma^{(i)})^2$ (where the $\sigma^{(i)}$'s are fixed, known, constants). Show that finding the maximum likelihood estimate of θ reduces to solving a weighted linear regression problem. State clearly what the $w^{(i)}$'s are in terms of the $\sigma^{(i)}$'s.

(b) [6 points] Visualizing the data

i. [2 points] Use the normal equations to implement (unweighted) linear regression ($y = \theta^T x$) on the *first* training example (i.e. first non-header row). On one figure, plot both the raw data and the straight line resulting from your fit. State the optimal θ resulting from the linear regression. Remember the intercept term (your optimal θ should lie in $\mathbb{R}^2$).

ii. [2 points] Implement locally weighted linear regression on the *first* training example. Use the normal equations you derived in part (a)(ii). On a different figure, plot both the raw data and the smooth curve resulting from your fit. When evaluating $h(\cdot)$ at a query point x, use weights

$$w^{(i)} = \exp\left(-\frac{(x - x^{(i)})^2}{2\tau^2}\right),$$

with bandwidth parameter $\tau = 5$.

iii. [2 points] Repeat (b)(ii) four more times with $\tau = 1, 10, 100$ and 1000. Plot the resulting curves. You can submit one plot with all four τ values or submit four separate plots. If you submit one plot, make sure all curves are visible. Additionally, in **2-3 sentences**, comment on what happens to the locally weighted linear regression line as τ varies.

(c) [19 points] Predicting quasar spectra with functional regression

We now go a step beyond what we have covered explicitly in class, and we wish to predict an entire part of a spectrum—a curve—from noisy observed data. We begin by supposing that we observe a random sample of m absorption-free spectra, which is possible for quasars very close (in a sense relative to the size of the universe!) to Earth. For a given spectrum f, define f_{right} to be the spectrum to the right of the Lyman-α line. Let f_{left} be the spectrum within the Lyman-α forest region, that is, for lower wavelengths. To make the results cleaner, we define:

$$f(\lambda) = \begin{cases} f_{\text{left}}(\lambda) & \text{if } \lambda < 1200 \\ f_{\text{right}}(\lambda) & \text{if } \lambda \geq 1300 \end{cases}$$

We will learn a function r (for regression) that maps an observed f_{right} to an unobserved target f_{left} (note that f_{left} and f_{right} don't cover the entire spectrum). This is useful in practice because we observe f_{right} with *only* random noise: there is no systematic absorption, which we cannot observe directly, because hydrogen does not absorb photons with higher wavelengths. By predicting f_{left} from a noisy version of f_{right}, we can estimate the unobservable spectrum of a quasar as well as the absorption function. Imaging systems collect data of the form

$$f_{\text{obs}}(\lambda) = \text{absorption}(\lambda) \cdot f(\lambda) + \text{noise}(\lambda)$$

for $\lambda \in \{\lambda_1, \ldots, \lambda_n\}$, a *finite* number of points λ, because they must quantize the information. That is, even in the quasars-close-to-Earth training data, our observations of f_{left} and f_{right} consist of noisy evaluations of the true spectrum f at multiple wavelengths. In our case, we have $n = 450$ and $\lambda_1 = 1150, \ldots, \lambda_n = 1599$.

We formulate the functional regression task as the goal of learning the function r mapping f_{right} to f_{left}:

$$r(f_{\text{right}})(\lambda) = \mathbb{E}(f_{\text{left}} \mid f_{\text{right}})(\lambda)$$

for λ in the Lyman-α forest.

i. [1 points] First, we must smooth the data in the training dataset to make it more useful for prediction. For each $i = 1, \ldots, m$, define $f^{(i)}(\lambda)$ to be the weighted linear regression estimate the i^{th} spectrum. Use your code from part (b)(ii) above to smooth all spectra in the training set using $\tau = 5$. Do the same for the test set. Apply smoothing to the entire spectrums (including both f_{left} and f_{right}) for both train and test. We will now operate on these smoothed spectra.

ii. [14 points] Using your estimated regression functions $f^{(i)}$ for $i = 1, \ldots, m$, we now wish to estimate the unobserved spectrum f_{left} of a quasar from its (noisy) observed spectrum f_{right}. To do so, we perform a weighted regression of the *locally weighted regressions.* In particular, given a new noisy spectrum observation:

$$f_{\text{obs}}(\lambda) = f(\lambda) + \text{noise}(\lambda) \quad \text{for } \lambda \in \{1300, \ldots, 1599\}.$$

We define a metric d which takes as input, two spectra f_1 and f_2, and outputs a scalar:

$$d(f_1, f_2) = \sum_i \Big(f_1(\lambda_i) - f_2(\lambda_i)\Big)^2.$$

The metric d computes squared distance between the new datapoint and previous datapoints. If f_1 and f_2 are right spectra, then we take the preceding sum only over $\lambda \in \{1300, \ldots, 1599\}$, rather than the entire spectrum.

Based on this distance function, we may define the nonparametric *functional* regression estimator, which is a locally weighted sum of *functions* f_{left} from the training data (this is like locally weighted linear regression, except that instead of predicting $y \in \mathbb{R}$ we predict a function f_{left}). Specifically, let f_{right} denote the right side of a spectrum, which we have smoothed using locally weighted linear regression (as you were told to do in the previous part of the problem). We wish to estimate the associated *left* spectrum f_{left}. Define the function $\ker(t) = \max\{1 - t, 0\}$ and let $\mathsf{neighb}_k(f_{\text{right}})$ denote the k indices $i \in \{1, 2, \ldots, m\}$ of the training set that are closest to f_{right}, that is

$$d(f^{(i)}_{\text{right}}, f_{\text{right}}) < d(f^{(j)}_{\text{right}}, f_{\text{right}}) \quad \text{for all } i \in \mathsf{neighb}_k(f_{\text{right}}),\ j \notin \mathsf{neighb}_k(f_{\text{right}})$$

and $\mathsf{neighb}_k(f_{\text{right}})$ contains exactly k indices. In addition, let

$$h := \max_{i \in \{1, \ldots, m\}} d(f^{(i)}_{\text{right}}, f_{\text{right}}).$$

Then define the estimated function $\widehat{f_{\text{left}}} : \mathbb{R} \to \mathbb{R}$ by

$$\widehat{f_{\text{left}}}(\lambda) = \frac{\sum\limits_{i \in \mathsf{neighb}_k(f_{\text{right}})} \ker(d(f_{\text{right}}^{(i)}, f_{\text{right}})/h) f_{\text{left}}^{(i)}(\lambda)}{\sum_{i \in \mathsf{neighb}_k(f_{\text{right}})} \ker(d(f_{\text{right}}^{(i)}, f_{\text{right}})/h)}. \tag{1}$$

Include f_{right} from training in its own neighborhood. Recall that $f_{\text{right}}^{(i)}$ is the *smoothed* (weighted linear regression) estimate of the ith training spectrum.

Construct the functional regression estimate (1) for each spectrum in the entire training set using $k = 3$ nearest neighbors: for each $j = 1, \ldots, m$, construct the estimator $\widehat{f_{\text{left}}}$ from (1) using $f_{\text{right}} = f_{\text{right}}^{(j)}$. Then compute the error $d(f_{\text{left}}^{(j)}, \widehat{f_{\text{left}}})$ between the true spectrum $f_{\text{left}}^{(j)}$ and your estimated spectrum $\widehat{f_{\text{left}}}$ for each j, and return the average over the training data. What is your average training error?

iii. [4 points] Perform functional regression on the test set using the same procedure as in the previous subquestion. Note: You must use neighbors $f_{\text{right}}^{(i)}, f_{\text{left}}^{(i)}$ from the **training set**, and f_{right} in the test set to predict the corresonding f_{left} in the test set. What is your average test error? For test examples 1 and 6, include a plot with both the entire smooth spectrum and the fitted curve $\widehat{f_{\text{left}}}$ curve on the same graph. You should submit two plots: one for test example 1 and one for test example 6.

Reminder: Please include in your submission a copy of your code and figures for the programming questions.

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?

   (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.

   (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.

   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).

   iii. Adding a regularization term $\|\theta\|_2^2$ to the loss function.

   iv. Linear scaling of the input features.

   v. Adding zero-mean Gaussian noise to the training data or labels.

   (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.

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 $0 \leq a < b \leq 1$, 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.

(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.

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

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 $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$ be given as usual. The maximum likelihood estimate of the parameters θ is given by

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

   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:

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

   Prove that

   $$||\theta_{\mathrm{MAP}}||_2 \leq ||\theta_{\mathrm{ML}}||_2$$

   [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.

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.]

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:

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

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 `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`

[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 .

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.

(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?

[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.

CS 229 Autumn 2017
Problem Set #3: Deep Learning & Unsupervised learning

Due Wednesday, Nov 15 at 11:59 pm on Gradescope.

Notes: (1) These questions require thought, but do not require long answers. Please be concise where 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.

Remember to tag all question parts in Gradescope to avoid docked points. If you are skipping a question, please include it on your PDF/photo, but leave the question blank and tag it appropriately on Gradescope. This includes extra credit problems. If you are scanning your document by cellphone, please see `https://gradescope.com/help#help-center-item-student-scanning` for suggested practices.

1. **[20 points] A Simple Neural Network**

Let $X = \{x^{(1)}, \cdots, x^{(m)}\}$ be a dataset of m samples with 2 features, i.e $x^{(i)} \in \mathbb{R}^2$. The samples are classified into 2 categories with labels $y^{(i)} \in \{0, 1\}$. A scatter plot of the dataset is shown in Figure 1:

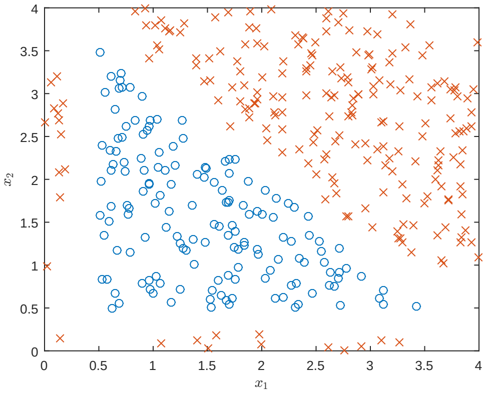


Figure 1: Plot of dataset X.

The examples in class 1 are marked as as "×" and examples in class 0 are marked as "○". We want to perform binary classification using a simple neural network with the architecture shown in Figure 1:

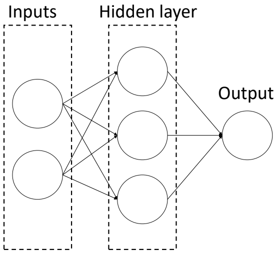


Figure 2: Architecture for our simple neural network.

Denote the two features x_1 and x_2, the three neurons in the hidden layer h_1, h_2, and h_3, and the output neuron as o. Let the weight from x_i to h_j be $w_{i,j}^{[1]}$ for $i \in \{1, 2\}, j \in \{1, 2, 3\}$, and the weight from h_j to o be $w_j^{[2]}$. Finally, denote the intercept weight for h_j as $w_{0,j}^{[1]}$, and the intercept weight for o as $w_0^{[2]}$. For the loss function, we'll use average squared loss instead of the usual negative log-likelihood:

$$l = \frac{1}{m} \sum_{i=1}^{m} (o^{(i)} - y^{(i)})^2,$$

where $o^{(i)}$ is the result of the output neuron for example i.

(a) [5 points] Suppose we use the sigmoid function as the activation function for h_1, h_2, h_3 and o. What is the gradient descent update to $w_{1,2}^{[1]}$, assuming we use a learning rate of α? Your answer should be written in terms of $x^{(i)}$, $o^{(i)}$, $y^{(i)}$, and the weights.

(b) [10 points] Now, suppose instead of using the sigmoid function for the activation function for h_1, h_2, h_3 and o, we instead used the step function $f(x)$, defined as

$$f(x) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases}$$

What is one set of weights that would allow the neural network to classify this dataset with 100% accuracy? Please specify a value for the weights in the order given below and explain your reasoning.

$$\begin{gathered} w_{0,1}^{[1]} =?, w_{1,1}^{[1]} =?, w_{2,1}^{[1]} =? \\ w_{0,2}^{[1]} =?, w_{1,2}^{[1]} =?, w_{2,2}^{[1]} =? \\ w_{0,3}^{[1]} =?, w_{1,3}^{[1]} =?, w_{2,3}^{[1]} =? \\ w_0^{[2]} =?, w_1^{[2]} =?, w_2^{[2]} =?, w_3^{[2]} =? \end{gathered}$$

Hint: There are three sides to a triangle, and there are three neurons in the hidden layer.

(c) [5 points] Let the activation functions for h_1, h_2, h_3 be the linear function $f(x) = x$ and the activation function for o be the same step function as before. Is there a specific set of weights that will make the loss 0? If yes, please explicitly state a value for every weight. If not, please explain your reasoning.

2. **[15 points] EM for MAP estimation**

   The EM algorithm that we talked about in class was for solving a maximum likelihood estimation problem in which we wished to maximize

$$\prod_{i=1}^{m} p(x^{(i)};\theta) = \prod_{i=1}^{m} \sum_{z^{(i)}} p(x^{(i)}, z^{(i)};\theta),$$

   where the $z^{(i)}$'s were latent random variables. Suppose we are working in a Bayesian framework, and wanted to find the MAP estimate of the parameters θ by maximizing

$$\left(\prod_{i=1}^{m} p(x^{(i)}|\theta)\right) p(\theta) = \left(\prod_{i=1}^{m} \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}|\theta)\right) p(\theta).$$

   Here, $p(\theta)$ is our prior on the parameters. Generalize the EM algorithm to work for MAP estimation. You may assume that $\log p(x, z|\theta)$ and $\log p(\theta)$ are both concave in θ, so that the M-step is tractable if it requires only maximizing a linear combination of these quantities. (This roughly corresponds to assuming that MAP estimation is tractable when x, z is fully observed, just like in the frequentist case where we considered examples in which maximum likelihood estimation was easy if x, z was fully observed.)

   Make sure your M-step is tractable, and also prove that $\prod_{i=1}^{m} p(x^{(i)}|\theta)p(\theta)$ (viewed as a function of θ) monotonically increases with each iteration of your algorithm.

3. **[25 points] EM application**

Consider the following problem. There are P papers submitted to a machine learning conference. Each of R reviewers reads each paper, and gives it a score indicating how good he/she thought that paper was. We let $x^{(pr)}$ denote the score that reviewer r gave to paper p. A high score means the reviewer liked the paper, and represents a recommendation from that reviewer that it be accepted for the conference. A low score means the reviewer did not like the paper.

We imagine that each paper has some "intrinsic," true value that we denote by μ_p, where a large value means it's a good paper. Each reviewer is trying to estimate, based on reading the paper, what μ_p is; the score reported $x^{(pr)}$ is then reviewer r's guess of μ_p.

However, some reviewers are just generally inclined to think all papers are good and tend to give all papers high scores; other reviewers may be particularly nasty and tend to give low scores to everything. (Similarly, different reviewers may have different amounts of variance in the way they review papers, making some reviewers more consistent/reliable than others.) We let ν_r denote the "bias" of reviewer r. A reviewer with bias ν_r is one whose scores generally tend to be ν_r higher than they should be.

All sorts of different random factors influence the reviewing process, and hence we will use a model that incorporates several sources of noise. Specifically, we assume that reviewers' scores are generated by a random process given as follows:

$$\begin{aligned} y^{(pr)} &\sim \mathcal{N}(\mu_p, \sigma_p^2), \\ z^{(pr)} &\sim \mathcal{N}(\nu_r, \tau_r^2), \\ x^{(pr)}|y^{(pr)}, z^{(pr)} &\sim \mathcal{N}(y^{(pr)} + z^{(pr)}, \sigma^2). \end{aligned}$$

The variables $y^{(pr)}$ and $z^{(pr)}$ are independent; the variables (x, y, z) for different paper-reviewer pairs are also jointly independent. Also, we only ever observe the $x^{(pr)}$'s; thus, the $y^{(pr)}$'s and $z^{(pr)}$'s are all latent random variables.

We would like to estimate the parameters $\mu_p, \sigma_p^2, \nu_r, \tau_r^2$. If we obtain good estimates of the papers' "intrinsic values" μ_p, these can then be used to make acceptance/rejection decisions for the conference.

We will estimate the parameters by maximizing the marginal likelihood of the data $\{x^{(pr)}; p = 1, \ldots, P, r = 1, \ldots, R\}$. This problem has latent variables $y^{(pr)}$ and $z^{(pr)}$, and the maximum likelihood problem cannot be solved in closed form. So, we will use EM. Your task is to derive the EM update equations. Your final E and M step updates should consist only of addition/subtraction/multiplication/division/log/exp/sqrt of scalars; and addition/subtraction/multiplication/inverse/determinant of matrices. For simplicity, you need to treat only $\{\mu_p, \sigma_p^2; p = 1 \ldots P\}$ and $\{\nu_r, \tau_r^2; r = 1 \ldots R\}$ as parameters. I.e. treat σ^2 (the conditional variance of $x^{(pr)}$ given $y^{(pr)}$ and $z^{(pr)}$) as a fixed, known constant.

(a) In this part, we will derive the E-step:

(i) The joint distribution $p(y^{(pr)}, z^{(pr)}, x^{(pr)})$ has the form of a multivariate Gaussian density. Find its associated mean vector and covariance matrix in terms of the parameters $\mu_p, \sigma_p^2, \nu_r, \tau_r^2$, and σ^2.

[Hint: Recognize that $x^{(pr)}$ can be written as $x^{(pr)} = y^{(pr)} + z^{(pr)} + \epsilon^{(pr)}$, where $\epsilon^{(pr)} \sim \mathcal{N}(0, \sigma^2)$ is independent Gaussian noise.]

(ii) Derive an expression for $Q_{pr}(y^{(pr)}, z^{(pr)}) = p(y^{(pr)}, z^{(pr)}|x^{(pr)})$ (E-step), using the rules for conditioning on subsets of jointly Gaussian random variables (see the notes on Factor Analysis).

(b) Derive the M-step updates to the parameters $\{\mu_p, \nu_r, \sigma_p^2, \tau_r^2\}$. [Hint: It may help to express the lower bound on the likelihood in terms of an expectation with respect to $(y^{(pr)}, z^{(pr)})$ drawn from a distribution with density $Q_{pr}(y^{(pr)}, z^{(pr)})$.]

Remark. In a recent machine learning conference, John Platt (whose SMO algorithm you've seen) implemented a method quite similar to this one to estimate the papers' true scores μ_p. (There, the problem was a bit more complicated because not all reviewers reviewed every paper, but the essential ideas are the same.) Because the model tried to estimate and correct for reviewers' biases ν_r, its estimates of μ_p were significantly more useful for making accept/reject decisions than the reviewers' raw scores for a paper.

4. **[15 points] KL divergence and Maximum Likelihood**

   The Kullback-Leibler (KL) divergence between two discrete-valued distributions $P(X), Q(X)$ is defined as follows:[1]

$$KL(P\|Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$$

   For notational convenience, we assume $P(x) > 0, \forall x$. (Otherwise, one standard thing to do is to adopt the convention that "$0 \log 0 = 0$.") Sometimes, we also write the KL divergence as $KL(P||Q) = KL(P(X)||Q(X))$.

   The KL divergence is an assymmetric measure of the distance between 2 probability distributions. In this problem we will prove some basic properties of KL divergence, and work out a relationship between minimizing KL divergence and the maximum likelihood estimation that we're familiar with.

   (a) [5 points] Nonnegativity. Prove the following:

$$\forall P, Q \quad KL(P\|Q) \geq 0$$

   and

$$KL(P\|Q) = 0 \quad \text{if and only if } P = Q.$$

   [Hint: You may use the following result, called **Jensen's inequality**. If f is a convex function, and X is a random variable, then $E[f(X)] \geq f(E[X])$. Moreover, if f is strictly convex (f is convex if its Hessian satisfies $H \geq 0$; it is *strictly* convex if $H > 0$; for instance $f(x) = -\log x$ is strictly convex), then $E[f(X)] = f(E[X])$ implies that $X = E[X]$ with probability 1; i.e., X is actually a constant.]

   (b) [5 points] **Chain rule for KL divergence.** The KL divergence between 2 conditional distributions $P(X|Y), Q(X|Y)$ is defined as follows:

$$KL(P(X|Y)\|Q(X|Y)) = \sum_y P(y) \left(\sum_x P(x|y) \log \frac{P(x|y)}{Q(x|y)} \right)$$

   This can be thought of as the expected KL divergence between the corresponding conditional distributions on x (that is, between $P(X|Y = y)$ and $Q(X|Y = y)$), where the expectation is taken over the random y.

   Prove the following chain rule for KL divergence:

$$KL(P(X,Y)\|Q(X,Y)) = KL(P(X)\|Q(X)) + KL(P(Y|X)\|Q(Y|X)).$$

   (c) [5 points] **KL and maximum likelihood.**

   Consider a density estimation problem, and suppose we are given a training set $\{x^{(i)}; i = 1, \ldots, m\}$. Let the empirical distribution be $\hat{P}(x) = \frac{1}{m} \sum_{i=1}^{m} 1\{x^{(i)} = x\}$.

[1] If P and Q are densities for continuous-valued random variables, then the sum is replaced by an integral, and everything stated in this problem works fine as well. But for the sake of simplicity, in this problem we'll just work with this form of KL divergence for probability mass functions/discrete-valued distributions.

($\hat{P}$ is just the uniform distribution over the training set; i.e., sampling from the empirical distribution is the same as picking a random example from the training set.)

Suppose we have some family of distributions P_θ parameterized by θ. (If you like, think of $P_\theta(x)$ as an alternative notation for $P(x;\theta)$.) Prove that finding the maximum likelihood estimate for the parameter θ is equivalent to finding P_θ with minimal KL divergence from $\hat{P}$. I.e. prove:

$$\arg\min_\theta KL(\hat{P}\|P_\theta) = \arg\max_\theta \sum_{i=1}^m \log P_\theta(x^{(i)})$$

Remark. Consider the relationship between parts (b-c) and multi-variate Bernoulli Naive Bayes parameter estimation. In the Naive Bayes model we assumed P_θ is of the following form: $P_\theta(x,y) = p(y)\prod_{i=1}^n p(x_i|y)$. By the chain rule for KL divergence, we therefore have:

$$KL(\hat{P}\|P_\theta) = KL(\hat{P}(y)\|p(y)) + \sum_{i=1}^n KL(\hat{P}(x_i|y)\|p(x_i|y)).$$

This shows that finding the maximum likelihood/minimum KL-divergence estimate of the parameters decomposes into $2n+1$ independent optimization problems: One for the class priors $p(y)$, and one for each of the conditional distributions $p(x_i|y)$ for each feature x_i given each of the two possible labels for y. Specifically, finding the maximum likelihood estimates for each of these problems individually results in also maximizing the likelihood of the joint distribution. (If you know what Bayesian networks are, a similar remark applies to parameter estimation for them.)

5. **[20 points] K-means for compression**

   In this problem, we will apply the K-means algorithm to lossy image compression, by reducing the number of colors used in an image.

   We will be using the following files:

   - `http://cs229.stanford.edu/ps/ps3/mandrill-small.tiff`
   - `http://cs229.stanford.edu/ps/ps3/mandrill-large.tiff`

   The `mandrill-large.tiff` file contains a 512x512 image of a mandrill represented in 24-bit color. This means that, for each of the 262144 pixels in the image, there are three 8-bit numbers (each ranging from 0 to 255) that represent the red, green, and blue intensity values for that pixel. The straightforward representation of this image therefore takes about $262144 \times 3 = 786432$ bytes (a byte being 8 bits). To compress the image, we will use K-means to reduce the image to $k = 16$ colors. More specifically, each pixel in the image is considered a point in the three-dimensional (r, g, b)-space. To compress the image, we will cluster these points in color-space into 16 clusters, and replace each pixel with the closest cluster centroid.

   Follow the instructions below. Be warned that some of these operations can take a while (several minutes even on a fast computer)![2]

   (a) MATLAB/Octave: Start up MATLAB/Octave, and type `A = double(imread('mandrill-large.tiff'));` to read in the image. Now, `A` is a "three dimensional matrix," and `A(:,:,1)`, `A(:,:,2)` and `A(:,:,3)` are 512x512 arrays that respectively contain the red, green, and blue values for each pixel. Enter `imshow(uint8(round(A)));` to display the image.

   Python: Start up Python, and type `from matplotlib.image import imread; import matplotlib.pyplot as plt;` and run `A = imread('mandrill-large.tiff')` . Now, `A` is a "three dimensional matrix," and `A[:,:,0]`, `A[:,:,1]` and `A[:,:,2]` are 512x512 arrays that respectively contain the red, green, and blue values for each pixel. Enter `plt.imshow(A); plt.show()` to display the image.

   (b) Since the large image has 262144 pixels and would take a while to cluster, we will instead run vector quantization on a smaller image. Repeat (a) with `mandrill-small.tiff`. Treating each pixel's (r, g, b) values as an element of $\mathbb{R}^3$, run K-means[3] with 16 clusters on the pixel data from this smaller image, iterating (preferably) to convergence, but in no case for less than 30 iterations. For initialization, set each cluster centroid to the (r, g, b)-values of a randomly chosen pixel in the image.

   (c) Take the matrix `A` from `mandrill-large.tiff`, and replace each pixel's (r, g, b) values with the value of the closest cluster centroid. Display the new image, and compare it visually to the original image. Hand in all your code and a copy of your compressed image.

   (d) If we represent the image with these reduced (16) colors, by (approximately) what factor have we compressed the image?

[2] In order to use the `imread` and `imshow` commands in octave, you have to install the Image package from octave-forge. This package and installation instructions are available at: `http://octave.sourceforge.net`

[3] Please implement K-means yourself, rather than using built-in functions.

CS 229, Autumn 2017
Problem Set #4: EM, DL & RL

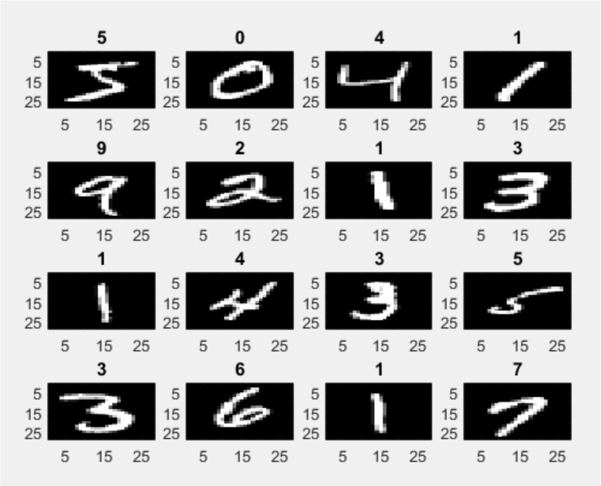
Due Wednesday, Dec 6 at 11:59 pm on Gradescope.

Notes: (1) These questions require thought, but do not require long answers. Please be concise where 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.

Remember to tag all question parts in Gradescope to avoid docked points. If you are skipping a question, please include it on your PDF/photo, but leave the question blank and tag it appropriately on Gradescope. This includes extra credit problems. If you are scanning your document by cellphone, please see `https://gradescope.com/help#help-center-item-student-scanning` for suggested practices.

1. **[25 points] Neural Networks: MNIST image classification**

   In this problem, you will implement a simple neural network to classify grayscale images of handwritten digits (0 - 9) from the MNIST dataset. The dataset contains 60,000 training images and 10,000 testing images of handwritten digits, 0 - 9. Each image is 28×28 pixels in size, and is generally represented as a flat vector of 784 numbers. It also includes labels for each example, a number indicating the actual digit (0 - 9) handwritten in that image. A sample of a few such images along with their labels are shown below.



   You can download the data and starter code at `http://cs229.stanford.edu/ps/ps4/q1`. The starter code splits the set of 60,000 training images and labels into a sets of 50,000 examples as the training set and 10,000 examples for dev set.

   To start, you will implement a neural network with a single hidden layer and cross entropy loss, and train it with the provided data set. Use the sigmoid function as activation for the hidden layer, and softmax function for the output layer. Recall that for a single example (x, y), the cross entropy loss is:

$$CE(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y_k},$$

   where $\hat{y} \in \mathbb{R}^K$ is the vector of softmax outputs from the model for the training example x, and $y \in \mathbb{R}^K$ is the ground-truth vector for the training example x such that $y = [0, ..., 0, 1, 0, ..., 0]^\top$ contains a single 1 at the position of the correct class (also called a "one-hot" representation).

   For m training examples, we average the cross entropy loss over the m examples.

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} CE(y^{(i)}, \hat{y}^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)}.$$

   The starter code already converts labels into one hot representations for you.

   Instead of batch gradient descent or stochastic gradient descent, common practice is to use mini-batch gradient descent for deep learning tasks. In this case, the cost function is defined as follows:

$$J_{MB} = \frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})$$

where B is the batch size, i.e. the number of training example in each mini-batch.

(a) **[15 points]**

Implement both forward-propagation and back-propagation for the above loss function. Initialize the weights of the network by sampling values from a standard normal distribution. Initialize the bias/intercept term to 0. Set the number of hidden units to be 300, and learning rate to be 5. Set $B = 1,000$ (mini batch size). This means that we train with 1,000 examples in each iteration. Therefore, for each epoch, we need 50 iterations to cover the entire training data. The images are pre-shuffled. So you don't need to randomly sample the data, and can just create mini-batches sequentially.

Train the model with mini-batch gradient descent as described above. Run the training for 30 epochs. At the end of each epoch, calculate the value of loss function averaged over the entire training set, and plot it (y-axis) against the number of epochs (x-axis). In the same image, plot the value of the loss function averaged over the dev set, and plot it against the number of epochs.

Similarly, in a new image, plot the accuracy (on y-axis) over the training set, measured as the fraction of correctly classified examples, versus the number of epochs (x-axis). In the same image, also plot the accuracy over the dev set versus number of epochs.

Submit both the plots along with the code.

Also, at the end of 30 epochs, save the learnt parameters (i.e all the weights and biases) into a file, so that next time you can directly initialize the parameters with these values from the file, rather than re-training all over. You do NOT need to submit these parameters.

(b) **[7 points]** Now add a regularization term to your cross entropy loss. The loss function will become

$$J_{MB} = \left(\frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})\right) + \lambda\left(||W^{[1]}||^2 + ||W^{[2]}||^2\right)$$

Be careful not to regularize the bias/intercept term (remember why? from PS2?). Set λ to be 0.0001. Implement the regularized version and plot the same figures as part (a). Be careful NOT to include the regularization term to calculate the loss value (regularization should only be used to calculate the gradients). Submit the two new plots obtained.

Compare the plots obtained from the regularized version with the plots obtained from the non-regularized version, and summarize your observations in a couple of sentences.

As in the previous part, save the learnt parameters (weights and biases) into a different file so that we can initialize from them next time.

(c) **[3 points]** All this while you should have stayed away from the test data completely. Now that you have convinced yourself that the model is working as expected (i.e, the observations you made in the previous part matches what you learnt in class about regularization), it is finally time to measure the model performance on the test set. Once we measure the test set performance, we report it whatever value it may be, and NOT go back and refine the model any further.

Initialize your model from the parameters saved in part (a) (i.e, the non-regularized model), and evaluate the model performance on the test data. Repeat this using the parameters saved in part (b) (i.e, the regularized model).

Report your test accuracy for both regularized model and non-regularized model.

Hint: Be sure to vectorize your code as much as possible! Training can be very slow otherwise.

2. **[15 points] EM Convergence**

We again consider the problem of estimating the parameters of a model in the case of partially observed data. Let $x^{(i)}$ indicate an observed example, and $z^{(i)}$ indicate the corresponding latent variable associated with that observed example. Let θ indicate the parameter of the joint model $p(x, z; \theta)$.

Let us denote the log-marginal density of the observed data as

$$\ell(\theta) = \sum_{i=1}^{m} \log p(x^{(i)}; \theta).$$

We saw in class that the EM algorithm monotonically improves $\ell(\theta)$ by iteratively arriving at better estimates of θ, ensuring that

$$\ell(\theta^{(t+1)}) \geq \ell(\theta^{(t)})$$

where t indicates the iteration number. It does so by constructing a lower bound of $\ell(\theta)$ in each iteration (such that the bound is tight at the current estimate of θ), then maximizing this lower bound with respect to θ, and use the resulting θ as the updated estimate. The algorithm stops when we obtain parameter θ^* such that an M-step from that point results in the same parameter θ^*.

This algorithm will always converge. The reason is that every new θ monotonically increases $\ell(\theta)$, and ℓ is a bounded function. So the algorithm cannot go on for ever.

However, it was not shown that when the algorithm converges, we would have completely maximized $\ell(\theta)$. This is a subtle point. Maybe the EM iterations converged to a value of θ^*, but for some reason that θ^* was not a local maxima of $\ell(\theta)$? Remember, we never directly maximized $\ell(\theta)$ and therefore do not have a trivial guarantee that θ^* is at a local maxima of $\ell(\theta)$!

Show that when the EM algorithm converges, we would have indeed maximized (locally) the log-marginal of the data, by showing that $\nabla_\theta \ell(\theta) = 0$ at the converged value of θ^*.

Hint: Be careful and clear about using θ and θ^* in the right places, especially while taking gradients. Remember, while θ represents a variable, θ^* is a constant.

Hint: The following observation can be helpful:

$$\frac{\nabla_\theta f(\theta) \mid_{\theta=\theta^*}}{f(\theta^*)} = \nabla_\theta [\log f(\theta)]_{\theta=\theta^*}$$

Hint: Start from the fact that when EM converges, the M-step would have reached a fixed point (i.e performing M-step from the converged parameter θ^* will return the same parameter θ^*)

3. **[10 points] PCA**

In class, we showed that PCA finds the "variance maximizing" directions onto which to project the data. In this problem, we find another interpretation of PCA.

Suppose we are given a set of points $\{x^{(1)}, \dots, x^{(m)}\}$. Let us assume that we have as usual preprocessed the data to have zero-mean and unit variance in each coordinate. For a given unit-length vector u, let $f_u(x)$ be the projection of point x onto the direction given by u. I.e., if $\mathcal{V} = \{\alpha u : \alpha \in \mathbb{R}\}$, then

$$f_u(x) = \arg\min_{v \in \mathcal{V}} ||x - v||^2.$$

Show that the unit-length vector u that minimizes the mean squared error between projected points and original points corresponds to the first principal component for the data. I.e., show that

$$\arg \min_{u:u^T u=1} \sum_{i=1}^{m} \|x^{(i)} - f_u(x^{(i)})\|_2^2 \ .$$

gives the first principal component.

Remark. If we are asked to find a k-dimensional subspace onto which to project the data so as to minimize the sum of squares distance between the original data and their projections, then we should choose the k-dimensional subspace spanned by the first k principal components of the data. This problem shows that this result holds for the case of $k = 1$.

4. **[10 points] Independent components analysis**

   For this question you will implement the Bell and Sejnowski ICA algorithm, as covered in class. The files you'll need for this problem are in `http://cs229.stanford.edu/ps/ps4/q4`. The file `mix.dat` contains a matrix with 5 columns, with each column corresponding to one of the mixed signals x_i. The file `bellsej.m` (and `bellsej.py`) contains starter code for your implementation.

   Implement and run ICA, and report what was the W matrix you found. Please make your code clean and very concise, and use symbol conventions as in class. To make sure your code is correct, you should listen to the resulting unmixed sources. (Some overlap in the sources may be present, but the different sources should be pretty clearly separated.)

   Note: In our implementation, we **annealed** the learning rate α (slowly decreased it over time) to speed up learning. We briefly describe in `bellsej.m` (and `bellsej.py`) what we did, but you should feel free to play with things to make it work best for you. In addition to using the variable learning rate to speed up convergence, one thing that we also tried was choosing a random permutation of the training data, and running stochastic gradient ascent visiting the training data in that order (each of the specified learning rates was then used for one full pass through the data); this is something that you could try, too.

5. **[15 points] Markov decision processes**

   Consider an MDP with finite state and action spaces, and discount factor $\gamma < 1$. Let B be the Bellman update operator with V a vector of values for each state. I.e., if $V' = B(V)$, then

   $$V'(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P_{sa}(s')V(s').$$

   (a) **[10 points]** Prove that, for any two finite-valued vectors V_1, V_2, it holds true that

   $$||B(V_1) - B(V_2)||_\infty \leq \gamma ||V_1 - V_2||_\infty.$$

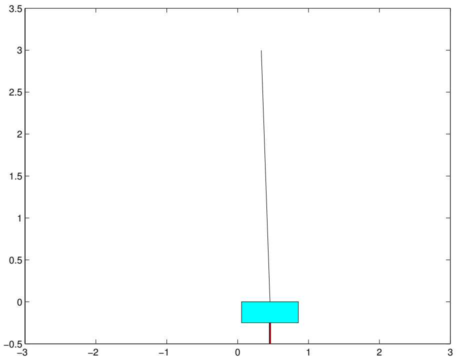
   where

   $$||V||_\infty = \max_{s \in S} |V(s)|.$$

   (This shows that the Bellman update operator is a "γ-contraction in the max-norm.")

   (b) **[5 points]** We say that V is a **fixed point** of B if $B(V) = V$. Using the fact that the Bellman update operator is a γ-contraction in the max-norm, prove that B has at most one fixed point—i.e., that there is at most one solution to the Bellman equations. You may assume that B has at least one fixed point.

   Remark: The result you proved in part(a) implies that value iteration converges geometrically to the optimal value function V^*. That is, after k iterations, the distance between V and V^* is at most γ^k.



6. **[25 points] Reinforcement Learning: The inverted pendulum**

In this problem, you will apply reinforcement learning to automatically design a policy for a difficult control task, without ever using any explicit knowledge of the dynamics of the underlying system.

The problem we will consider is the inverted pendulum or the pole-balancing problem.[1]

Consider the figure shown. A thin pole is connected via a free hinge to a cart, which can move laterally on a smooth table surface. The controller is said to have failed if either the angle of the pole deviates by more than a certain amount from the vertical position (i.e., if the pole falls over), or if the cart's position goes out of bounds (i.e., if it falls off the end of the table). Our objective is to develop a controller to balance the pole with these constraints, by appropriately having the cart accelerate left and right.

We have written a simple Matlab simulator for this problem. The simulation proceeds in discrete time cycles (steps). The state of the cart and pole at any time is completely characterized by 4 parameters: the cart position x, the cart velocity $\dot{x}$, the angle of the pole θ measured as its deviation from the vertical position, and the angular velocity of the pole $\dot{\theta}$. Since it'd be simpler to consider reinforcement learning in a discrete state space, we have approximated the state space by a discretization that maps a state vector $(x, \dot{x}, \theta, \dot{\theta})$ into a number from 1 to `NUM_STATES`. Your learning algorithm will need to deal only with this discretized representation of the states.

At every time step, the controller must choose one of two actions - push (accelerate) the cart right, or push the cart left. (To keep the problem simple, there is no *do-nothing* action.) These are represented as actions 1 and 2 respectively in the code. When the action choice is made, the simulator updates the state parameters according to the underlying dynamics, and provides a new discretized state.

We will assume that the reward $R(s)$ is a function of the current state only. When the pole angle goes beyond a certain limit or when the cart goes too far out, a negative reward is given, and the system is reinitialized randomly. At all other times, the reward is zero. Your program must learn to balance the pole using only the state transitions and rewards observed.

The files for this problem are in `http://cs229.stanford.edu/ps/ps4/q6`. Most of the the code has already been written for you, and you need to make changes only to `control.m` (or `control.py`) in the places specified. This file can be run in Matlab to show a display

[1] The dynamics are adapted from `http://www-anw.cs.umass.edu/rlr/domains.html`

and to plot a learning curve at the end. Read the comments at the top of the file for more details on the working of the simulation.[2]

(a) To solve the inverted pendulum problem, you will estimate a model (i.e., transition probabilities and rewards) for the underlying MDP, solve Bellman's equations for this estimated MDP to obtain a value function, and act greedily with respect to this value function.

Briefly, you will maintain a current model of the MDP and a current estimate of the value function. Initially, each state has estimated reward zero, and the estimated transition probabilities are uniform (equally likely to end up in any other state).

During the simulation, you must choose actions at each time step according to some current policy. As the program goes along taking actions, it will gather observations on transitions and rewards, which it can use to get a better estimate of the MDP model. Since it is inefficient to update the whole estimated MDP after every observation, we will store the state transitions and reward observations each time, and update the model and value function/policy only periodically. Thus, you must maintain counts of the total number of times the transition from state s_i to state s_j using action a has been observed (similarly for the rewards). Note that the rewards at any state are deterministic, but the state transitions are not because of the discretization of the state space (several different but close configurations may map onto the same discretized state).

Each time a failure occurs (such as if the pole falls over), you should re-estimate the transition probabilities and rewards as the average of the observed values (if any). Your program must then use value iteration to solve Bellman's equations on the estimated MDP, to get the value function and new optimal policy for the new model. For value iteration, use a convergence criterion that checks if the maximum absolute change in the value function on an iteration exceeds some specified tolerance.

Finally, assume that the whole learning procedure has converged once several consecutive attempts (defined by the parameter `NO_LEARNING_THRESHOLD`) to solve Bellman's equation all converge in the first iteration. Intuitively, this indicates that the estimated model has stopped changing significantly.

The code outline for this problem is already in `control.m` (and `control.py`), and you need to write code fragments only at the places specified in the file. There are several details (convergence criteria etc.) that are also explained inside the code. Use a discount factor of $\gamma = 0.995$.

Implement the reinforcement learning algorithm as specified, and run it. How many trials (how many times did the pole fall over or the cart fall off) did it take before the algorithm converged?

(b) Plot a learning curve showing the number of time-steps for which the pole was balanced on each trial. Matlab/Octave users just need to execute `plot_learning_curve.m` after `control.m` to get this plot. Python starter code already includes the code to plot.

[2] Note that the routine for drawing the cart does not work in Octave.