CS531 Fall 2014 - Assignment 02
Due Date: Sep. 28, 2014 by 2355 Hrs
Max Marks: 50

September 16, 2014

General Instructions

There are two components of this assignment: i) Algorithm, and ii) Programming. The algorithm component has questions on Linear Regression, while the programming component has two sub-components. Maximum obtainable marks are 50 for this assignment.

Algorithm Component

1 Linear Regression

In this section, we will cover some interesting properties of linear regression.
Review: In the lectures, we have described the least mean square (LMS) solution for linear regression $y = \sum_d w_d x_d$:

$$\mathbf{w}^{LMS} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{1}$$

where $\mathbf{X}$ is our design matrix (N rows, D columns) for the training data $\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^N$ and $\mathbf{y}$ is the N-dimensional column vector of the true values in the training data set. (Note that, for convenience, we have assumed the input has been appended with the constant 1 so we do not have to explicitly introduce w_0.)

To invert the matrix $\mathbf{X}^T\mathbf{X}$, the computational complexity is $O(D^3)$. On the other hand, if D=1, the training data $\mathcal{D}$ would be $\{(x^{(n)}, y^{(n)})\}_{n=1}^N$ (here $x^{(n)}$ is a scalar), and the solution is simply

$$w^{LMS} = \frac{\sum_n x^{(n)} y^{(n)}}{\sum_n \left(x^{(n)}\right)^2} \tag{2}$$

Note that the computation is quite straightforward and no matrix inversion is involved.

1.1 Question 1 (5 points)

Let us consider a special way of doing linear regression for $D > 1$. Instead of using the input $\mathbf{x}_n$ in eq.(1), we are to use each dimension of $\mathbf{x}_n$ separately in eq. (2). Namely,

- For each dimension d, we (imaginarily) construct a new training dataset $\mathcal{D}_d = \{(x_d^{(n)}, y^{(n)})\}_{n=1}^N$.

- With $\mathcal{D}_d$, we use the one-dimensional input to predict $y^{(n)}$. The LMS solution is thus computed according to eq. (2). Let us call the resulting parameter v_d:

$$v_d = \frac{\sum_n x_d^{(n)} y^{(n)}}{\sum_n \left(x_d^{(n)}\right)^2}$$

- We combine all v_d together to form a parameter vector $\mathbf{v}$.

Note that every step above is very simple - we definitely do not have to compute $\mathbf{X}^T\mathbf{X}$ and invert it, as in eq. (1).

However, the question is: is $\mathbf{v}$ going to be the same as $\mathbf{w}^{LMS}$ as computed by eq. (1)?

Unfortunately, the two are *not* the same in the general case. Nonetheless, the two are the same if the columns of $\mathbf{X}$ are orthogonal:

$$\sum_n x_d^{(n)} x_{d'}^{(n)} = 0, \quad \forall \quad d \neq d'$$

This condition can be referred as "the dimensions d and d' are uncorrelated". **Prove** that under this condition, $\mathbf{w}^{LMS}$ and $\mathbf{v}$ are indeed the same.

1.2 Question 2. (15 points)

In real-world data sets, features are rarely uncorrelated. Thus, the "trick" in Question 1 does not apply. However, we can transform the data such that features become uncorrelated. In the following, we will build towards that goal.

1.2.1 (2 points)

Consider univariate linear regression $y = wx$. For a training data set $\mathcal{D} = \{(x^{(n)}, y^{(n)})\}_{n=1}^N$, after computing the LMS solution w^{LMS}, we obtain the residual error

$$e^{(n)} = y^{(n)} - w^{LMS} x^{(n)}$$

Prove the $e^{(n)}$ and $x^{(n)}$ are uncorrelated, i.e., $\sum_n e^{(n)} x^{(n)} = 0$.

1.2.2 (5 points)

Suppose $\mathbf{X} \in \mathbb{R}^{N \times D}$ is our design matrix. Let $\mathbf{u}_d \in \mathbb{R}^N$ denote $\mathbf{X}$'s d-th columns. We will transform $\mathbf{u}_d$ into $\mathbf{z}_d$ such that those dimensions become uncorrelated. We define our transformation in the following recursive way:

- $\mathbf{z}_1 = \mathbf{u}_1$. Namely, the first dimension is unchanged.
- The d-th $\mathbf{z}_d$ is given by

$$\mathbf{z}_d = \mathbf{u}_d - \sum_{d'=1}^{d} \gamma_{dd'} \mathbf{z}'_d$$

  where the coefficient $\gamma_{dd'}$ is the parameter of the following uivariate problem: $\mathbf{z}_{d'}$ is the input and $\mathbf{u}_d$ is the target output.

Let's pause for a second and think about what $\mathbf{z}_d$ is. First, let us say $d = 2$ and we will be looking at $\mathbf{z}_2$, which is given by

$$\mathbf{z}_2 = \mathbf{u}_2 - \gamma_{21}\mathbf{z}_1$$

That is, $\mathbf{z_2}$ is the *residual error* of using $\mathbf{z_1}$ to predict $\mathbf{u_2}$! Thus, by the statement in Question 2.2.1, this error $\mathbf{z_2}$ is uncorrelated with $\mathbf{z_1}$

Then what is $\mathbf{z_3}$? by the definition, we have

$$\mathbf{z_3} = \mathbf{u_3} - \gamma_{31}\mathbf{z_1} - \gamma_{32}\mathbf{z_2}$$

Now, we already know $\mathbf{z_1}$ and $\mathbf{z_2}$ are uncorrelated. Then, by the statement in Question 2.1, we know that we can use $\mathbf{z_1}$ and $\mathbf{z_2}$ to *separately* predict $\mathbf{u_3}$, which gives rise to γ_{31} and γ_{32} (Please check the definition of $\gamma_{dd'}$ above).

In other words, $\mathbf{z_3}$ is the residual error for using $\mathbf{z_1}$ and $\mathbf{z_2}$ to predict $\mathbf{u_3}$. Once we have this observation, we are ready to prove some interesting results. Please prove

- $\mathbf{z_3}$ and $\mathbf{z_1}$ are uncorrelated, so are $\mathbf{z_3}$ and $\mathbf{z_2}$.
- Extending our argument, (show rigorously that) all pairs of $\mathbf{z_d}$ and $\mathbf{z_{d'}}$ are uncorrelated, as long as $1 \leq d \neq d' \leq D$.

1.2.3 (3 points)

We are almost there! Now, let us look at $\mathbf{z}_D$, which is given by

$$\mathbf{z}_D = \mathbf{u}_D - \sum_{d'=1}^{D-1} \gamma_{Dd'}\{\mathbf{z}_{d'}\}$$

We will use $\mathbf{z}_D$ to predict $\mathbf{y}$. This is a univariate linear regression and we can compute the parameter as

$$\beta_D = \frac{\sum_n z_{Dn} y^{(n)}}{\sum_n z_{Dn}^2}$$

where z_{Dn} is the n-th element of $\mathbf{z}_D$.

Prove β_D is the same as w_D^{LMS}, the D-th element of w^{LMS}. In other words, while in Questions 2.2.2, we have been using univariate regressions to transform the data to $\mathbf{z}_d$ and now we use univariate regression to get β_D, we arrive at the same solution as if we have used eq (1).

1.2.4 (5 points)

Note that in Question 2.2.3, we only state how to compute w_D^{LMS} using β_D, corresponding to the D-th dimension x_D. What if we want w_d^{LMS} for other dimensions $d \neq D$?

The strategy is surprisingly simple. Suppose we have 3 dimensions x_1, x_2 and x_3. Following the procedure outline in Question 2.2.3, we will get β_3 (i.e., w_3^{LMS}). To get w_1^{LMS}, we use the same procedure but work in the following sequence: x_2, x_3 and x_1.

That is, we "shuffle"' the dimensions of the data so that the first dimension is now the last dimension. It is not difficult to see that, for this shuffled data, the new β_3 will be the same as w_1^{LMS}, corresponding to the original data. We can then use the same procedure to get w_2^{LMS}.

To summarize, our new method of doing linear regression when $D > 1$ is to use univariate regression many times - in Question 2.2.2 to transform the data, in Question 2.2.3 to get the parameter for the last dimension and then above to get the parameter for every dimension.

Please use the big O-notation to analyze the computational complexity of this strategy. The advantage of using univariate regression is to avoid the matrix inversion in eq. (1). What is the computational complexity of the new method? Is it $O(D^{\alpha})$ with $\alpha < 3$, thus, more efficient than directly computing $\mathbf{w}^{LMS}$? If not, (i.e., for the new method $\alpha \geq 3$), then what is the advantage of this new method?

Programming Component

2 Linear Regression

In this part, you will implement linear regression with multiple variables to predict the prices of houses. Suppose you are selling your house and you want to know what a good market price would be. One way to do this is to rst collect information on recent houses sold and make a model of housing prices. The le *hw2data.txt* contains a training set of housing prices in Portland, Oregon. The rst column is the size of the house (in square feet), the second column is the number of bedrooms, and the third column is the price of the house.

2.1 Feature Normalization

Start by loading and displaying some values from this dataset. By looking at the values, note that house sizes are about 1000 times the number of bedrooms. When features differ by orders of magnitude, rst performing feature scaling can make gradient descent converge much more quickly.

Your task here is to complete the code in *featureNormalize.m* to:

- Subtract the mean value of each feature from the dataset.
- After subtracting the mean, additionally scale (divide) the feature values by their respective 'standard deviations."

The standard deviation is a way of measuring how much variation there is in the range of values of a particular feature (most data points will lie within 2 standard deviations of the mean); this is an alternative to taking the range of values (max-min). In Matlab, you can use the std function to compute the standard deviation. For example, inside *featureNormalize.m*, the quantity $X(:, 1)$ contains all the values of x_1 (house sizes) in the training set, so $std(X(:, 1))$ computes the standard deviation of the house sizes. At the time that *featureNormalize.m* is called, the extra column of 1s corresponding to $x_0 = 1$ has not yet been added to X. You will do this for all the features and your code should work with datasets of all sizes (any number of features / examples). Note that each column of the matrix X corresponds to one feature. You will have to submit the code for feature normalization.

Implementation Note: When normalizing the features, it is important to store the values used for normalization - the mean value and the standard deviation used for the computations. After learning the parameters from the model, we often want to predict the prices of houses we have not seen before. Given a new $\mathbf{x}$ value (living room area and number of bedrooms), we must rst normalize $\mathbf{x}$ using the mean and standard deviation that we had previously computed from the training set.

2.1.1 What to submit for feature normalization

Code. (2 points) Submit your *featureNormalize.m* file.

2.2 Least Mean Squares

You need to implement least mean squares solution for regression problem with multiple features. Since the design matrix may be singular make sure that you use pseudo inverse in your implementation. Complete the code in *lmsSolve.m* to implement your solution.

2.2.1 Experiments with Least Mean Squares method

You need to perform three different experiments on the dataset. For each experiment, use 5 fold cross validation to compare the error rate (the value of the residual sum of squares function) for three experiments. You would want to keep the folds the same for all of the following experiments. Write a separate function, *trainLMS.m* that loads the data from file, appends x_0, performs feature normalization and then report average error rate across 5 folds by calling *lmsSolve.m* on appropriate portion of data to compute optimal parameters for each fold and then computing the RSS value for that fold.

1. Use only one feature, size of the house, to predict the sale price of the house.
2. Use only one feature, number of bedrooms, to predict the sale price of the house.
3. Use both the features to predict the sale price of the house

2.2.2 What to submit for Linear Regression using LMS

Question 3. (6 points)
Report the error rate for each fold and the average across folds for each experiment. Present the information in a table where each row represent one experiment and each column presents error rate. Identify which feature (or the combination) gave you the best performance.

Question 4. (2 points)
Report the error rate and the parameter vector θ for whole dataset using the best feature (or combination).

Code. (4 points)
Submit *lmsSolve.m* and *trainLMS.m*.

2.3 Gradient Descent

You need to implement gradient descent for regression problem. The hypothesis function and the batch gradient descent update rule is the same that we discussed in class. You should complete the code in *computeCostMulti.m* and *gradientDescentMulti.m* to implement the cost function and gradient descent for linear regression with multiple variables. Make sure your code supports any number of features and is **well-vectorized**.
Question 5. (1 point)
Give the vectorized form for computing the value of the cost function $J(\theta)$. This will you to implement cost function and gradient descent in vectorized form. The vectorized version is efficient when youre working with numerical computing tools like Matlab.

2.3.1 Experiments - Selecting learning rates

In this part of the exercise, you will get to try out different learning rates for the dataset and nd a learning rate that converges quickly. You need to write code, *trainGS.m* that loads data into memory, appends x_0, normalizes features and then search for optimal parameters for the training data by calling *gradientDescentMulti.m* for a particular learning rate α. Run gradient descent for each learning rate for **up to** 50 iterations. The gradient descent function should also return the history of $J(\theta)$ values in a vector J. After the last iteration, your script should plots the J values against the number of the iterations.

We have discussed in the class how the graph should look if the gradient descent is working properly and the learning rate is fine. If your graph looks very different, especially if your value of $J(\theta)$ increases or even blows up, adjust your learning rate and try again. I recommend trying values of the learning rate α on a log-scale, at multiplicative steps of about 3 times the previous value (i.e., 0.3, 0.1, 0.03, 0.01 and so on). You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve.

Implementation Note: If your learning rate is too large, $J(\theta)$ can diverge and blow up, resulting in values which are too large for computer calculations. In these situations, Matlab will tend to return NaNs. NaN stands for not a number and is often caused by undened operations that involve $-\infty$ and $+\infty$. Matlab Tip: To compare how different learning learning rates affect convergence, its helpful to plot J for several learning rates on the same gure. In Matlab, this can be done by performing gradient descent multi-ple times with a hold on command between plots. Concretely, if youve tried three different values of alpha (you should probably try more values than this) and stored the costs in J1, J2 and J3, you can use the following commands to plot them on the same gure: plot(1:length(J1), J1, b); hold on; plot(1:length(J2), J2, r); plot(1:length(J3), J3, k); The nal arguments b, r, and k specify different colors for the plots.

2.3.2 What to submit for Gradient Descent

Notice the changes in the convergence curves as the learning rate changes. With a small learning rate, you should nd that gradient descent takes a very long time to converge to the optimal value. Conversely, with a large learning rate, gradient descent might not converge or might even diverge!

Question 6. (6 points)
Report all the learning rates you tried and the corresponding number of iterations it took for the algorithm to converge. Also submit a graph between number of iterations and the value of $J(\theta)$ for at least three of your choice of learning rates. (One graph with multiple curves is preferred!)

Question 7. (3 points)
Using the best learning rate that you found, run the *trainGS.m* script to run gradient descent until convergence to nd the nal values of θ. Report the number of iterations it took to converge and the optimal parameter vector θ. Next, report if the optimal parameter vector θ obtained using gradient descent is the same as the one obtained using LMS (mention the Euclidean distance between the two vectors). Finally, report the finding of your comparison of the error rate using the two methods.

Code. (6 points) Submit your *trainGS.m*, *gradientDescentMulti.m* and *computeCostMulti.m* files.

Submission instruction

Solutions for both algorithm and programming components should be turned in via SLATE. You must submit a single zip archive file. The file should be named kXX-YYYY-HW2.zip (where XX is your batch and YYYY is your roll number) and contain: i) a PDF for the algorithm component and ii) a folder for the programming component. Specific instructions for both components are given below:

Algorithm Component: Up to two people can work on the algorithm component, however, each student must submit the solution and name of the collaborator must be mentioned clearly. You can turn in handwritten answers by scanning them into a PDF file; however, your writing must be legible because I can't grade what I cannot read. So you are better of typing your answers to the questions in algorithm component and generating a PDF file. The PDF should be named kXX-YYYY-HW2-AC.pdf, where XX is your batch and YYYY is your roll number. Note that using Latex may help you with equations. What you will need for that is: A Latex interpreter (such as MikTeX) and a Latex editor (such as TeXnicCenter). For a good tutorial go to http://www.andy-roberts.net/writing/latex

Programming Component: The folder that you submit for programming component should be named in a similar way, i.e. kXX-YYYY-HW2-PC. The folder should contain all your source code files and a single PDF report named kXX-YYYY-HW2-PCReport.pdf

Due date and time: You must complete your submission by 11.55pm on September 28, 2014. To avoid *technical glitches* with SLATE, you can submit your assignment ahead of time. No excuses will be accepted for missing the deadline.

Remember that up to 2 assignments can be turned in late by up to 24 hours past the original deadline.