Machine Learning and Computational Statistics, Spring 2015
Homework 1: Ridge Regression and SGD

Due: Friday, February 6, 2015, at 4pm (Submit via NYU Classes)

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. You may include your code inline or submit it as a separate file. You may either scan hand-written work or, preferably, write your answers using software that typesets mathematics (e.g. LaTeX, LyX, or MathJax via iPython).

1 Introduction

In this homework you will implement ridge regression using gradient descent and stochastic gradient descent. We've provided a lot of support Python code to get you started on the right track. References below to particular functions that you should modify are referring to the support code, which you can download from the website. Take advantage of this being a relatively straightforward assignment (it's not as long as it looks!) by improving your programming skills and/or pursuing independent investigations. For example:

- Study up on numpy's "broadcasting" to see if you can simplify and/or speed up your code: `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`
- Think about how you could make the code more modular so that you could easily try different loss functions and step size methods.
- Experiment with more sophisticated approaches to setting the step sizes for SGD (e.g. try out the recommendations in "Bottou's SGD Tricks" on the website)
- Learn about and implement backtracking line search for gradient descent.
- Investigate what happens to the convergence rate when you intentionally make the feature values have vastly different orders of magnitude. Try a dataset (could be artificial) where $\mathcal{X} \subset \mathbf{R}^2$ so that you can plot the convergence path of GD and SGD.
- Instead of taking 1 data point at a time, as in SGD, try "mini-batch gradient descent" where you only use randomly selected subsets of data points to determine each step. How does this effect convergence speed? Are you getting computational speedup as well by using vectorized code?
- What kind of loss function will give us "quantile regression"?

Include any investigations you do in your submission, and we may award up to 5% extra credit.

I encourage you to develop the habit of asking "what if?" questions and then seeking the answers. I guarantee this will give you a much deeper understanding of the material (and likely better performance on the exam and job interviews, if that's your focus). You're also encouraged to post your interesting questions on Piazza under "questions."

Now have fun, and go learn!

2 Linear Regression

2.1 Feature Normalization

When feature values differ greatly, we can get much slower rates of convergence of gradient-based algorithms. Furthermore, when we start using regularization, features with larger values can have a much greater effect on the final output for the same regularization cost – in effect, features with larger values become more important once we start regularizing. One common approach to feature normalization is to linearly transform (i.e. shift and rescale) each feature so that all feature values are in $[0, 1]$. Each feature gets its own transformation. We then apply the same transformations to each feature on the test set. It's important that the transformation is "learned" on the training set, and then applied to the test set. It is possible that some transformed test set values will lie outside the $[0, 1]$ interval.

Modify function `feature_normalization` to normalize all the features to $[0, 1]$. (Can you use numpy's "broadcasting" here?)

2.2 Gradient Descent Setup

In linear regression, we consider the hypothesis space of linear functions $h_\theta : \mathbf{R}^d \to \mathbf{R}$, where

$$h_\theta(x) = \theta^T x,$$

for $\theta, x \in \mathbf{R}^d$, and we choose θ that minimizes the following "square loss" objective function:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x_i) - y_i\right)^2,$$

where $(x_1, y_1), \ldots, (x_m, y_m) \in \mathbf{R}^d \times \mathbf{R}$ is our training data.

While this formulation of linear regression is very convenient, the more standard linear hypothesis space is:

$$h_{\theta,b}(x) = \theta^T x + b,$$

which allows a "bias" or nonzero intercept term. The standard way to achieve this, while still maintaining the convenience of the first representation, is to add an extra dimension to x that is always a fixed value, such as 1. You should convince yourself that this is equivalent. We'll assume this representation, and thus we'll actually take $\theta, x \in \mathbf{R}^{d+1}$.

1. Let $X \in \mathbf{R}^{m \times d+1}$ be "design matrix", where the i'th row of X is x_i. Let $y = (y_1, \ldots, y_m)^T \in \mathbf{R}^{m \times 1}$ be a the "response". Write the objective function $J(\theta)$ as an matrix/vector expression, without using an explicit summation sign.

$$J(\theta) = \frac{1}{2}(X\theta - y)^T(X\theta - y)$$

2. Write down an expression for the gradient of J.

$$\nabla_\theta J(\theta) = X^T X\theta - X^T y$$

3. In our search for a θ that minimizes J, suppose we take a step from θ to $\theta + \eta\Delta$, where $\Delta \in \mathbf{R}^{d+1}$ is a unit vector giving the direction of the step, and $\eta \in \mathbf{R}$ is the length of the step. Use the gradient to write down an approximate expression for $J(\theta + \eta\Delta) - J(\theta)$.

$$J(\theta + \eta(\frac{\nabla_\theta J(\theta)}{|\nabla_\theta J(\theta)|})) - J(\theta)$$

4. Write down the expression for updating θ in the gradient descent algorithm. Let η be the step size.

$$\theta := \theta - n(\frac{\nabla_\theta J(\theta)}{|\nabla_\theta J(\theta)|})$$

$$\theta := \theta - n(\Delta_\theta J(\theta))$$

5. Modify the function `compute_square_loss`, to compute $J(\theta)$ for a given θ.

6. Modify the function `compute_square_loss_gradient`, to compute $\nabla_\theta J(\theta)$.

2.3 Gradient Checker

For many optimization problems, coding up the gradient correctly can be tricky. Luckily, there is a nice way to numerically check the gradient calculation. If $J : \mathbf{R}^d \to \mathbf{R}$ is differentiable, then for any direction vector $\Delta \in \mathbf{R}^d$, the directional derivative of J at θ in the direction Δ is given by:

$$\lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon\Delta) - J(\theta - \varepsilon\Delta)}{2\epsilon}$$

We can approximate this derivative by choosing a small value of $\varepsilon > 0$ and evaluating the quotient above. We can get an approximation to the gradient by approximating the directional derivatives in each coordinate direction and putting them together into a vector. See `http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimization` for details.

1. Complete the function `grad_checker` according to the documentation given.

2. (Optional) Write a generic version of `grad_checker` that will work for any objective function. It should take as parameters a function that computes the objective function and a function that computes the gradient of the objective function.

2.4 Batch Gradient Descent

At the end of the skeleton code, the data is loaded, split into a training and test set, and normalized. We'll now finish the job of running regression on the training set. Later on we'll plot the results together with SGD results.

1. Complete `batch_gradient_descent`.

2. Starting with a step-size of 0.1 (not a bad one to start with), try various different fixed step sizes to see which converges most quickly. Plot the value of the objective function as a function of the number of steps. Briefly summarize your findings.

   0.1 was a good step size that took about 60 steps to converge. 0.01 was not as good, This one took about 600 steps. This was moving 10 times as slow and took a magnitude of 10 more steps. Intuition shows that 0.1 probably is not taking too large of steps at all, so I tried a larger step size. After moving the step size to 1, I had it converge in roughly 6 steps, which shows that we did not have the issue of moving too far with a step size of 1. I have tried going larger, and have tried a step size of 3, but we never converge, meaning we are taking too large of steps and never hit the min

3. (Optional) Implement backtracking line search (google it), and never have to worry choosing your step size again. How does it compare to the best fixed step-size you found in terms of number of steps? In terms of time? How does the extra time to run backtracking line search at each step compare to the time it takes to compute the gradient? (You can also compare the operation counts.)

2.5 Ridge Regression (i.e. Linear Regression with L_2 regularization)

When we have large number of features compared to instances, regularization can help control overfitting. Ridge regression is linear regression with L_2 regularization. The objective function is

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x_i) - y_i\right)^2 + \lambda\theta^T\theta,$$

where λ is the regularization parameter which controls the degree of regularization. Note that the bias term is being regularized as well. We will address that below.

1. Compute the gradient of $J(\theta)$ and write down the expression for updating θ in the gradient descent algorithm.

$$J(\theta) = \frac{1}{2}(X\theta - y)^T(X\theta - y) + \frac{\lambda}{2}\theta^T\theta$$

$$\nabla_\theta J(\theta) = (X^TX + \lambda I)^{-1}X^TY$$

$$\theta := \theta - n\left(\frac{\nabla_\theta J(\theta)}{|\nabla_\theta J(\theta)|}\right)$$

$$\theta := \theta - n(\Delta_\theta J(\theta))$$

2. Implement `compute_regularized_square_loss_gradient`.

3. Implement `regularized_grad_descent`.

4. For regression problems, we may prefer to leave the bias term unregularized. One approach is to rewrite $J(\theta)$ and re-compute $\nabla_\theta J(\theta)$ in a way that separates out the bias from the other parameter. Another approach that can achieve approximately the same thing is to use a very large number B, rather than 1, for the extra bias dimension. Explain why making B large decreases the effective regularization on the bias term, and how we can make that regularization as weak as we like (though not zero).

   If we use a large B, then the bias term will add a massive amount to our total error. This means that when gradient descent converges while using a large B, the weight for the bias will always be quite small, effectively making it so our bias term never taken into account in the regression. We can make that regularization weak by instead of using $B = 1$, we can make B start as a very small number. We can set $B = .1, B = 0.01, B = 0.001$, etc

5. Start with $B = 1$. Choosing a reasonable step-size, find the θ^*_λ that minimizes $J(\theta)$ for a range of λ and plot both the training loss and the validation loss as a function of λ. (Note that this is just the square loss, not including the regularization term.) You should initially try λ over several orders of magnitude to find an appropriate range (e.g .$\lambda \in \left\{10^{-2}, 10^{-1}, 1, 10, 100\right\}$. You may want to have $\log(\lambda)$ on the x-axis rather than λ. Once you have found the interesting range for λ, repeat the fits with different values for B, and plot the results on the same graph. For this dataset, does regularizing the bias help, hurt, or make no significant difference?

6. Estimate the average time it takes on your computer to compute a single gradient step.

7. What θ would you select for deployment and why?

2.6 Stochastic Gradient Descent

When the training data set is very large, evaluating the gradient of the loss function can take a long time, since it requires looking at each training example to take a single gradient step. In this case, stochastic gradient descent (SGD) can be very effective. In SGD, the gradient of the risk is approximated by a gradient at a single example. The approximation is poor, but it is unbiased. The algorithm sweeps through the whole training set one by one, and performs an update for each training example individually. One pass through the data is called an *epoch.* Note that each epoch of SGD touches as much data as a single step of batch gradient descent. It is also important that as we cycle through the training examples, they are in a random order, though it doesn't seem that one needs to use a different shuffling for each epoch. (You should want to investigate to confirm or refute that last claim.)

1. *W*rite down the update rule for θ in SGD.

2. Implement `stochastic_grad_descent`.

3. Use SGD to find θ^*_λ that minimizes the ridge regression objective for the λ and B that you selected in the previous problem. Try several different fixed step sizes, as well as step sizes that decrease with the step number according to the following schedules: $\eta_t = \frac{1}{t}$ and $\eta_t = \frac{1}{\sqrt{t}}$. Plot the value of the objective function (or the log of the objective function if that is more clear) as a function of epoch (or step number) for each of the approaches to step size. How

do the results compare? (Note: In this case we are investigating the convergence rate of the optimization algorithm, thus we're interested in the value of the objective function, which includes the regularization term.)

4. Estimate the amount of time it takes on your computer for a single epoch of SGD.
5. Comparing SGD and gradient descent, if your goal is to minimze the total number of epochs (for SGD) or steps (for batch gradient descent), which would you choose? If your goal were to minimize the total time, which would you choose?

3 Risk Minimization

Recall the statistical learning framework, which $(X, Y) \sim P_{\mathcal{X} \times \mathcal{Y}}$, the **expected loss** or **"risk"** of a decision function $f : \mathcal{X} \to \mathcal{A}$ is

$$R(f) = \mathbb{E}\ell(f(X), Y),$$

and a **Bayes decision function** $f_* : \mathcal{X} \to \mathcal{A}$ is a function that achieves the *minimal risk* among all possible functions:

$$R(f_*) = \inf_f R(f).$$

Here we consider the regression setting, in which $\mathcal{A} = \mathcal{Y} = \mathbf{R}$.

1. Show that for the square loss $\ell(\hat{y}, y) = \frac{1}{2}(y - \hat{y})^2$, the Bayes decision function is a $f_*(x) = \mathbb{E}[Y \mid X = x]$. [Hint: Consider constructing $f_*(x)$, one x at a time.]
2. (Optional challenge) Show that for the absolute loss $\ell(\hat{y}, y) = |y - \hat{y}|$, the Bayes decision function is a $f_*(x) = \text{median}[Y \mid X = x]$. [Hint: Again, consider one x at time, and you can use the following characterization of a median: m is a median of the distribution for random variable Y if $P(Y \geq m) \geq \frac{1}{2}$ and $P(Y \leq m) \geq \frac{1}{2}$.] Note: This loss function leads to "median regression", There are other loss functions that lead 'quantile regression' for any chosen quantile.

4 Feedback (not graded)

1. Approximately how long did it take to complete this assignment?
2. Did you find the Python programming challenging? The mathematical part?
3. Any other feedback?