CS168, Spring 2016

Mini-Project #3

Due by 11:59 PM on Tuesday, Apr 19

Instructions

- You can work individually or with one partner. If you work in a pair, both partners will receive the same grade.
- If you've written code to solve a certain part of a problem, or if the part explicitly asks you to implement an algorithm, you must also include the code in your pdf submission. See the problem parts below for instructions on where in your writeup to put the code.
- Make sure plots you submit are easy to read at a normal zoom level.
- Detailed submission instruction can be found on the course website (`http://cs168.stanford.edu`) under the "Coursework - Assignment" section. If you work in pairs, only one member should submit all of the relevant files.
- a) Use 12pt or higher font for your writeup. b) Code marked as "Deliverable" gets pasted into the relevant section, rather than into the appendix (though feel free to put it in both). Keep variable names consistent with those used in the problem statement, and with general conventions. No need to include import statements and other scaffolding, if it is clear from context. Also, please use the `verbatim` environment to paste code in LaTeX from now on, rather than the listings package:

```
def example():
print "Your code should be formatted like this."
```

- **Reminder:** No late assignments will be accepted, but we will drop your lowest mini-project grade when calculating your final grade.

Part 1: Gradient Descent

Goal:
In this exercise you will get some experience with different variations of Gradient Descent and how it is used on the Least Squares Regression problem.

Description: Recall that Gradient Descent updates a solution $\mathbf{a}$ at time step j for objective function f, with step size α according to the following rule:

$$\mathbf{a^{(j+1)}} = \mathbf{a^{(j)}} - \alpha \cdot \nabla \mathbf{f}(\mathbf{a^{(j)}}) \qquad \text{and for the } i\text{th coordinate: } \mathbf{a_i^{(j+1)}} = \mathbf{a_i^{(j)}} - \alpha \cdot \frac{\mathbf{d}}{\mathbf{da_i}} \mathbf{f}(\mathbf{a^{(j)}})$$

The idea behind *stochastic gradient descent* is that in many settings, the function f to be minimized is a complicated function, whose gradient is difficult to compute; nevertheless, $f = \sum_i f_i$ may be expressed as a sum over a number of much simpler functions, f_i whose gradients can be easily computed. Given such a setting, in each step of stochastic gradient descent, the update selects one function f_i (where, i is chosen at random, or according to a round-robin ordering), and performs the gradient update corresponding to f_i.

In this exercise, you will apply Gradient Descent and Stochastic Gradient Descent to the problem of least-squares linear regression. Given input data $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^n$ and $y^{(1)}, \ldots, y^{(m)} \in \mathbb{R}$, the goal is to find $\mathbf{a}$ to minimize the sum of the squared errors, namely $f(\mathbf{a}) = \mathbf{0.5} \cdot \sum_{\mathbf{i=1}}^{\mathbf{m}} (\mathbf{a^T x^{(i)}} - \mathbf{y^{(i)}})^{\mathbf{2}}$, which decomposes

into the sum $f(\mathbf{a}) = \sum_{\mathbf{i=1}}^{\mathbf{m}} \mathbf{f_i}(\mathbf{a})$, with $f_i(\mathbf{a}) = \mathbf{0.5} \cdot (\mathbf{a^T x^{(i)}} - \mathbf{y^{(i)}})^\mathbf{2}$ representing the error in the ith data point.
The following python code below will generate the data used in parts (a) and (b). This code selects m data points $\mathbf{x^{(1)}}, \ldots, \mathbf{x^{(m)}}$, each drawn from an n-dimensional Gaussian, and corresponding y values $y^{(1)}, \ldots, y^{(m)}$ with $y^{(i)}$ set to be a linear function of $\mathbf{x^{(i)}}$ with some (independent Gaussian) noise.

```
n = 100 # dimensions of data
m = 1000 # number of data points
X = np.random.normal(0,1, size=(m,n))
a_true = np.random.normal(0,1, size=(n,1))
y = X.dot(a_true) + np.random.normal(0,0.1,size=(m,1))
```

(a) (2 points) In class we learned that least-squares regression actually has the closed form solution $\mathbf{a} = (\mathbf{X^T X})^{-1} \mathbf{X^T y}$. Find the optimal solution $\mathbf{a}$ and report the value of the objective function using $\mathbf{a}$.

Note: This algorithm runs in time $O(mn^2 + n^3)$, which can be slow for large n. Although gradient descent methods will not give this same exact solution, it will give a close approximation in much less time. For the purpose of this assignment, you can use the closed form solution as a good sanity check for the following parts.

(b) (5 points) Use gradient descent to find the $\mathbf{a}$ that minimizes the least squares objective function. Run gradient descent three times using step sizes $0.0001, 0.001, 0.00125$. You should initialize $\mathbf{a}$ to be the zero vector for all three runs (for parts (b) and (d) as well). Plot the objective function value for 20 iterations for all 3 step sizes on the same graph. Comment 3-4 sentences on how the step size can affect the convergence of gradient descent (feel free to experiment with other step sizes). Also report the step size that had the best final objective value and the corresponding objective value.

(c) (6 points) Now run stochastic gradient descent three times using step sizes $0.001, 0.01, 0.02$. Plot the objective function value for 1000 iterations for all 3 step sizes on the same graph. Comment 3-4 sentences on how the step size can affect the convergence of stochastic gradient descent and how it compares to gradient descent. Compare the performance of the two methods. How do the best final objective values compare? How many times does each algorithm need to use each data point? Also report the step size that had the best final objective value and the corresponding objective value.

In the next two parts, we investigate L_2 regularization as a means for avoiding overfitting, and improving predictive accuracy. Given a dataset $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^n$ and $y^{(1)}, \ldots, y^{(m)} \in \mathbb{R}$ as above, the regularized objective function is given by the following expression:

$$\sum_{i=1}^{m} 0.5 \cdot (\mathbf{a^T x^{(i)}} - \mathbf{y^{(i)}})^\mathbf{2} + \lambda ||\mathbf{a}||_\mathbf{2}^\mathbf{2},$$

where $\lambda \in \mathbb{R}$ is the regularization parameter. In the following problem we investigate the effect of varying λ. Use the following python code for parts (c) and (d)

```
train_m = 100
test_m = 1000
n = 100
X_train = np.random.normal(0,1, size=(train_m,n))
a_true = np.random.normal(0,1, size=(n,1))
y_train = X_train.dot(a_true) + 0.5*np.random.normal(0,1,size=(train_m,1))
X_test = np.random.normal(0,1, size=(test_m,n))
y_test = X_test.dot(a_true) + 0.5*np.random.normal(0,1,size=(test_m,1))
```

(d) (2 points) Run standard gradient descent (with a reasonable step size of your choice) on your training data (X_{train}, y_{train}). Report the value of the objective function. Now report the value of the objective function if X_{test} and y_{test} are used instead (make sure to use the same value of a_{true} for both the test and training data). Repeat the above experiment with the size of the training set equal to 20 (i.e. set $train_m = 20$ in the above code).

(e) (10 points) Now run gradient descent with regularization on your training data with λ values of $100, 10, 1, 0.1, 0.01, 0.001$. For each regularization constant, report the objective function value on both training and testing data. Comment 2-3 sentences on how λ affects the objective function value on the testing data.
Note: You might wish to experiment with different training set sizes (i.e. varying $train_m$). Because the data is generated randomly, you should run your experiments several times to get an understanding of the 'typical' performance.

Deliverables: Objective value for part (a). Code, plot, discussion and optimal step size and objective function value for part (b) and (c). Code and objective function values for part (d). Code, objective function values and discussion for part (e).

Part 2: QWOP[1]

Goal: In this exercise you will get some hands-on experience solving a tricky optimization problem that captures some of the difficulties of some real-world robotics problems.

Description: In 2010, the QWOP game[2] took the Internet by storm. The purpose of the game is to get your avatar to run the 100-meter event at the Olympic Games by controlling their thigh and knee angles. If you haven't played the game, give it a try—this assignment will be much more amusing if you have first spent a few minutes with the flash game; even getting your runner to fall forwards rather than backwards can be quite a challenge.

Rather than learn how to play the game, you'll just write a computer program to do it for you! We have implemented the QWOP physics engine in Python and MATLAB. These functions take a list of 40 floating point numbers as input (corresponding to 20 instructions for the angle of the thighs and 20 for the angle of the knees). The output of the function is a single number, representing the final $x-$position of the head of the avatar. The more efficiently you get the avatar to run, the further it will go and the larger the output number. The goal of this problem is for you to find an input that makes the avatar run as far as possible—namely find a length 40 vector that results in the QWOP code evaluating to as large a number as possible.

For your amusement, the MATLAB and Python implementations also provide an animation of the movements of the avatar on any given input.

(a) *[do not hand in]* Make sure you can call either the python `sim(plan)` function, or the MATLAB `qwop(plan)` functions with a `plan` consisting of 40 floating point numbers in the range $[-1, 1]$. The MATLAB and Python implementations come with a visualization of the game, and we highly encourage you to look at what actually happens. (In the MATLAB implementation, you can suppress the animation by including an additional argument of "0" in the function call, ie `qwop(plan, 0)`.)

(b) (20 points) Optimize QWOP. Let d be the distance you get your avatar to go. The number of points you receive will be $\frac{d^2}{5}$ with anything above 20 points regarded as bonus..

Note: this is a non-convex optimization problem that likely has many local optima. Feel free to be creative in adapting gradient-descent inspired approaches that might be robust to such phenomena. Please do NOT use optimization code that you found online.

(c) (5 points) Describe which techniques (or hacks) you tried, and whether or not they were successful. Also report the best distance you get.

Deliverables: For part (b), the code, input and a link to a YouTube video of the run (the Python code gives the option to save to an mp4 video; you may need to install ffmpeg for the code to work; if you are using MATLAB or do not want to install the extra python package, just record the animation on your phone and upload it). The input `plan` *has* to be formatted as either a MATLAB vector or Python list: e.g. `[0.1, -0.3, ..., 0.21]`. Discussion for part (c) and best distance your method achieves.

[1] This part of the assignment, and the implementations of the QWOP physics engine, are based on an assignment developed by Paul Valiant at Brown University, and we thank him for sharing his code, etc.

[2] `http://www.foddy.net/Athletics.html`