Problem Set 6

All parts are due Thursday, May 5 at 11:59PM. Please download the .zip archive for this problem set, and read the comments in the gradient descent template file carefully to ensure you format your solutions correctly.

Part A

Problem 6-1. Gradient Descent [20 points]

In this problem, we will use gradient descent to locate the value of x that minimizes a parabolic function $f(x) = ax^2 + bx + c$, where $a > 0$. We use the notation x^* to denote this "minimizing" value of x. The idea is to start with an initial value x_0. Then for each successive value of $i = 0, 1, \ldots$, we compute $x_{i+1} = x_i - \eta f'(x_i)$, where η is some constant positive-real-valued *step size*.

Please provide explanations to your solutions wherever appropriate.

(a) [2 points] First consider the particular function $f(x) = (2x - 3)^2$, with starting value $x_0 = 0$. Find a specific value of η for which the sequence of x_i values fails to converge to the correct value $x^* = \frac{3}{2}$. What is the sequence $x_0, x_1, \ldots$ generated using this value of η?

(b) [2 points] For the same function $f(x) = (2x - 3)^2$, with the same starting value $x_0 = 0$, find another value of η for which the sequence does converge to $x^* = \frac{3}{2}$. Try to choose η so that the algorithm converges quickly. In particular, consider an "accuracy" parameter $\epsilon > 0$, and define an index i to be ϵ-good provided that $|f(x_j) - f(x^*)| \leq \epsilon$ for every $j \geq i$. Find η so that the smallest index i that is ϵ-good is $O(\log \frac{1}{\epsilon})$.

(c) [3 points] Generalize your work in part (b) to give a good step size for an arbitrary quadratic function of the form $f(x) = ax^2 + bx + c$, where $a > 0$, and with x_0 an arbitrary real number. In particular choose a step size η so you approach the minimizer x^* exponentially fast. In other words, choose η such that the smallest index $i > 1$ that is $\epsilon-$good is equal to $c \log\left(\frac{a|x_0 - x^*|}{\epsilon}\right)$ for some constant c where x^* is the minimizer of the function $f(x)$. Try to choose η so that the smallest index i that is ϵ-good is $O(\log\left(\frac{a \cdot |x_0 - x^*|}{\epsilon}\right))$, where x^* is the minimizer of the function f.

Using your step size, give pseudocode for an algorithm that takes an arbitrary quadratic function of the given form and an arbitrary real number x_0, and outputs a value x such that $|f(x) - f(x^*)| \leq \epsilon$.

(d) [3 points] Again consider an arbitrary quadratic function of the form $f(x) = ax^2 + bx + c$, where $a > 0$. Now consider a different strategy for finding an approximation to x^*, based on binary search. (Note that this strategy works well for single-variable functions, but does not extend to multi-variable functions, whereas gradient descent still works in such cases.) This time, suppose we are given two values x_0 and x_1, such that $f'(x_0) < 0$ and $f'(x_1) > 0$. We are also given an accuracy parameter $\epsilon > 0$. Describe in words and pseudocode a simple algorithm that outputs a value x such that $|f(x) - f(x^*)| \leq \epsilon$. Your algorithm should take time $O(\log\left(\frac{a \cdot |x_0 - x_1|}{\epsilon}\right))$.

(e) [4 points] Show that for this very special case (very simple single variable convex function) Gradient Descent can find the optimal minimizer x^* in one single step. In particular find a better setting of the step size η such that one can decrease the objective function $f(x)$ as much as possible with the current estimate of the gradient for any Gradient Descent update. Briefly, comment on why your method works as well as it does for this type of function and why it might not work for other more complicated functions.

[**Hint:** consider the Taylor expansion of $f(x)$ and the error term considered in lecture and choose a step size that causes the greatest decrease in $f(x)$ while still approaching $f(x)$]

(f) [4 points] In the previous question one was able to find an extremely good step-size η in part because quadratic functions are very simple convex functions. In general that is not the case, however, one can still get a good step size in general. In particular given the trajectory of greatest change of $f(x)$ one can try to compute the step-size η that gives us the greatest decrease in our objective function $f(x)$ while still approaching some local minimum of f without overshooting it. Concretely, one could solve the following optimization problem at each gradient descent iteration to get the ideal step-size:

$$\eta^{(t+1)} = arg\min_{\eta \in R} f(x^{(t+1)}), \text{ subject to } x^{(t+1)} = x^{(t)} - \eta \nabla f(x^{(t)})$$

Describe an efficient algorithm to compute an approximation to the optimal step-size if you are given an upper bound η_{max} on the best η^*. Also provide the update step for Gradient Descent such that it incorporates this method to choose the step size (notice that you may break ties arbitrarily).

[**Hint 1:** notice the above optimization is over a single variable]

[**Hint 2:** use part (d) as inspiration and assume $f(x)$ is easy to compute]

(g) [2 points] Now we consider gradient search again, but now for a cubic function, say $f(x) = x^3$. Let the step size η be 1. For which values of x_0 does gradient search converge? (You don't need to identify the value of x to which it converges.)

Problem 6-2. [20 points] **Optimal Merging**

Fodder's, Inc. is trying to finalize its 2016 rankings of Boston restaurants. It has two lists of restaurants, say $A_1, A_2, \ldots, A_m$ and $B_1, B_2, \ldots, B_n$, each of which is already ranked from highest to lowest. (For example, A might contain relatively new restaurants, whereas the B restaurants have been around awhile.) All Fodder's must do is to merge the lists into a single list, preserving the order of the two sublists.

It turns out that some of the A restaurants care very much that they appear ahead of certain B restaurants on the merged list, and vice versa. In fact, they care enough that they offer to donate money to a local food bank if they obtain that ordering.

Formally, the A restaurants' proposed donations appear in a matrix MA, where $MA_{i,j}$ represents the (nonnegative integer) number of dollars that A_i will donate if it is ranked ahead of B_j. Similarly, the B restaurants' proposed donations appear in a matrix MB, where $MB_{i,j}$ represents the number of dollars that B_j will donate if it is ranked ahead of A_i.

The *value* of a merge is the total amount of money that would be donated as a result of that merge. Formally, that is the sum of all the entries $MA_{i,j}$ for pairs such that A_i precedes B_j in the merged list, plus all the entries $MB_{i,j}$ such that B_j precedes A_i in the merged list. Fodder's is interested in determining the maximum possible value of any merge of the two lists.

In all parts of this problem, provide both a clear explanation and pseudocode.

(a) [5 points] Describe a recursive algorithm to solve the optimal merge problem. The inputs are:

- A pair of lists $A_1, \ldots, A_m$ and $B_1, B_2, \ldots, B_n$.
- Matrices MA and MB of integers; both have dimensions m by n.

The output should be the largest value of a list that can be obtained by merging lists $A_1, \ldots, A_m$ and $B_1, \ldots, B_n$, where "value" is defined above.

Analyze the time complexity of your algorithm.

(b) [5 points] Use Dynamic Programming with memoization to make your recursive algorithm from part (a) more efficient. Analyze the time complexity of the resulting improved algorithm.

(c) [5 points] Now use Dynamic Programming with bottom-up calculation to make your recursive algorithm from part (a) more efficient. Analyze the time complexity of the resulting improved algorithm.

(d) [5 points] Modify one of your efficient algorithms to produce an actual optimal merge of the two input lists. Analyze the time complexity of the merge algorithm.

Problem 6-3. [20 points] **Optimal Graph Paths**

Here is another problem about finding "best paths" in the directed graphs. This time, we are given a weighted directed graph $G = (V, E, w)$ with designated start node s and designated target node $t \neq s$. The weights are nonnegative integers. We are also given a positive integer k and a particular vector $w = (w_1, w_2, \ldots, w_k)$ of nonnegative integer weights, representing the desirable number

of edges in the path and the desirable weights for the edges (in order on the path). Our job is to find a k-edge path from s to t whose weight vector is closest to the vector w, in the sense of minimizing the sum of squares of the differences of the edge weights. That is, we need a vector $x = (x_1, x_2, \ldots, x_k)$ for which the sum $\Sigma_{i=1}^{n}(x_i - w_i)^2$ is minimum.

(a) [5 points] Give pseudocode for a recursive algorithm to determine the minimum achievable cost of a path, in the problem described above. The inputs should be the directed graph G, the number k and the vector w. You may assume that G is given in the usual adjacency-list format, and may also assume an adjacency-list representation for the "reverse" of G—the same graph in which all the edges are reversed (that might help). The output should be the smallest cost for a path from s to t consisting of exactly k edges, where the cost of the path is the sum of squares of differences in edge weights, as described above. If no k-edge path from s to t exists, then your algorithm should output ∞.

Analyze the time complexity of your algorithm.

(b) [4 points] Use Dynamic Programming with memoization to make your recursive algorithm from part (a) more efficient. Analyze the time complexity of the resulting improved algorithm.

(c) [4 points] Now use Dynamic Programming with bottom-up calculation to make your recursive algorithm from part (a) more efficient. Analyze the time complexity of the resulting improved algorithm.

(d) [3 points] Describe a situation in which the memoized version of the algorithm is more efficient than the bottom-up version.

(e) [4 points] Modify one of your efficient algorithms to produce an actual best path. Analyze the time complexity of the path-finding algorithm.

Part B

Problem 6-4. [40 points] **Campaign Madness**

Ben Bitdiddle, starting from his humble beginnings at MIT, is now running for president of the United States under the "Fewer Psets Party". In order to most effectively campaign, Ben wants to identify swing voters, that is voters who have a decent chance of voting for Ben given some more directed campaigning. To this end, Ben has examined polling data very closely, trying to figure out how factors such as age or income influence how positively voters view him. Using this data, he wants to come up with a prediction how much a previously-unseen voter may approve of Ben.

Explicitly, Ben is given a $n \times m$ matrix, θ which contains m different pieces of demographic information about n individuals. For instance, the first row of θ will consist of m different non-negative real numbers that correspond to some quantitative information (like age, income, number of children, years of formal education, hand size, etc.) about a particular person. For instance, if Ben polled three voters about just their age and handsize in centimeters, the matrix might look like this:

$$\begin{bmatrix} 21 & 20 \\ 44 & 15 \\ 64 & 17 \end{bmatrix}$$

Ben also has a length n vector $\mathbf{y}$ of real numbers, the approval scores, which represent how much a person approves of Ben. The more positive the score, the more a person approves of Ben. For instance, the i^{th} element of $\mathbf{y}$ corresponds to the approval score of the i^{th} person.

In order to predict the approval score of a new voter, Ben wants to construct a model that generates a guess, $\hat{y}$ based on the demographic information of that voter. He does this for the i^{th} voter as

$$\hat{y}(\theta_{\mathbf{i}}) = \sum_{j=1}^{m} \theta_{ij} x_j = \theta_{\mathbf{i}} \cdot \mathbf{x} \tag{1}$$

Where $\theta_{\mathbf{i}}$ denotes the i^{th} row of θ and θ_{ij} the element in the j^{th} column of the i^{th} row. In other words, we compute the approval score as the inner product between a voter's attributes and some vector of parameters $\mathbf{x}$. Note that for our input we will set all $\theta_{i1} = 1$, so that the x_1 term functions as a constant offset for our model.

Ben wants to build the best model possible and wants to do so by picking the vector of parameters $\mathbf{x}$ that minimize the mean-squared error, $J(\mathbf{x})$, that is he wants to find a $\mathbf{x}^*$ that minimizes

$$J(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \theta_{\mathbf{i}} \cdot \mathbf{x})^2 = \frac{1}{n} (\mathbf{y} - \theta\mathbf{x})^\top (\mathbf{y} - \theta\mathbf{x}) \tag{2}$$

Once Ben finds the optimal $\mathbf{x}$ for this cost function, he will generate guesses of the approval score of brand new voters and in turn decide whether or not to devote resources campaigning for those people. You won't have to make the decision of where Ben has to campaign, but he needs your help to build this model!

(a) [5 points] In order to perform the gradient descent algorithm, we first need an update equation. Analytically derive the update step for the gradient descent algorithm. You may leave the step size as η for now.

(b) [10 points] Implement gradient descent as specified in the template for the function **gradient_descent**. Your input will be a $n \times m$ 2-d array representing m different pieces of demographic information for n different voters, a length n array for those voter's approval scores for Ben, a length m vector representing your initial guess for the parameters of this model, and a float for the step size you should use. You should return the vector of nearly optimal $\mathbf{x} = [x_1, ...x_m]$ as a list. Also, consider stopping the gradient descent procedure when the mean-squared value of the vector $\mathbf{x^t} - \mathbf{x^{t-1}}$ is below a threshold. In other words, once the update stops changing the parameters much, we should stop. See the code template for more details.

(c) [10 points] Ben has managed to poll an incredible amount of people, but realizes that if he picks a random subset of people on which to perform his update to the gradient, the overall algorithm will converge to an optimal $\mathbf{x}$ much quicker. Specifically, if he chooses a subset of ten voters, S, at random, he can perform an update with the gradient of a modified cost function for just these voters.

$$J(\mathbf{x})^{mini} = \frac{1}{10}\sum_{i \in S}(y_i - \theta_\mathbf{i} \cdot \mathbf{x})^2 \tag{3}$$

He can then iterate through all the voters in sets of ten following some random order. Given the same set of inputs as part (b), implement **minibatch_gradient_descent** by choosing a random subset of 10 voters on which to perform a subgradient update on.

Hint: This is a slightly new algorithm! The comments in the code template should be helpful in figuring out how to adapt your algorithm from part (b).

(d) [10 points] Ben realizes something else as he's refining his algorithm. The error is convex in the parameters $\mathbf{x}$. In other words, when traveling along a trajectory determined by the gradient, there is a single minimum value. Starting with a minimum estimate of the step size at some point and the gradient vector, he decides to use a repeated doubling procedure to find the step size that gives us a new $\mathbf{x}$ along that trajectory at iteration of his algorithm. Your task is to implement a search along the direction of the gradient following this pseudocode:

DOUBLING-LINE-SEARCH$(x_{start}, \nabla f, \eta_{min}, \eta_{max})$

```
1  Initialize η_current = η_min, x_current = x_start − η_current ∇f(x_current)
2  while η_current < η_max
3          x_temp = x_current − η_current ∇f(x_current)
4          if J(x_temp) < J(x_current)
5                  x_current = x_temp
6                  η_current = 2η_current
7          else break
8  return x_current
```

where J is our error function. Implement this in the function **line_search**. Furthermore, implement two other simple helper functions, **compute_gradient** (which is exactly the same as in part (b)) and **prediction_error**. These functions will be used as subroutines in an otherwise complete **gradient_descent_complete** which is the function that will be tested.

(e) [5 points] Alyssa P. Hacker, Ben's longtime friend and data science consultant, notes that there is a single global minimum for $J(\mathbf{x})$ and that he could have just come up with a closed-form solution for $\mathbf{x}$. Your last task is to find the closed form for $\mathbf{x}$ in terms of the matrix θ and vector $\mathbf{y}$.

Hint: It may be useful to know that for a matrix $\mathbf{A}$ and vector $\mathbf{v}$ that $\frac{\partial \mathbf{A}\mathbf{v}}{\partial \mathbf{v}} = \mathbf{A}^\top$ and that $\frac{\partial \mathbf{v}^\top \mathbf{A}}{\partial \mathbf{v}} = \mathbf{A}$.