New York University Tandon School of Engineering
Computer Science and Engineering

CS-GY 6763: Homework 3.
Due Tuesday, November 21st, 2021, 11:59pm.

Collaboration is allowed on this problem set, but solutions must be written-up individually. Please list collaborators for each problem separately, or write "No Collaborators" if you worked alone.

Problem 1: Concentration of Random Vectors

(10 pts) In Stochastic Gradient Descent, we replace the true gradient vector with a stochastic gradient that is equal to the true gradient in expectation. Our analysis in class *only used* equality in expectation, although more refined analysis of SGD often requires understanding how well the stochastic gradient *concentrates* around its expectation. Previously, all concentration results we studied apply to random numbers. For this problem, you will prove a basic concentration inequality for random vectors.

In particular, let $\mathbf{x}_1, \ldots, \mathbf{x}_k \in \mathbb{R}^d$ be i.i.d. random vectors in d dimensions (independent, drawn from the same distribution) with mean $\boldsymbol{\mu}$. I.e., $\mathbb{E}[\mathbf{x}_i] = \boldsymbol{\mu}$. Further suppose that $\mathbb{E}\left[\|\mathbf{x}_i - \boldsymbol{\mu}\|_2^2\right] = \sigma^2$. σ^2 is a natural generalization of "variance" to a random vector. Let $\mathbf{s} = \frac{1}{k}\sum_{i=1}^{k} \mathbf{x}_i$. Prove that if $k \geq O(\frac{1/\delta}{\epsilon^2})$ then

$$\Pr\left[\|\mathbf{s} - \boldsymbol{\mu}\|_2 \geq \epsilon\sigma\right] \leq \delta.$$

Problem 2: Regularization

(10 pts) Regularization is a popular technique in machine learning. It is often used to improve final test error, but can also help speed-up optimization methods like gradient descent by improving the condition number of the function being regularized. In particular, let $f(\mathbf{x})$ be a differentiable function mapping a length d vector $\mathbf{x}$ to a scalar value. Let g be the function with added Euclidean regularization:

$$g(\mathbf{x}) = f(\mathbf{x}) + \lambda\|\mathbf{x}\|_2^2$$

Above $\lambda > 0$ is a non-negative constant that controls the amount of regularization. Suppose f is α_1-strongly convex and β_1-smooth, so has condition number β_1/α_1. Prove that g is also convex and its condition number less than or equal to that of f.

Problem 3: Separation Oracles

(12 pts) Describe efficient separation oracles for each of the following families of convex sets. Here, "efficient" means linear time plus $O(1)$ calls to any additional oracles provided to you.

(a) The set $A \cap B$, given separation oracles for A and B.

(b) The ℓ_1 ball: $\|\mathbf{x}\|_1 \leq 1$.

(c) Any convex set A, given a *projection oracle* for A. Recall that a projection oracle, given a point $\mathbf{x}$, returns

$$\text{Proj}_A(\mathbf{x}) = \arg\min_{y \in A} \|\mathbf{x} - \mathbf{y}\|_2.$$

Above you may wish to use the following fact that was stated but not proven in class: for any point $\mathbf{x}$, convex set A, and $\mathbf{z} \in A$, $\|\mathbf{z} - \text{Proj}_A(\mathbf{x})\|_2 \leq \|\mathbf{z} - \mathbf{x}\|_2$.

Problem 4: Gradient Descent with Decaying Step-size

(10 pts) In class we showed that gradient descent with step size $\eta = R/G\sqrt{T}$ converges to an ϵ approximate minimizer of a convex G-Lipschitz function in $T = R^2G^2/\epsilon^2$ steps if our starting point $\mathbf{x}^{(0)}$ satisfies $\|\mathbf{x}^{(0)} - \mathbf{x}^*\|_2 \leq R$. Choosing this step size requires knowing G, R and moreover T in advance, which might not be reasonable in a lot of settings. For example, when training machine learning models, we might not be able to estimate how long it will take to reach a point where test accuracy levels off. Instead, we want to be able to keep running the algorithm, achieving better and better accuracy as we do.

Here, we analyze a variant of gradient descent with a variable step size that avoids this limitation. In particular, consider running gradient descent with the update $\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} - \eta\nabla f(\mathbf{x}^{(i)})$, where

$$\eta = \frac{f(\mathbf{x}^{(i)}) - f(\mathbf{x}^*)}{\|\nabla f(\mathbf{x}^{(i)})\|_2^2}.$$

This step size requires knowledge of $f(\mathbf{x}^*)$, but not $\mathbf{x}^*$, which may be reasonable in some settings. Moreover, since it's just one parameter, grid search can be more easily used to "guess" $f(\mathbf{x}^*)$ than the three parameters G, R, T. More complex approaches can remove the need to know this value entirely.

Prove that, if we run gradient descent for $T = O(R^2G^2/\epsilon^2)$ steps using the step size above then $\hat{\mathbf{x}} = \min_{i\in 0,\ldots,T} f(\mathbf{x}^{(i)})$ satisfies $f(\hat{\mathbf{x}}) \leq f(\mathbf{x}^*) + \epsilon$. **Hint:** Prove that our distance from the optimum $\|\mathbf{x}^{(i)} - \mathbf{x}^*\|_2$ always decreases with this choice of step size, and the decrease is larger if our gap from the objective value $f(\mathbf{x}^{(i)}) - f(\mathbf{x}^*)$ is larger.

Problem 5: Locating Points via the SVD

(15 pts) Suppose you are given all pairs distances between a set of points $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$. You can assume that $d \ll n$. Formally, you are given an $n \times n$ matrix $\mathbf{D}$ with $\mathbf{D}_{i,j} = \|\mathbf{x}_i - \mathbf{x}_j\|_2^2$. You would like to recover the location of the original points, at least up to possible rotation and translation (which do not change pairwise distances). Since we can only recover up to a translation, it may be easiest to assume that the points are centered around the origin. I.e. that $\sum_{i=1}^n \mathbf{x}_i = \mathbf{0}$.

(a) Under this assumption, describe an efficient algorithm for learning $\sum_{i=1}^n \|\mathbf{x}_i\|_2^2$ from $\mathbf{D}$.

(b) Next, describe an efficient algorithm for learning $\|\mathbf{x}_i\|_2^2$ for each $i \in 1, \ldots, n$.

(c) Finally, describe an algorithm for recovering a set of points $\mathbf{x}_1, \ldots, \mathbf{x}_n$ which realize the distances in $\mathbf{D}$. Hint: This is where you will use the SVD! It might help to prove that $\mathbf{D}$ has rank $\leq d + 2$.

(d) Implement your algorithm and run it on the U.S. cities dataset provided in `UScities.txt`. Note that the distances in the file are unsquared Euclidean distances, so you need to square them to obtain $\mathbf{D}$. Plot your estimated city locations on a 2D plot and label the cities to make it clear how the plot is oriented. Submit these images and your code with the problem set.