CVXOPT II: Pset 4

Justin Lewis

February 2019

1 Problems 1 and 2

1.1 Part a)

The update for the Bregman projection can be found in the written problem portion of my submission. As can the formulation for the Bregman divergence. The update for projection onto the simplex was derived in a previous homework for vectors in the positive orthant; however, the projection from anywhere is not much different, my implementation can be found in the code.

1.2 Part b)

Convergence Results

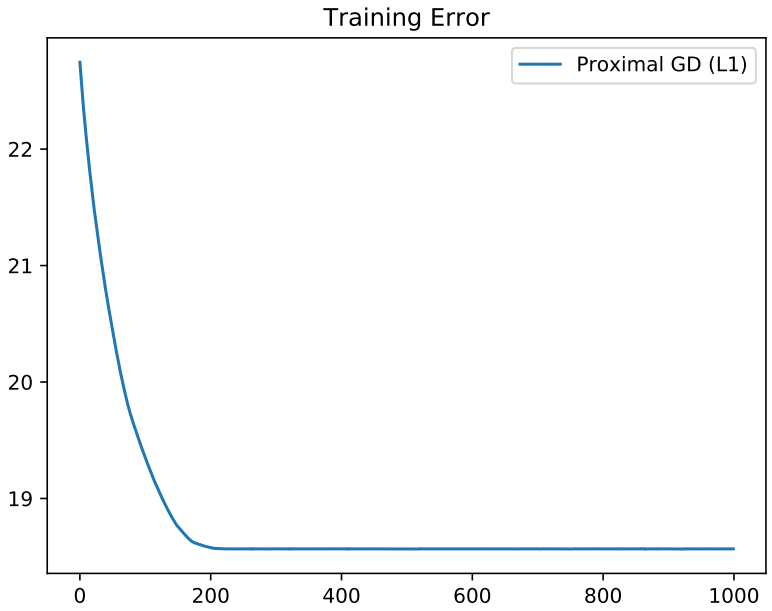

Figure 1: ℓ_1 training error (corrupted y) vs iterations for solving regression

Comments: I tried to compare the performance of using ℓ_1 vs. ℓ_2 loss in terms of robustness to value corruption; however, it seems that the corrupted version was actually the same as the original after comparing their values numerically. Mirror descent seemed to do much better in the end.

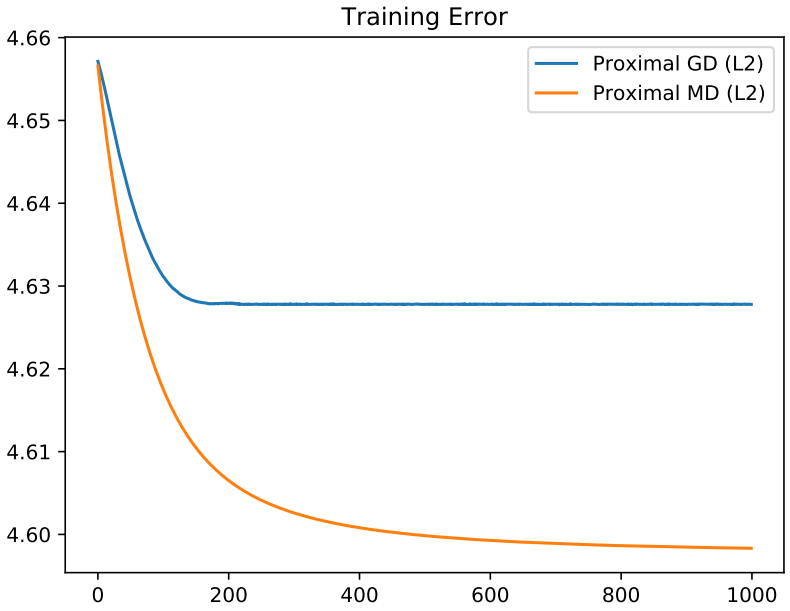

Figure 2: ℓ_2 Training error (corrupted y) vs iterations for solving regression

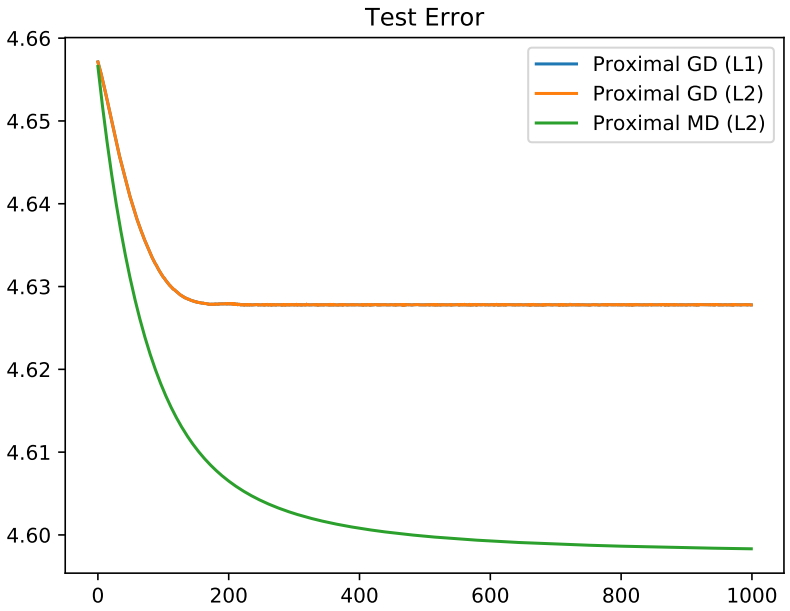

Figure 3: Test error (true y) vs iterations for solving regression

2 Problem 3

2.1 Part a)

An element of the subdifferential can be found by using np.linalg to find the SVD of the matrix X. This gives us U and V. Then, one can find W by again using np.linalg to find the null-space and left nullspace of the matrix UV^T.

To ensure the spectral norm of a matrix is ¡1, one can merely divide by the largest singular value of that matrix (found using np.linalg).

2.2 Part b)

I chose to use a naive approach of simply taking my current estimate $\hat{X}$ and seting $\hat{x}_{ij} = m_{ij}$ for all $(i, j) \in \Omega$ after each gradient step.

2.3 Part d)

Convergence Results

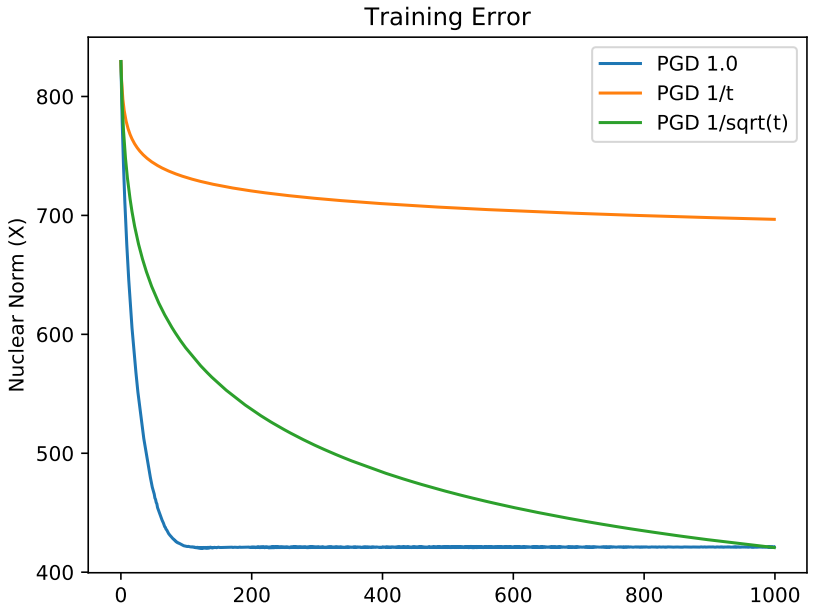

Figure 4: Three different step size strategies compared.

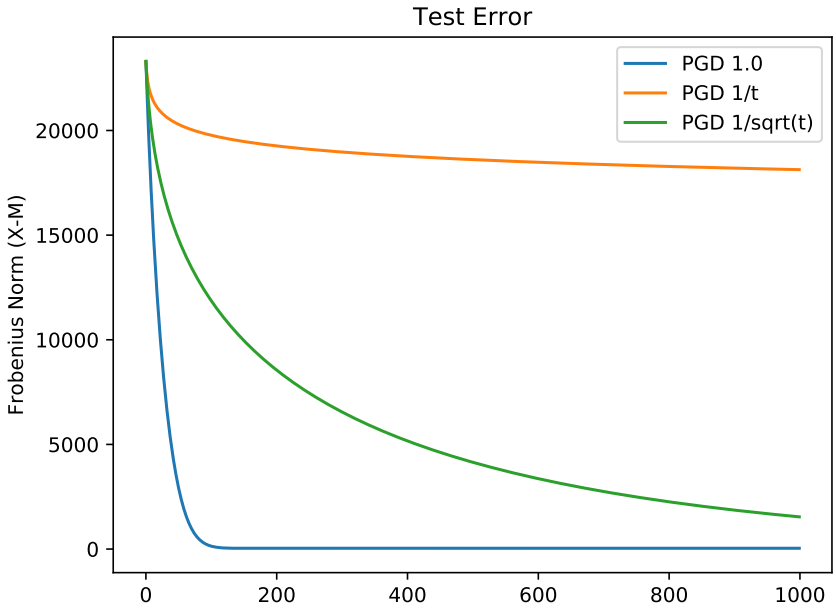

Figure 5: Three different step size strategies compared.

Comments: I compared the two step sizes asked for as well as a constant step size of 1. It seemed that the constant step size was most efficient at recovering the sparse matrix.

2.4 Part e)

My starting matrix was the all zeros matrix; therefore, the initial rank was 0. One would expect that as the process goes on, the rank of the estimate matrix should converge to the rank of the true matrix (if recovered exactly). However, perhaps this is only partially true, the nuclear norm is the sum of the matrix's SV's, thus minimizing it should promote sparsity in the SV's; however, it's unclear what would actually happen. I plotted this to see and surprisingly my matrix estimate quickly became full rank. (my guess is that most of the singular values are very small but not exactly 0).

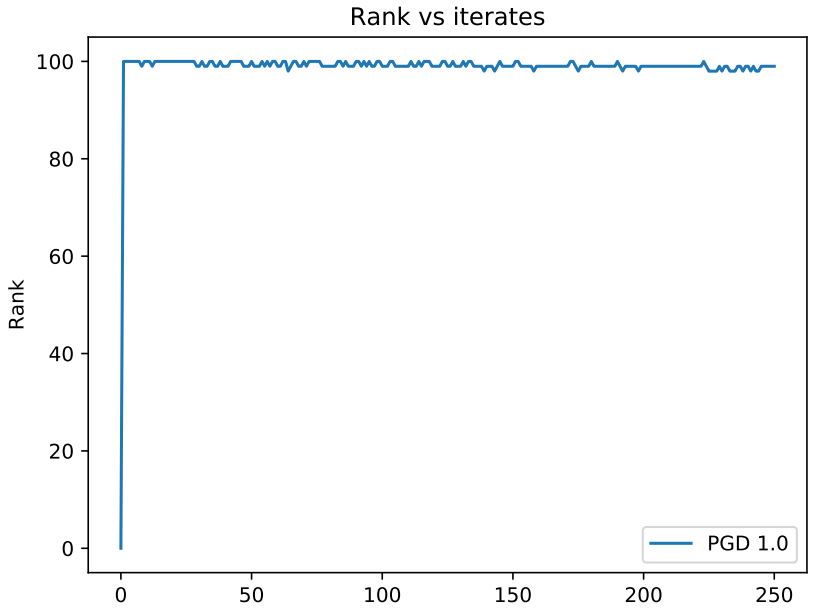

Figure 6:

3 Problem 4

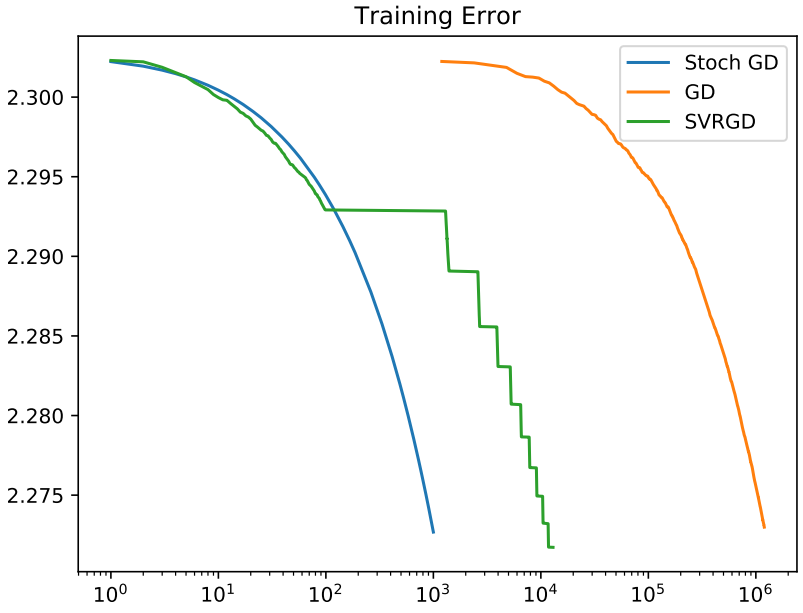

Figure 7: Training Error vs iterations. SVRG interpolates between SGD and GD

Comments: although, not shown, varying the frequency of the update step in SVRG essentially interpolates between SGD with a single sample, and full gradient descent. However, if the update step is not done frequently enough, I would guess that SVRG would gain no benefit and possibly suffer some detriment as the full gradient

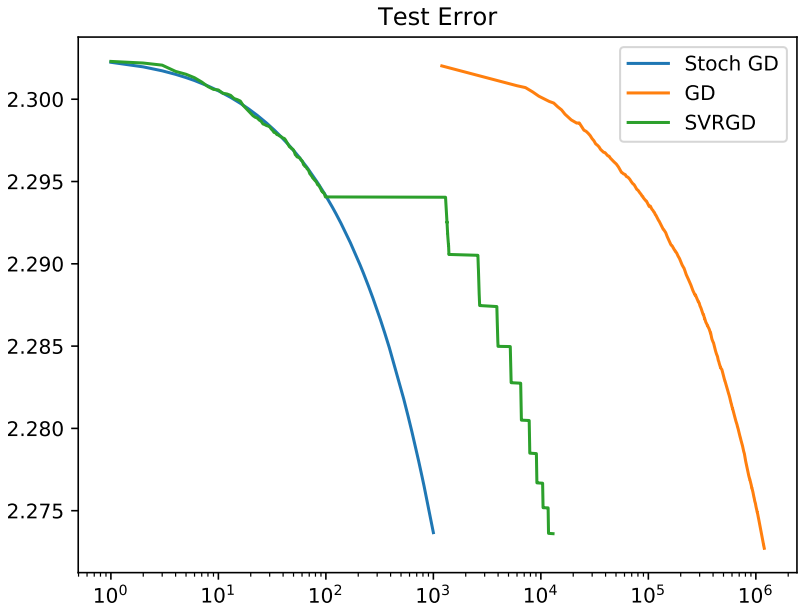

Figure 8: Test Error vs iterations. SVRG interpolates between SGD and GD

would become quite stale.

4 Problem 5

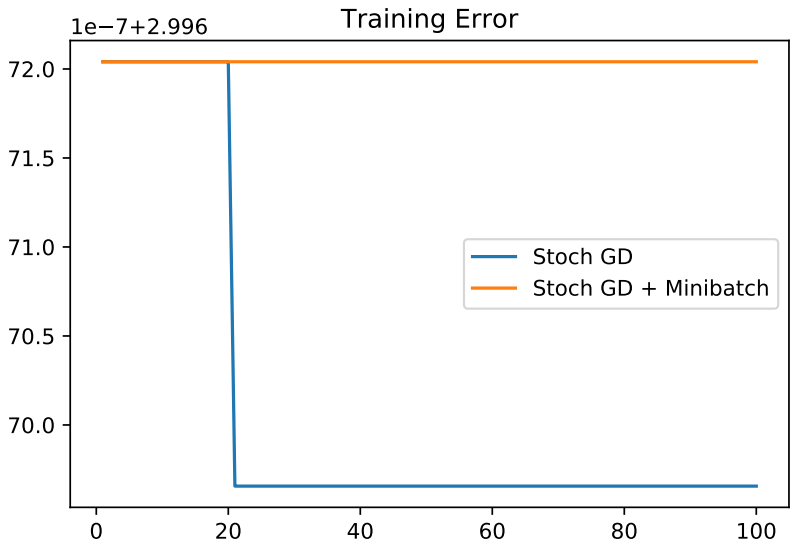

Figure 9: Logistic training loss vs. iterations

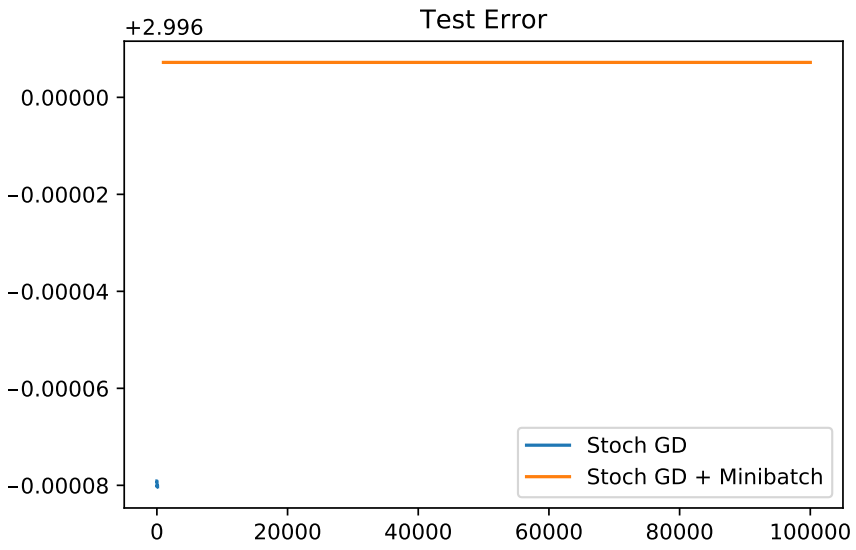

Figure 10: Logistic test loss vs. iteration

Comments: I had no luck using PGD with single samples or with minibatches to learn on the full dataset. I'm not sure if there is a bug in the code or if perhaps the minibatch size was not large enough. If i had more time, (running the training is prohibitively slow), I would have done the following to enhance training:

* implemented a chi-squared test to reduce the number of relevant feautres * used a combination of ℓ_2 and ℓ_1 regularization, known as the elastic net regularization (which is known to have nice properties). *used numpy's / torch's functions for sparse matrix operations.