EP3260: Fundamentals of Machine Learning over Networks
Computer Assignment 2, Due Date: March 4, 2020
Group 4: H. Hellström, M. Tavana, F. Vannella, and M. Zeng

Computer Assignment 2. *(Deterministic/stochastic algorithms in practice)*
Let us consider the logistic ridge regression

$$f(\boldsymbol{w}) = \frac{1}{N}\sum_{i\in[N]} f_i(\boldsymbol{w}) + \lambda \|\boldsymbol{w}\|_2^2,$$

where $f_i(\boldsymbol{w}) = \log\left(1+\exp\left\{-y_i \boldsymbol{w}^T \boldsymbol{x}_i\right\}\right)$ for the "Greenhouse gas observing network" dataset. Then, address the following:

(a) Solve the optimization problem using GD, stochastic GD, SVRG, and SAG;

(b) Tune a bit the hyper-parameters (including λ);

(c) Compare these solvers in terms of complexity the hyper-parameter tunning, convergence time, convergence rate (in terms of # outer-loop iterations), and memory requirement.

1 Data handling

Dataset description

The "Greenhouse gas observing network" dataset consists of 2921 files. Each file contains a matrix of real numbers of dimension 327x16 (after taking a transpose). The first 15 columns correspond to GHG concentrations of tracers emitted from regions 1-15 (we don't care what these regions correspond to) and the final column corresponds to GHG concentrations of synthetic observations. In our solution, we use a linear model

$$\underline{y_i = w_0 + w_1(x_i)_1 + w_2(x_i)_2 + \ldots + w_{15}(x_i)_{15},} \quad ✓. \tag{1}$$

where i corresponds to the row index.

This model is used to fit the 15 tracer GHG concentrations to the synthetic concentration. Even though the problem considers logistic regression, we have chosen to keep the y-value intact rather than discretizing it into binary classes. ✓

File handling

The process of opening, reading, and closing 2921 files in python is slow, so we do not want to read them at each run. Instead, we have chosen to write a preprocessing script (preformat.py) which reads all 2921 files and creates one huge (327*2921)x16 matrix. Still, the first 15 columns correspond to x and the final to y, we just have more rows. This matrix is written into a new file (merge.txt) which can be loaded much quicker than the 2921 small files.

Data processing

Before applying the solvers, we do a bit of data processing. First, the data is scaled down because we noticed overflow errors when multiplying $\boldsymbol{y}$ and X with the raw data. The data is scaled by applying

$$D = D/(b) * 100, \tag{2}$$

where D is a (327*2921)x16 matrix holding the entire dataset and b is the biggest element in the dataset. The reason we multiply by 100 is because there are some huge outliers in the dataset, so by scaling from 0-1 we saw that the majority of data got so small that the gradient would occassionally be rounded to 0.

Next, we make the dataset zero mean by applying:

$$D = D - \mu, \tag{3}$$

where μ is the mean of all elements in D. Finally, the dataset is split into X and $\boldsymbol{y}$ by taking the first (327*2921)x15 elements of D as X and the final (327*2921)x1 elements of D as $\boldsymbol{y}$.

2 Gradient

For all solvers, we need to calculate the gradient of the cost function. The derivation is fairly straight-forward and the result is:

$$\nabla f(w) = \frac{1}{N} \sum_{i \in [N]} -\frac{y_i \boldsymbol{x_i}}{\exp\{y_i \boldsymbol{w}^T \boldsymbol{x_i}\} + 1} + 2\lambda \boldsymbol{w} \tag{4}$$

3 Hyperparameter tuning

Regularization term λ

When selecting λ, our goal was to find a balance between the two terms in the gradient (4). If λ is too small, the first term will dominate and the generalization error becomes meaningless. On the flip side, an overly large λ will cause the solver to only care about reducing the size of the weights, completely neglecting fitting of the model. By comparing the size of these terms at runtime, we selected $\lambda = 0.01$. Of course, this is not the best way to do it, but it was very fast.

Something that seems better would be too select lambda by looking at overfitting. For instance we could try training on a range of λ's and comparing the fit on the test data. Unfortunately, we were short on time, so we did not do this sort of tuning.

Learning rate α

The idea behind selecting learning rate was to get something large, without causing divergence. Thus, we tried increasing α until the error started growing rather than shrinking, which happened around $\alpha = 10$. So we selected $\alpha = 1$ to have some margin, while still being fast enough to converge in around 20 iterations.

Batchsize

For the batchsize, the tradeoff is between fast computational time (for small batches) and good representation of the true gradient (for larger batches). By trying a few different ones, we quickly found that even very small batchsizes lead to nice convergence. However when the batchsize is 1, we saw some noticable hiccups where the cost would suddenly grow, so we ended up selecting batchsize 10.

Stopping limit - ϵ

The stopping limit (what is the proper name?) is used to interrupt the solver if it has converged. The idea is that the gradient is going to be very small near the optimum (zero at the optimum), so if the gradient is below some threshold we stop iterating. In our case, the solvers would consistently converge in a few iterations (less than 50) so we preferred continuing to gather more data on the cost decline. Thus, we selected a very small $\epsilon = 10^{-7}$, but this is never occurs before we run out of iterations.

Summary

All parameters can be seen in table 1.

Parameter	Value
λ	0.01
α	1
Batch Size	10
ϵ	10^{-7}

Table 1: Expected solution to stochastic problem.

4 Results

4.1 GD ✓

With weights that are randomly initialized according to $\mathcal{U}(0,1)$, we get an initial cost of about 0.7. After training GD for 80 iterations with the parameters in table 1 we get a cost of about 0.22. The 80 iterations takes about 63 seconds to complete, given that we use the entire dataset of $(327 * 2921)x16 \approx 1.5 \cdot 10^7$ samples. A convergence plot can be seen in figure 1.

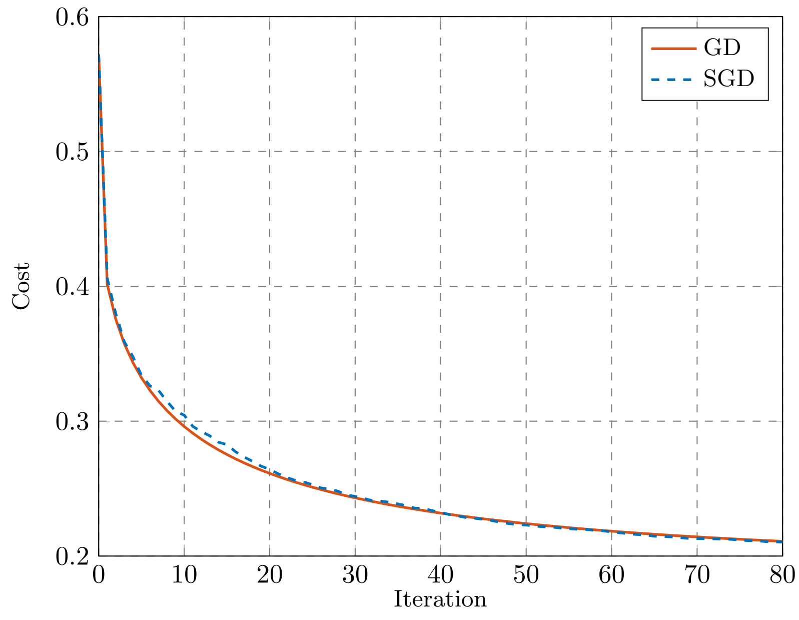

Figure 1: Caption

4.2 SGD

In an identical setup to the GD case (same parameters and initial weights), we can achieve similar results much faster. Once again, using the entire dataset for 80 iterations, with a batchsize of 10, it takes about 6 seconds to complete training. See fig 1 for a convergence plot.

4.3 SVRG

Once again, we attempted to use the same setup as GD, but with SVRG we would occassionaly diverge. Thus, the learning rate was dropped to $\alpha = 0.05$ while the remaining parameters stayed the same. Results were similar, ending up at a cost of about 0.22. The computational time was 11 seconds.

5 Comparisons ✓

Complexity of hyper-parameter tuning

Let's start by comparing GD and SGD. The majority of work was dedicated to tuning λ and α. In our case, we simply reused these parameters when moving from GD to SGD, and it immediately had good results. However, we have to presume that it would have been equally difficult if we started with SGD. On top of that though, we had to tune the batchsize for SGD, but it was very easy. The conclusion is that they are of very similar difficulty because tuning λ and α is the majority of the work, but SGD is slightly harder since it has one more parameter.

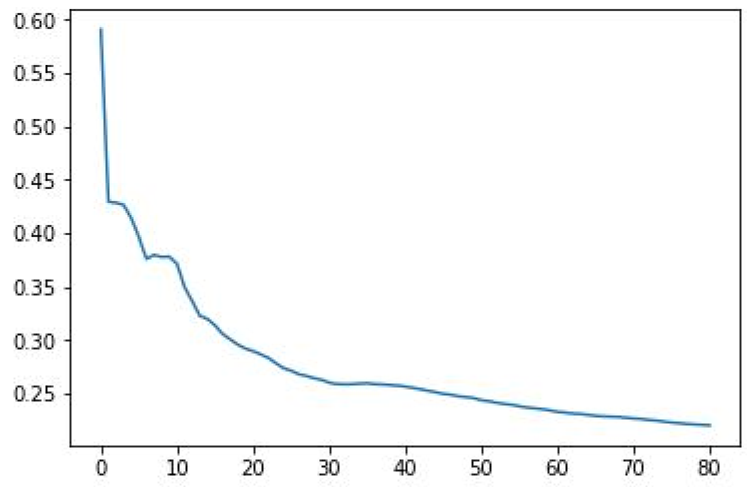

Figure 2: Cost per iteration for SVRG solver.

When it came to SVRG, the algorithm would often diverge using the same parameters, so the learning rate had to be significantly reduced to $\alpha = 0.05$. Despite this, the other parameters could remain the same though. I wouldn't say this was any more complicated than the others, probably about the same as SGD (since we also have batch size for SVRG).

Convergence time

Let's start by comparing GD and SGD. As expected, GD converges faster per iteration than SGD. This is natural, since GD uses the true gradient (which will always descend) while SGD uses a random sample (which could occassionaly ascend). However, the difference is not large, maybe a difference of 5 iterations, and SGD is a lot faster per iteration. So if we count convergence time in seconds, SGD wins out by a wide margin.

In an ideal world, SGD should run $(327 * 2921)/10 \approx 10^5$ (batchsize=10) times faster than GD, but when we ran 10 iterations of both, we saw an improvement of about 12.5 times faster, and when running 100 iterations we saw an improvement of 13.5 times. Our guess would be that overhead in terms of memory management is really significant so we don't see those kinds of improvements with 10 or 100 iterations. It seems to be slightly larger when the iterations increase, but not near the theoretical improvement. Perhaps the overhead is not constant with respect to the iterations.

SVRG was somewhere in between GD and SVRG. This makes sense, because we are on most iterations only calculating the batchsize gradient (as in SGD) but occassionaly take a new snapshot of the full gradient (as in GD) which makes it slightly slower than SGD. The convergence per iteration were worse than both other algorithms, probably because it had some troubles in the start. I think this is because we chose a fixed number of iterations between each "snapshot", which is not ideal when we are far from the optimum. As the weights approach optimum, the algorithm converges much faster.

Convergence rate

This was covered in the previous subsection.

Memory requirement

We could not collect any data since the resouce library in python is based on unix and we ran the experiments on a windows machine. However, it is clear that SGD should have the lowest memory requirements, since it's using 10 samples per iteration instead of several millions. Similarly, SVRG should be lower than GD since it only considers the full batch once every 10 iterations. Finally, GD will be by far the most consuming since it takes all samples for every iteration.

6 Conclusion

The three solvers we analyzed can be placed on a scale, from highest complexity to lowest. GD has a very high complexity since it requires a huge amount of samples for each iteration, but it is guaranteed to move in the right direction since there is no random sampling. On the other end, we have SGD, which is much more lighter as it's only considering a few samples per iteration. For big data scenarios, we are probably ill-advised using GD as it could take several hours to run single iterations on a modern PC, and then SGD is a better bet.

However, the implementation of SGD introduces random sampling to the solver which can cause the weights to diverge. To combat this, the batchsize can be increased, but this will of course also increase the computational complexity.

SVRG seems to be a middle ground between GD and SGD since it does consider both random sampling and the full gradient, but it does not update the full gradient for each iteration. My interpretation (which is probably terrible because I never heard of SVRG before doing this) is that the idea behind SVRG is to stabilize SGD by occassionaly taking a snapshot of the full gradient (which we know is pointing in the right direction) and use this same snapshot multiple times.

Unfortunately we ran out of time before implementing the SAG solver, we simply have to hope that the three solvers will be enough. ✓