Due: 8 Oct 2020

We selected the variables Median Income, Bachelor Rate, and Unemployment Rate for all our experiments. Both of us are submitting this report along with our own code. In Tyler's code, each question Q_j is answered in file `Qj.m` and supporting methods are coded in their own, separate files. Kusal's code is written in both Python and MATLAB: Q1 is in Project1/Question1.m, Q2 is in code.ipynb, Q3 is in Project1/Question3.m, and Q4 is in Project1/Question4.m. Additionally, supplementary methods (LBFGS_me.m, SLBFGS_me.m, SINewton_me.m) were put in Project1/ for minor changes in implementations of the Professors' code (e.g. adding a batch size parameter to function calls). Finally, extra data analysis was done in code.ipynb, which is discussed further at bottom.

1. **SVN.** Consider a subset of data from all 58 CA counties. The initial guess for the dividing plane is obtained by running inexact Newton's method. Running the code in `Project1main.m` yields `w`, whose first three components are w and whose last component is b.

(a) Set up the SVN constrained minimization problem with soft margins and solve it using the active-set method (ASM). Compare the dividing planes produced by minimizing the loss function and solving the SVN problem with soft margins. Which one looks more reasonable?

(b) Increase the dataset by adding counties from other states. Comment on the performance of the SVN. Do your observations motivate you to switch to unconstrained minimization of a reasonable loss function?

Solution. For part (a), we used both the active-set method (with a guess computed via inexact Newton's method) and the `FindInitGuess` function to minimize a ReLU loss function. Comparing the dividing planes created by these two methods as applied to the California counties, we concluded that using the inexact Newton's method was far superior for this classification problem (Figures 1 and 2). Minimizing the loss function yielded a plane which lumped many Democrat counties in with the Republican counties (Figure 1), while the inexact Newton's method's candidate for initial guess led to a dividing plane that cleanly separated the bulk of the Republican counties with the Democrat counties.

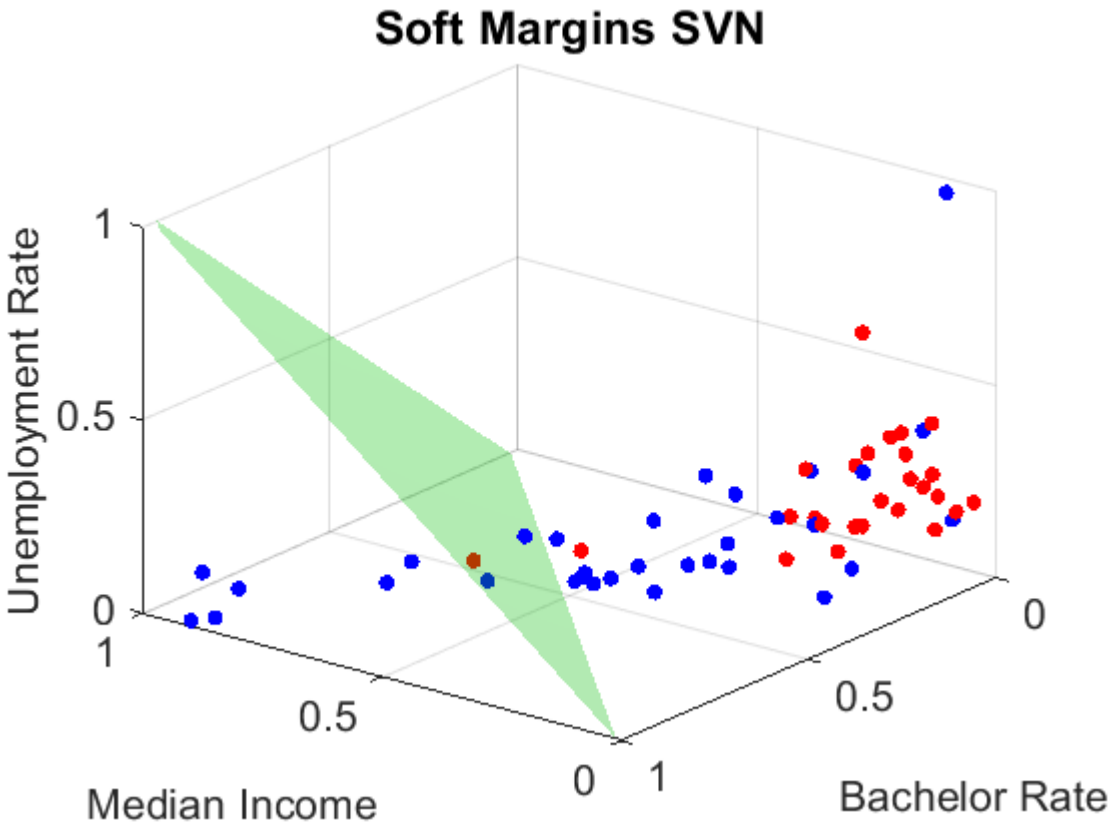

Figure 1: Dividing plane obtained from minimization of a (poorly chosen) loss function.

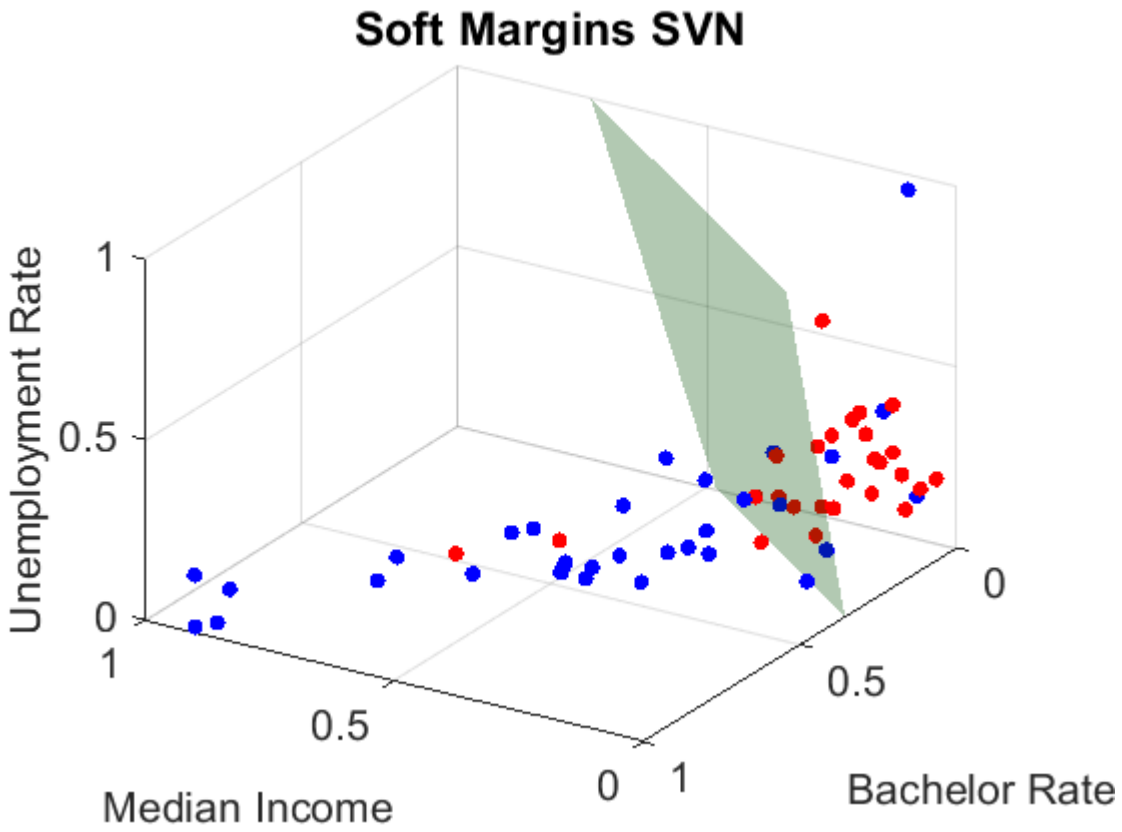

Figure 2: Dividing plane obtained from SVN with soft margins using the active-set method and applying inexact Newton's method for the initial guess.

To continue the analysis, we added the counties of Washington state and New Jersey to the California counties data. Figure 3 depicts the result of one run on all of these counties. We collected our performance information by using this extra data and averaging over $N = 100$ iterations. We found that the norm of the gradient decayed quite slowly, reaching the tolerance of 10^{-10} in 400-500 iterations (Figure 4). The runtimes between each of these trials varied by about 0.05 seconds with a few outliers (Figure 5). On the whole however, we found that ASM was often much slower than the unconstrained minimization methods that followed it, especially as the datasets grew—the addition of Washington and New Jersey counties made evident the poor scaling of ASM.

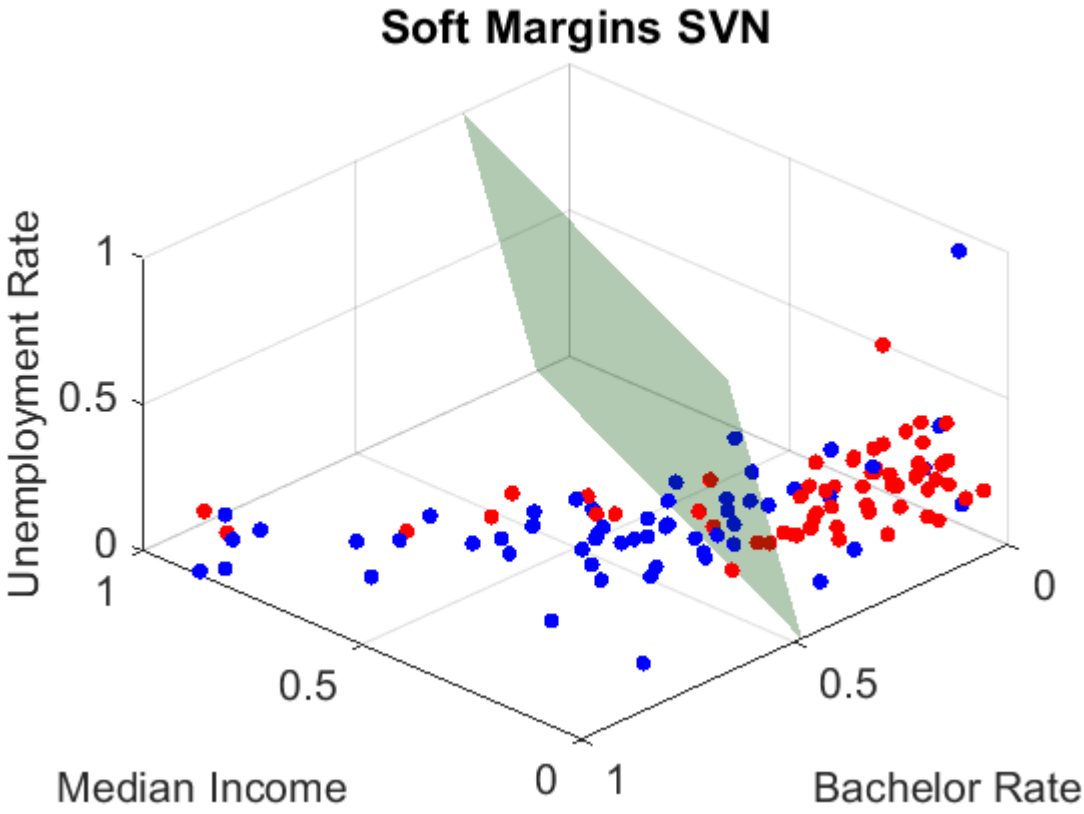

Figure 3: Result of SVN with soft margins on the counties data from California, Washington, and New Jersey.

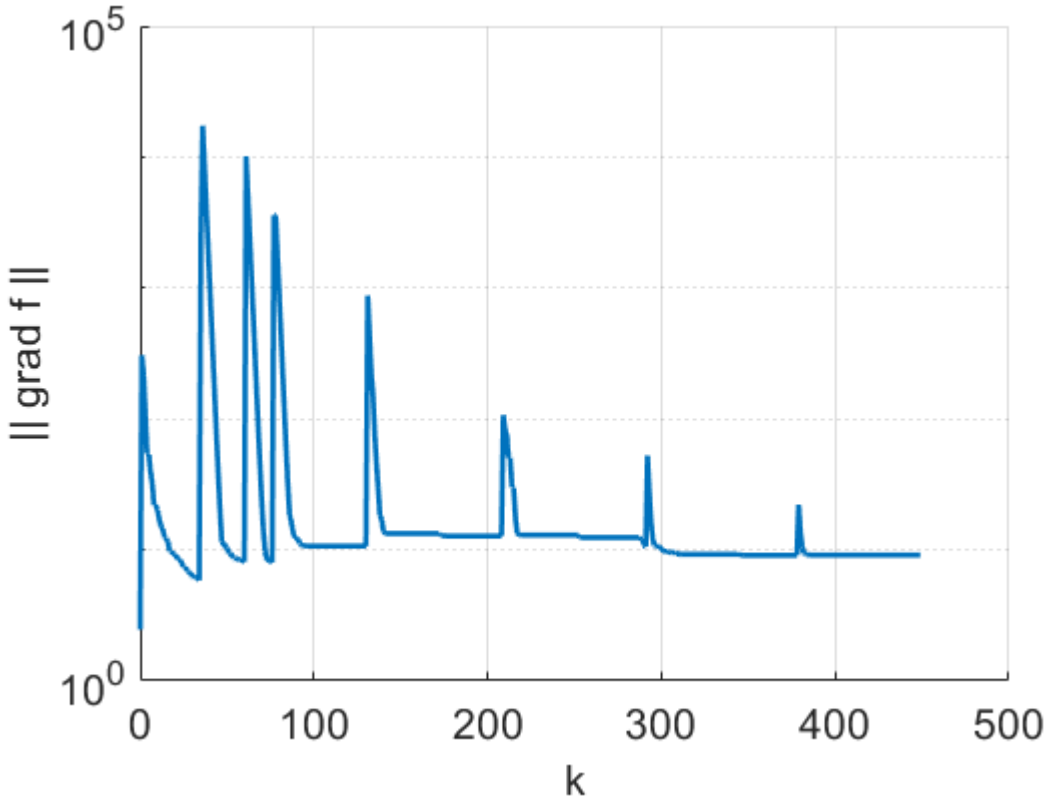

Figure 4: Norms of stochastic gradients at each iteration (x axis) averaged over 100 trials.

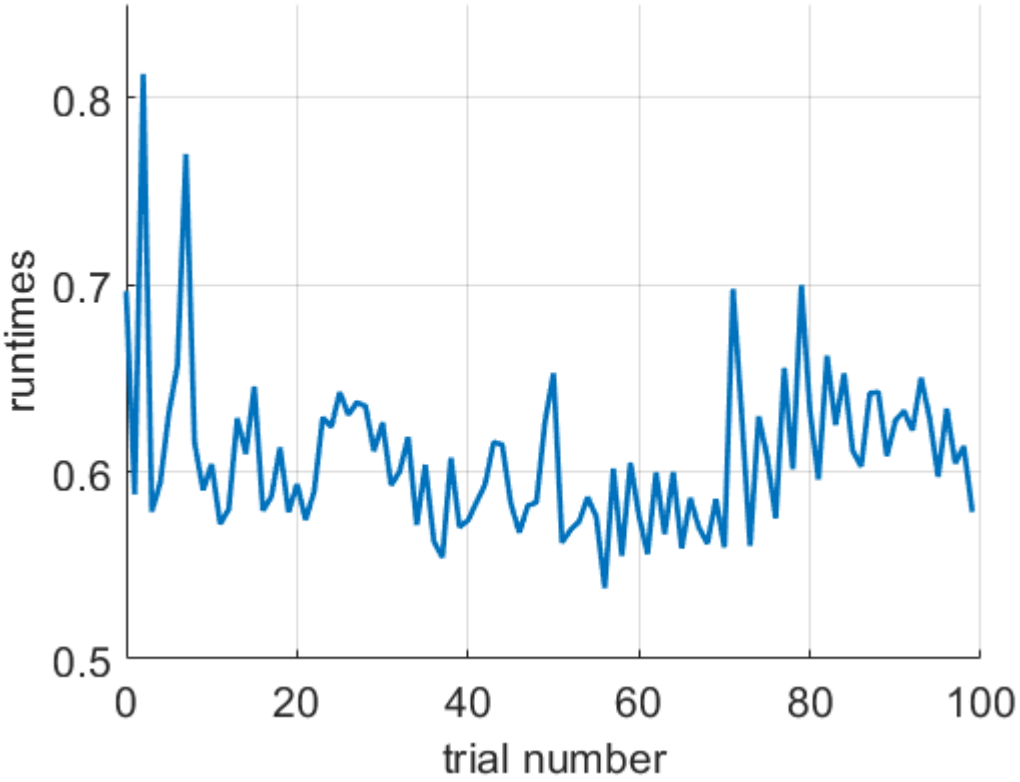

Figure 5: ASM runtimes for each trial of the averaging process.

2. **SG.**

Solution. Implementing the SG algorithm in a python codebase was quite intuitive and fluid. In total, the code took 32 lines. As input, the function took parameters: X, y, step_size_scheduler, regularization, and batch_size. X, y, batch_size, and regularization follow directly from the definition of the loss function in the problem statement. Step_size_scheduler is a 1D array containing the step size at each iteration; following, the length of this array was the total iterations of SG.

For experiments, the following parameters were used:

- Batch Size: 64, 128, 256, 512, 1024
- Scheduler: (See Figure 6 Below)
- Regularization: .001, .005, .01, .05, .1, .5, 1

```
{1:.1 / 2 ** (np.floor(np.log2(np.arange(1,512+1)))),
 2:.01 / 2 ** (np.floor(np.log2(np.arange(1,512+1)))),
 3:.1 / 1.5 ** (np.floor(np.log(np.arange(1,512+1))/np.log(1.5))),
 4:.01 / 1.5 ** (np.floor(np.log(np.arange(1,512+1))/np.log(1.5))),
 5:.1 / 3 ** (np.floor(np.log(np.arange(1,512+1))/np.log(3))),
 6:.01 / 3 ** (np.floor(np.log(np.arange(1,512+1))/np.log(3))),
 7:.1/np.arange(1,512),
 8:.01/np.arange(1,512+1),}
```

Figure 6: Rules for the 8 tested step_size schedulers.

Batch Size Results:

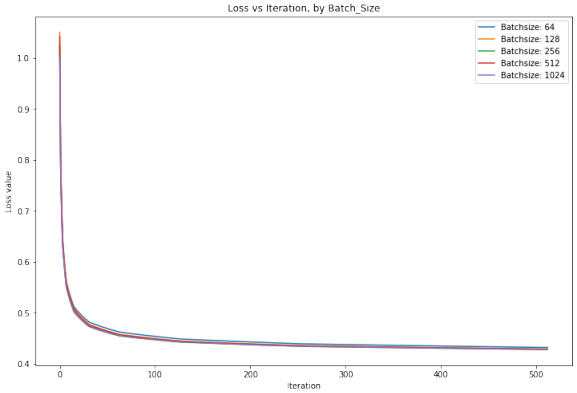

Figure 7: Loss vs. Iteration for different Batch Sizes

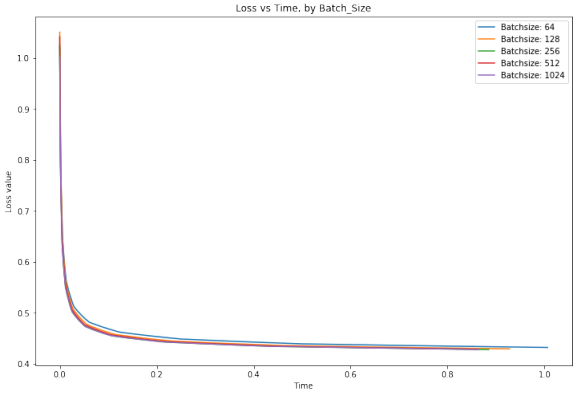

Figure 8: Loss vs. Time for different Batch Sizes

Above shows the loss v iter and loss v time graphs compared against the different batch sizes. We see on left that the loss itself did not depend much on the batch size, when looking at iteration; all of the lines were almost on top of each other. However, when this is broken down by time, it is apparent that the larger batch sizes win out, and create the same loss curve in a faster time. This agrees with standard deep learning state-of-the-art practices, which advise to use as large a batch size that can fit on your GPU.

Stepsize Strategy Results:

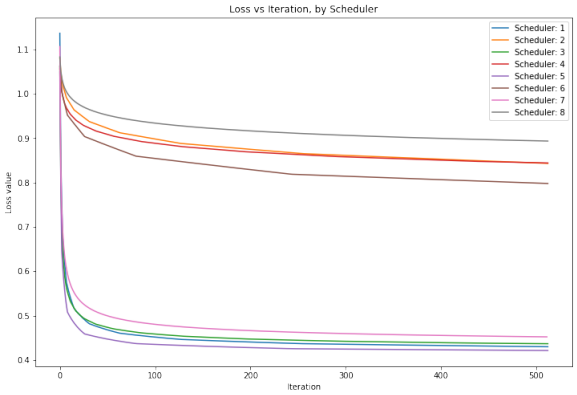

Figure 9: Loss vs. Iteration for different Schedulers

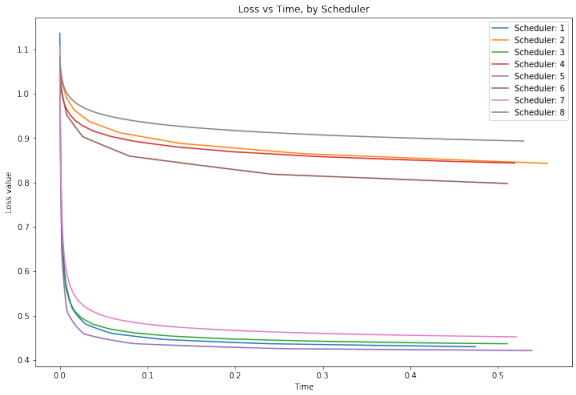

Figure 10: Loss vs. Time for different Schedulers

Above shows the loss versus iteration and loss versus time graphs compared against the different schedulers. The legend displays the scheduler number, which can be referenced in (Figure 6). As discussed in class, all of these schedulers are decreasing, with some decreasing exponentially and others doing so harmonically. This analysis gave quite interesting results. First and foremost, the schedulers that achieved the lower range of losses (bottom 4 curves) were numbers 1, 3, 5, and 7. These all start with a learning rate of .1, as opposed to the even-numbered schedulers which start with a learning rate of .01. This implies that the initial scale of learning rate greatly impacts overall learning: i.e. a .01 start is extremely low. Between the odd-numbered schedulers, 5 did best, followed by 1, 3, and 7. Again, of interest is that these are organized by gradient: that is, scheduler 5 decreased the most (exponential by factor 1/3), then scheduler 1 (exp by factor 1/2), etc. Thus, it seems that it is important to start with a high learning rate, then quickly decrease. Indeed, Scheduler 5 ended up with a smaller LR than Scheduler 2, 4, or 8, but achieved better loss due to this phenomenon. Intuitively, what is happening is that the model quickly converges to the neighborhood of the solution, but then relies on extremely small steps to get closer to the most optimal point.

Regularization Results:

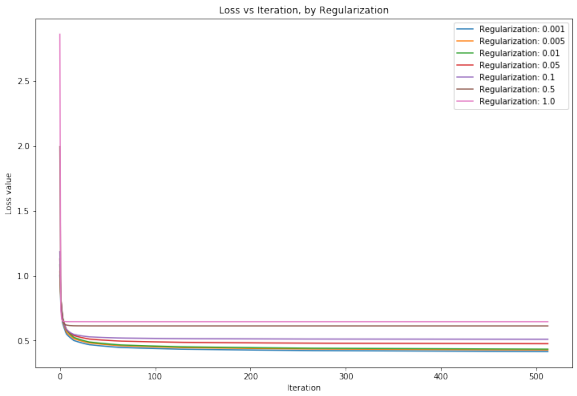

Figure 11: Loss vs. Iteration for different Lambdas

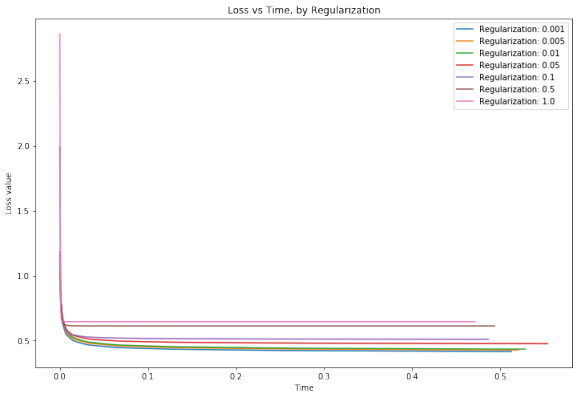

Figure 12: Loss vs. Time for different Lambdas

There is some difficulties in comparing different regularization schemes, as at heart these are different problems with different objectives. Obviously, the larger lambda is, the larger the loss (as these are linearly related). What we can say is that larger lambdas generally decrease runtime. This relationship isn't quite exact, but the larger lambdas would converge to a value much quicker than the smaller ones. This is probably due to the fact that larger lambdas would cause the L2 regularization loss to dominate the actual classification loss. As such, with a larger lambda, the model would be okay with having more classification loss, leading to earlier convergence (as classification loss is harder to decrease than regularization loss).

3. **Subsampled Inexact Newton.** Experiment with various batch sizes for subsampled inexact Newton's method. Plot average function value vs iteration number vs runtimes. Compare these plots with those for SG. Comment on how these two approaches compare.

Solution.

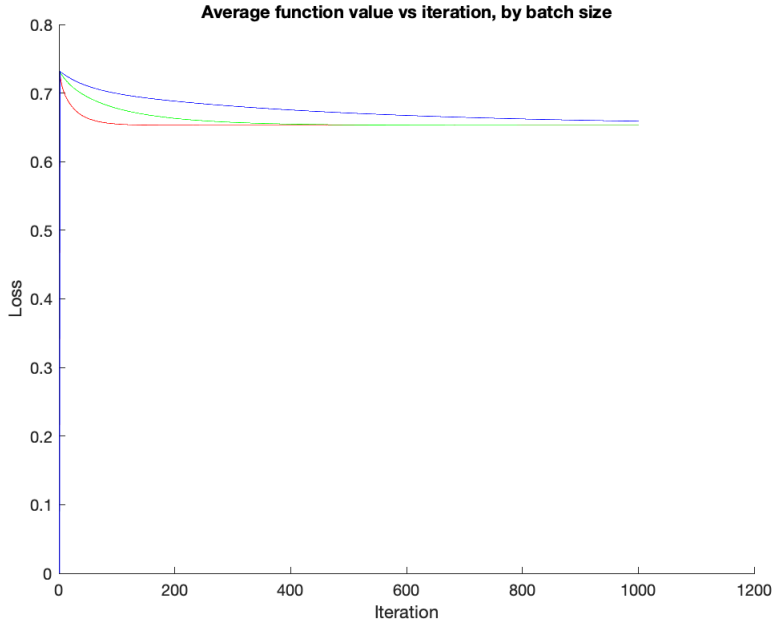

Figure 11: Loss vs. Iteration for different Batch sizes (red:64, green:256, blue:1024)

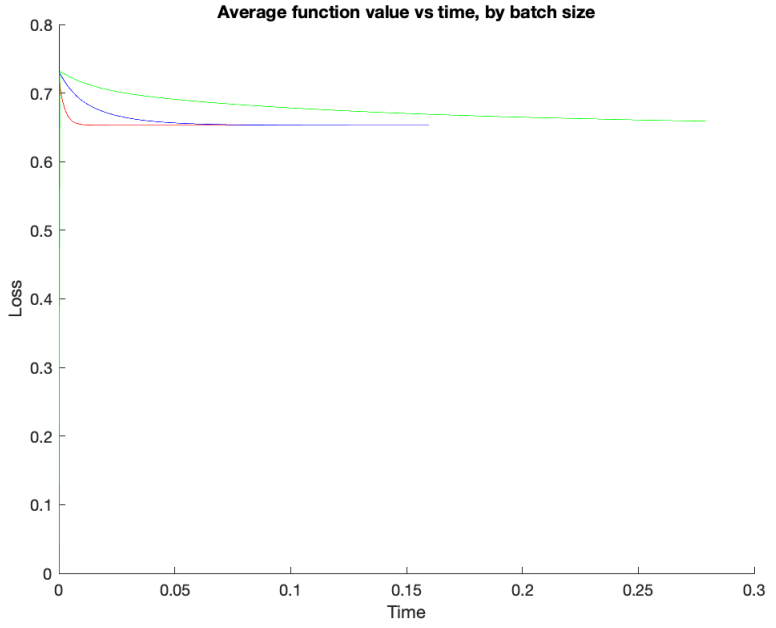

Figure 12: Loss vs. Time for different Batch sizes (red:64, green:256, blue:1024)

In this problem, there were a couple different trends that followed batch size. We see that, in general, lower batch sizes converged faster in both iteration number and real time; this was heavily noticed on our end, when we saw that each iteration for b=256 and b=1024 took *significantly* longer than the b=64 iterations. Unfortunately, as this code was written in MATLAB and the Q2 code was written in Python, it is difficult to make direct comparisons between

these plots. We do see that MATLAB is much faster for mathematical computation, as similar runs took less than a tenth of the time in MATLAB than in Python. Of course, some of this difference is caused by implementation-specific issues, but in general MATLAB performed better than Python. By iteration number, both of these methods took roughly the same amount (differing, of course, by batch size): most of the runs would converge to a final value around epoch 200 to 300.

Something that we can compare between the two setups is the loss values themselves. Indeed, we see that the SG runs achieved a minimum of about 0.45 loss, whereas Subsampled Inexact Newton converged at a much higher 0.67 value. This cannot be caused by language differences, as at heart these values are purely mathematical. This implies that though the SG algorithm might have a slower convergence rate, for this problem it converges to a better minima.

4. **Stochastic L-BFGS.** Program the stochastic L-BFGS method. Experiment with choosing stepsize, batch sizes for the gradient the pairs (s, y), and the frequency of updating the pairs. Fix $m = 5$. Compare its performance with stochastic Newton and SG.

Solution. We did not get stochastic L-BFGS to fully work. We were able to get L-BFGS to work and we successfully introduced stochasticity into the Hessian update every M iterations, but every time we tried to add a stochastic gradient update the method failed to converge. Figure 15 shows our results when we got the method working. In these scenarios, the step size α would begin positive but within 10 iterations collapsed to nearly 0 (precisely, 10^{-15} or so) and the gradient norm stopped reducing, leading to an infinite loop. This behavior was independent of the stepsize reduction schedule. We tried the following methods, none of which yielded convergent behavior when coupled with the stochastic gradient:

- $\alpha_{k+1} = 1/(k+1)$ (inverse step size reduction)
- $\alpha_{k+1} = \gamma\alpha_k$, $\gamma \in (0, 1)$ (geometric step size reduction, usually taking $\gamma = 0.9$)
- α_{k+1} (reduction defined by backtracking line search).

See Figure 17 for an example of updating the step size with the inverse reduction. For the rest of the analysis, we fixed the step size reduction to be defined by backtracking line search. The geometric updates gave almost identical behavior as the backtracking line search updates, while the inverse reduction updates were significantly more variable and often much slower than both.

The stochastic update for the pairs (s, y) took two forms:

1. $y_k = \nabla_{S_k^H} f(x_{k+1}, \xi_k^H) - \nabla_{S_k^H} f(x_k, \xi_k^H)$
2. $y_k = \nabla_{S_k^H} f(x_{k+1}, \xi_k^H) - \nabla_{S_k^H} f(x_{k-M}, \xi_k^H)$.

Most commonly, we used (1) as we found that (2) usually did not converge (exhibiting the same irregular behavior as mentioned before); however, when it did converge, (2) yielded the same dividing plane as (1) (Figure 16).

As we increased the delay between iterations for the Hessian update, we actually found that the norms of the gradients decreased slower and runtimes increased (Figure 19), leading us to conclude the optimal choice for M was 1 (update the (s, y) pairs on every iteration).

We naturally couldn't do numerical experiments for n_g (in our method, we used the full gradient—$n_g = n$), the batch size for the stochastic gradient, but we were able to do numerical experiments for n_h, the batch size for the stochastic Hessian (Figure 20). While increasing this batch size did not significantly affect the convergence, it did smoothen the decay of the gradient's norm. This behavior makes sense: as we use more and more components of the gradient for the update, we will naturally have a better approximation and see less variability in the method.

Surprisingly, even though we didn't implement the code correctly, our method performed generally better than SG and Subsampled Inexact Newton, converging in roughly 20 iterations (Figure 19) with optimal parameters—when it converged. Since we fixed the same tolerance for the gradient norm for the three methods, we're using the same standard to compare them all. As a result, we can conclude that our version of stochastic L-BFGS is a riskier method to use and should absolutely be implemented with a cap on the number of iterations. If the runtime exceeds this number of iterations (which happens very quickly), the user should switch to SG or Subsampled Inexact Newton.

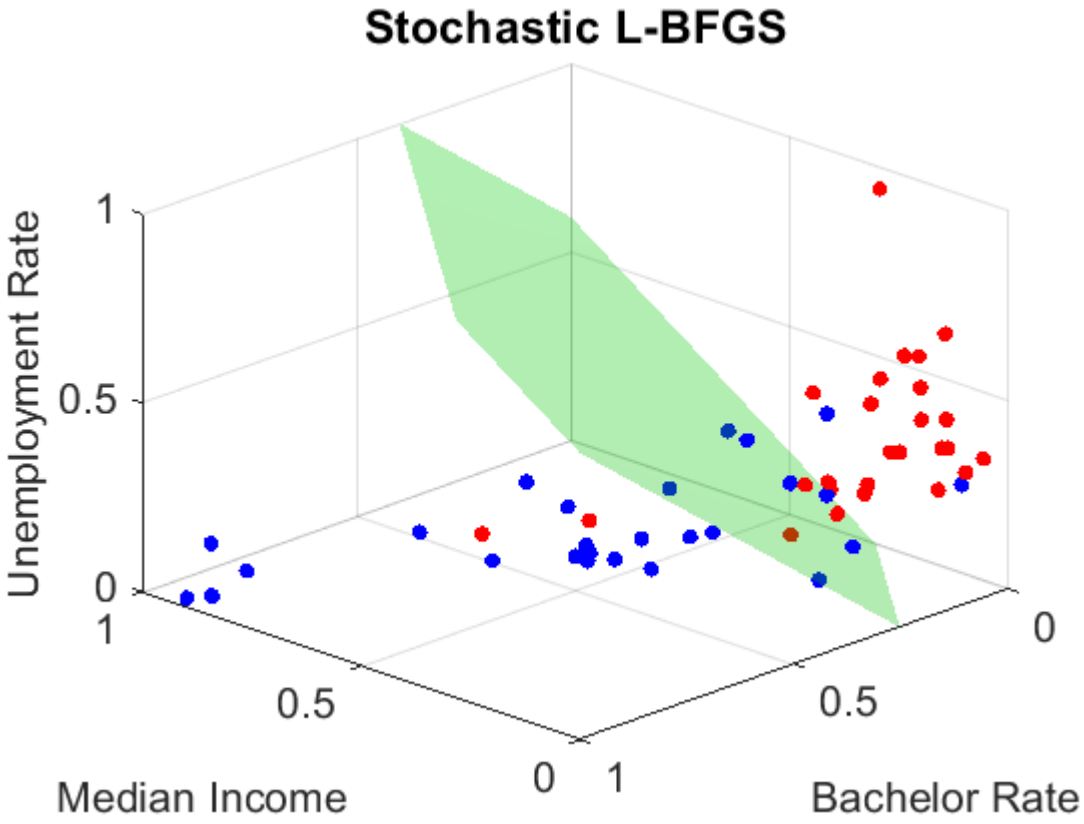

Figure 15: L-BFGS with stochastic Hessian update using previous x iterate, but not stochastic gradient update.

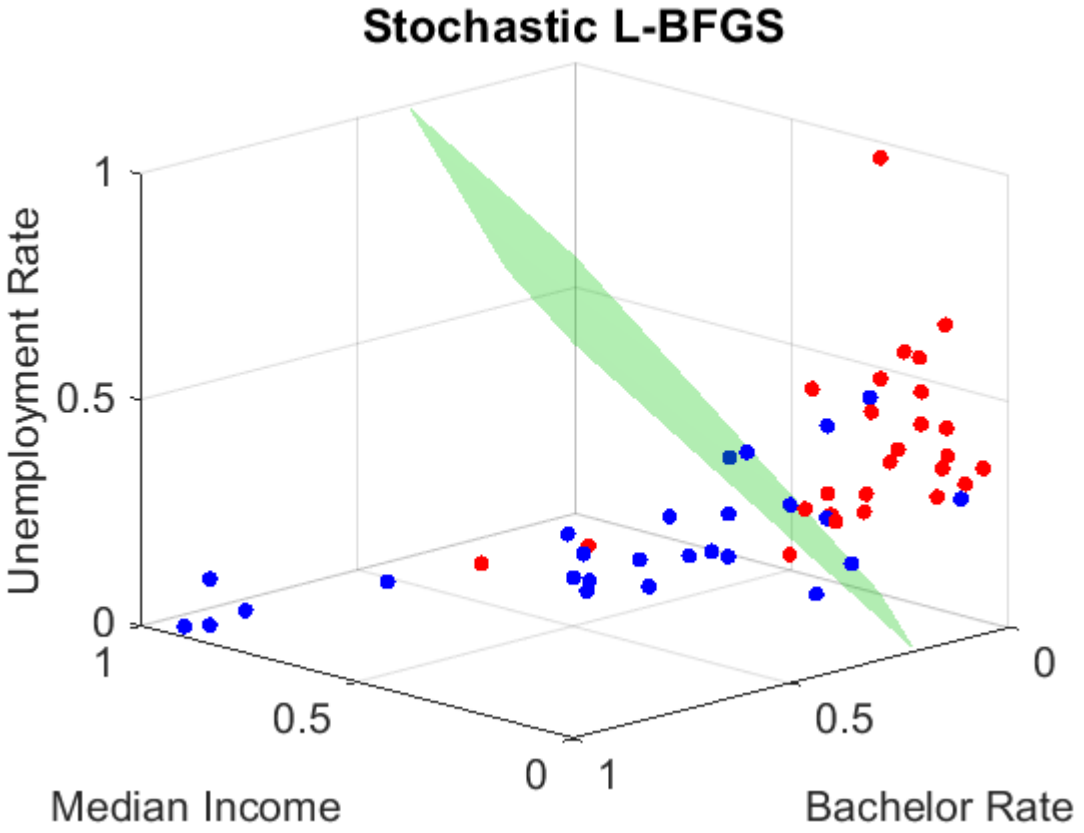

Figure 16: L-BFGS with stochastic Hessian update using Mth previous x iterate, but not stochastic gradient update. Note that this is virtually identical to the previous figure.

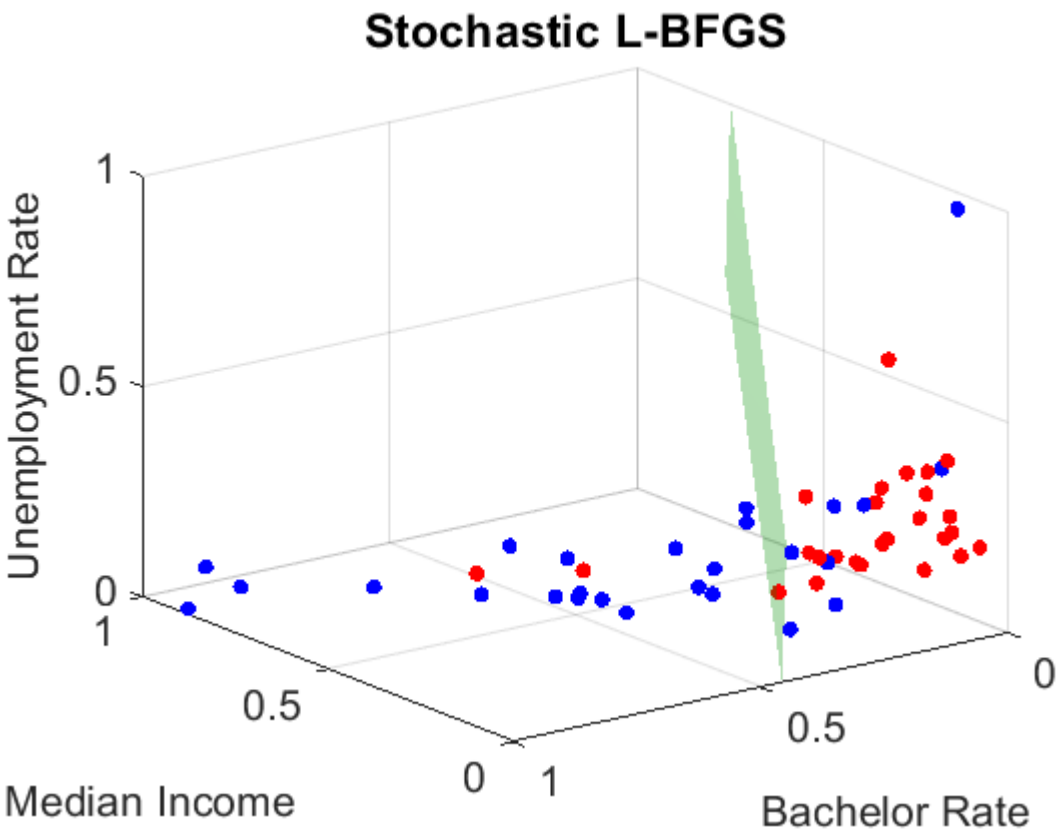

Figure 17: L-BFGS with stochastic Hessian update using the rule $\alpha_{k+1} = 1/(k+1)$ for the step size update. This particular run took 167 iterations, while runs with backtracking line search usually took around 25 iterations.

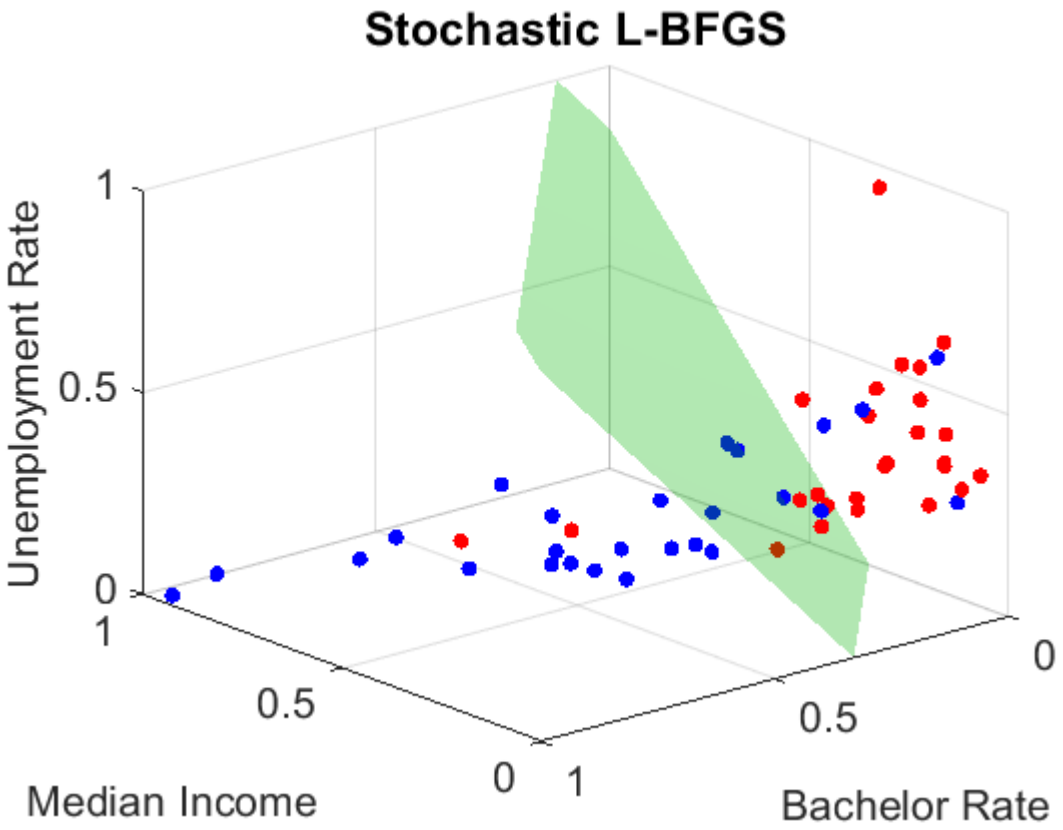

Figure 18: L-BFGS with stochastic Hessian update using the rule $\alpha_{k+1} = 0.9\alpha_k$ for the step size update.

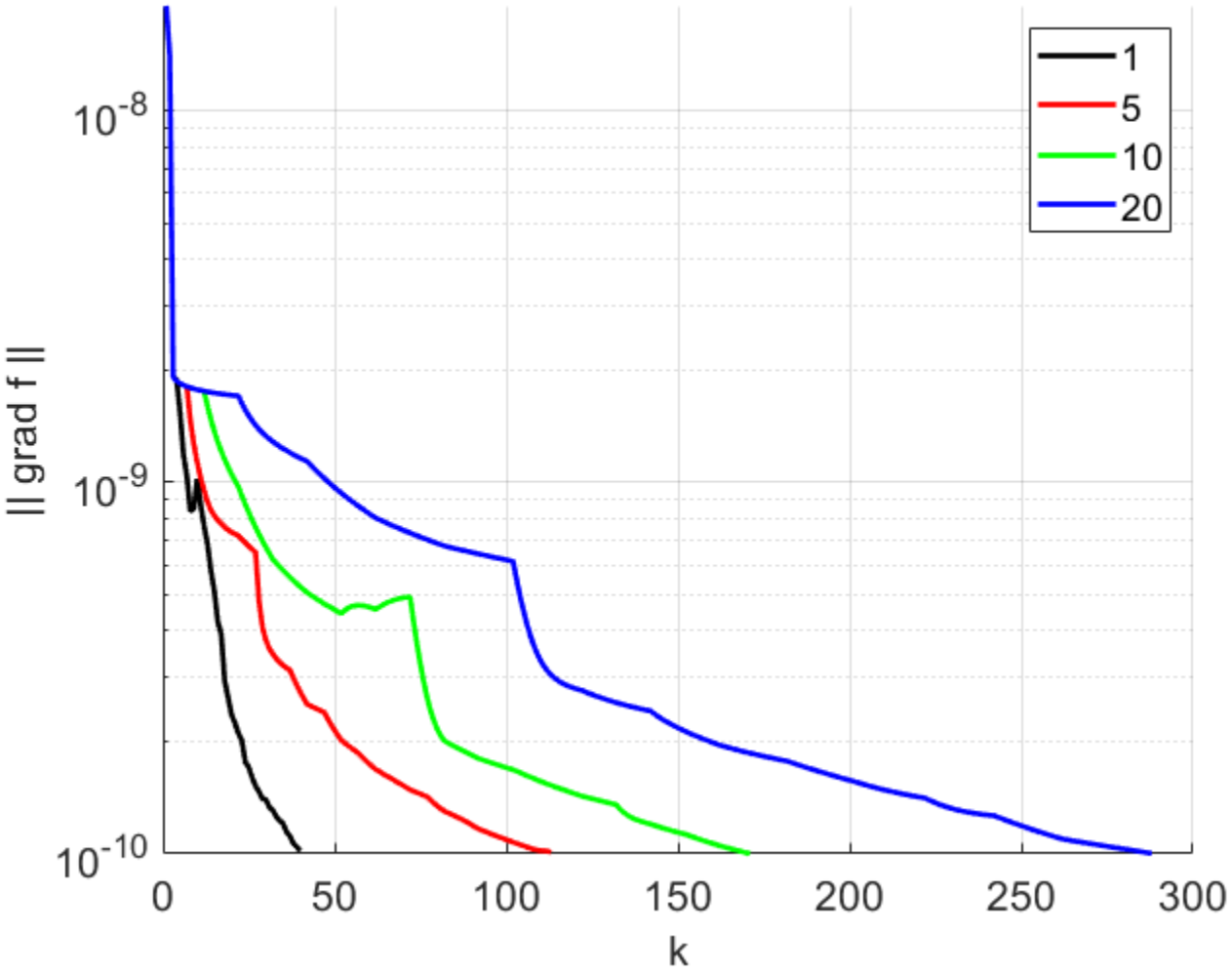

Figure 19: L-BFGS with stochastic Hessian update with various delays between iterations $M = 1, 5, 10, 20$.

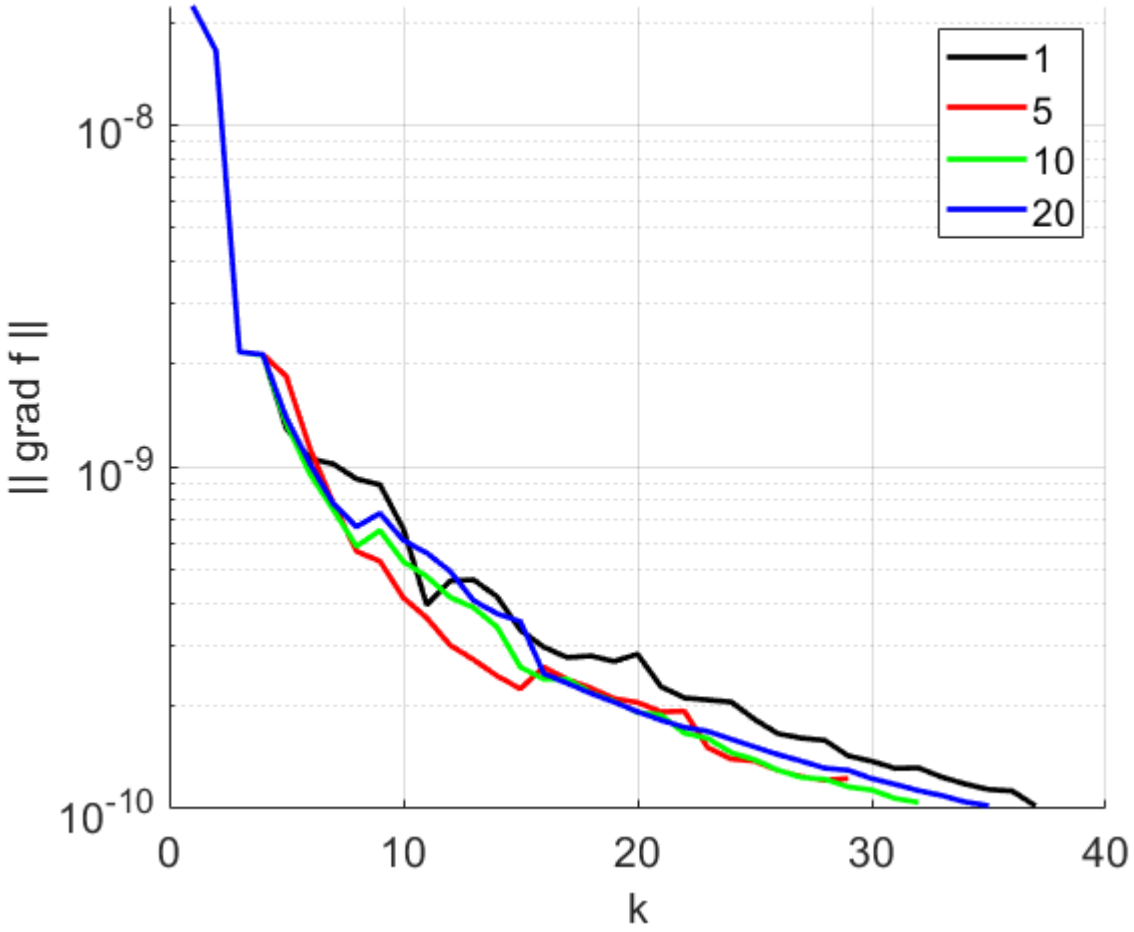

Figure 20: L-BFGS with stochastic Hessian update for various stochastic Hessian batch sizes $n_h = 1, 5, 10, 20$.

5. **Extra Analysis.** I thought it would be interesting to see what would happen if we broke the necessity for human visualization, a.k.a we used more than three parameters in our predictions. This first started with using all the parameters for a given year to predict the majority vote of that year, but then extended as follows. We create the new problem: *"Given all the election data from 2012 alongside the predictor variables for 2016, can we create a classifier for the 2016 election results"*.

Solution. Initially, we attempted to use an SVM classifier as an initial model for this data. Unfortunately, but as expected, this did not go far; the data has high dimensionality and is not *close* to being linearly separable. As such, probing from our ML background, we designed a simple two-layer model for this problem. This was trained using (non-Stochastic) Gradient Descent with cross-entropy loss and no L2 regularization. For the dataset, each feature was normalized by a standard batch norm procedure. An 80-20 data split was used for train-val, with an even amount of republican and democratic counties were used during training. The hidden layer was tested with various sizes, but ended up with 10 neurons, amounting to a total network size of 181 parameters. The detailed code for the NN is provided below:

```
def two_layer(X, y, m, step_size_scheduler=None):
    X = (X-X.mean(axis=0))/X.std(axis=0).copy()

    n = X.shape[0]
    d = X.shape[1]
    w = {}
    w['w1'] = np.random.randn(d,m)
    w['b1'] = np.random.randn(m)
    w['w2'] = np.random.randn(m)
    w['b2'] = np.random.randn()

    num_params = w['w1'].size + w['b1'].size + w['w2'].size + 1

    if step_size_scheduler is None:
        step_size_scheduler = .1 / 2 ** (np.floor(np.log2(np.arange(1,1024+1))))

    eps_min = 1e-12
    def l(w):
        a1 = X @ w['w1'] + w['b1']
        z1 = a1 * (a1 > 0)
        a2 = z1 @ w['w2'] + w['b2']
        z2 = 1/(1 + np.exp(-a2))

        z2[z2<eps_min] = eps_min
        z2[z2>1-eps_min] = 1-eps_min
        loss = -1/n * np.sum(y*np.log(z2) + (1-y)*np.log(1-z2))
        return loss, a1, z1, a2, z2

    losses = np.zeros(len(step_size_scheduler) + 1)
    losses[0], a1, z1, a2, z2 = l(w)

    for i in range(len(step_size_scheduler)):
        dw = {}
        dlda2 = 1/n * (z2 - y)
        dw['b2'] = dlda2.mean()
        dw['w2'] = dlda2 @ z1

        dlda1 = np.outer(dlda2,w['w2']) * (a1> 0)
        dw['b1'] = dlda1.mean()
        dw['w1'] = X.T @ dlda1

        step_size = step_size_scheduler[i]
        for key in w:
            w[key] -= step_size * dw[key]

        losses[i+1], a1, z1, a2, z2 = l(w)
    return w, losses, num_params

def predict(X, w):
    a1 = X @ w['w1'] + w['b1']
    z1 = a1 * (a1 > 0)
    a2 = z1 @ w['w2'] + w['b2']
    z2 = 1/(1 + np.exp(-a2))
    return 1.0 * (z2 > 0.5)

def accuracy(y,y_pred):
    acc = pd.DataFrame(columns=["Truth","Predicted","Count"])
    acc = acc.append({"Truth":1,"Predicted":1,"Count":sum(y[y_pred==1]==1)},ignore_index=True)
    acc = acc.append({"Truth":1,"Predicted":0,"Count":sum(y[y_pred==1]==0)},ignore_index=True)
    acc = acc.append({"Truth":0,"Predicted":1,"Count":sum(y[y_pred==0]==1)},ignore_index=True)
    acc = acc.append({"Truth":0,"Predicted":0,"Count":sum(y[y_pred==0]==0)},ignore_index=True)
    return acc
```

Figure 21: Code for Neural Network setup and training.

Train Results:

	Truth	Predicted	Count
0	1	1	205
1	1	0	63
2	0	1	165
3	0	0	307

Test Results:

	Truth	Predicted	Count
0	1	1	302
1	1	0	20
2	0	1	204
3	0	0	96

Figure 22: Accuracy of NN after training on train and test set. Republican = 1, Democrat = 0

Unfortunately, the predictive results of this experiment weren't too impressive (Figure 22). The model was able to correctly classify 69% of the train data, but generalized even worse. During test time, the model predicted republican for nearly all of the counties, and only achieved a 63% test time accuracy. One idea behind this would be that the model highly preferred the results of the 2012 election, and directly copied those to the 2016 results; this would be a very hard local minima to get out of. However, more analysis would need to be done to figure out the exact issue here.