1 Exercise2

1.1 (B)

Show that the random vector $ng_i(\beta)$ is an unbiased estimate of $\nabla l(\beta)$:

$$\mathbf{E}\{ng_i(\beta)\} = \nabla l(\beta)$$

Solution:

Since

$$\mathbf{E}\{g_i(\beta)\} = \sum_{j=1}^{j=N} Pr(i=j) * g_j(\beta) = \frac{1}{N} * \sum_{j=1}^{j=N} g_j(\beta) = \frac{1}{N} * \nabla l(\beta)$$

if we multiply N on both sides, we get $\mathbf{E}\{ng_i(\beta)\} = \nabla l(\beta)$.

1.2 (C)

In this question I used $N = 2000$, $P = 50$, $max\_iteration = 20000$. I first performed SGD with a 0.01 stepsize on some simulated data and plotted out ground-truth beta versus the estimated beta.

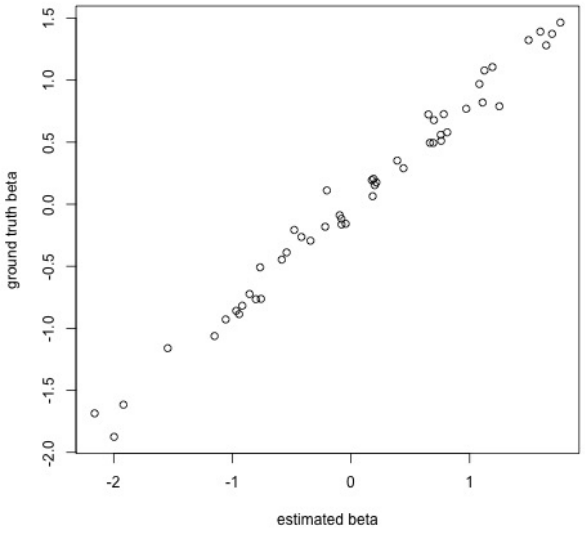


Figure 1: Beta

As the result looks reasonable, I moved on to try a smaller step size 0.001. Below are the plotted negative log likelihood throughout the 20000 iterations. The red one is computed using all samples, the green one is the running average and the blue one is an exponentially weighted average with weight on last sample = 0.01.

(a) stepsize = 0.01 (b) stepsize = 0.001

Figure 2: Negative log likelihood

I learned from the experiment that small step size leads to very stable decrease, whereas large step size makes it decrease faster. Also, as can be seen from the plot on the right, it seems that the running average is affected by high negative log likelihood at the beginning. The exponentially weighted average oscillates even when I use a small weight for the new sample(0.01).

1.3 (D)

In this question I tested different values of C and alpha. $t_0 = 1, C = 1, alpha = 0.6$ worked the best for me. Below is a comparison of Robbins-monro SGD with C=1 and alpha=0.6 against standard SGD.

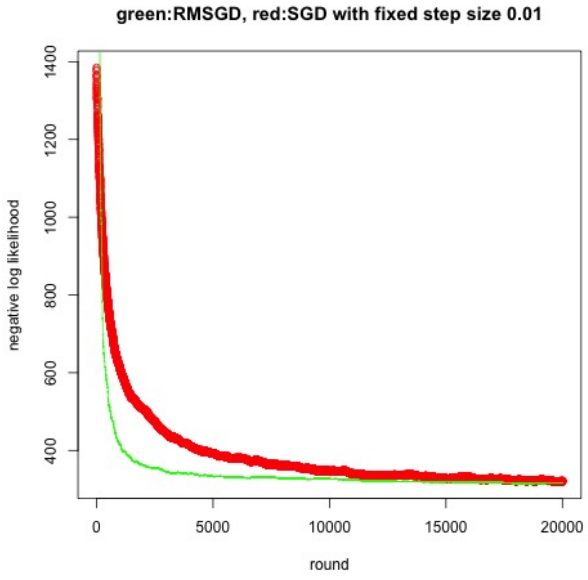


Figure 3: Negative log likelihood

It's obvious that Robbins-monro SGD decreases much faster than standard SGD.

1.4 (E)

I followed the RM decreasing stepsize from (D) and set a burn-in period of 10000 iterations. Below is the plot of negative log likelihood of SGD with and without PR averaging. Although they share the same learning process, not only does the PR averaging SGD decreases more smoothly, the PR averaging also helps to reduce the negative log likelihood a little bit.

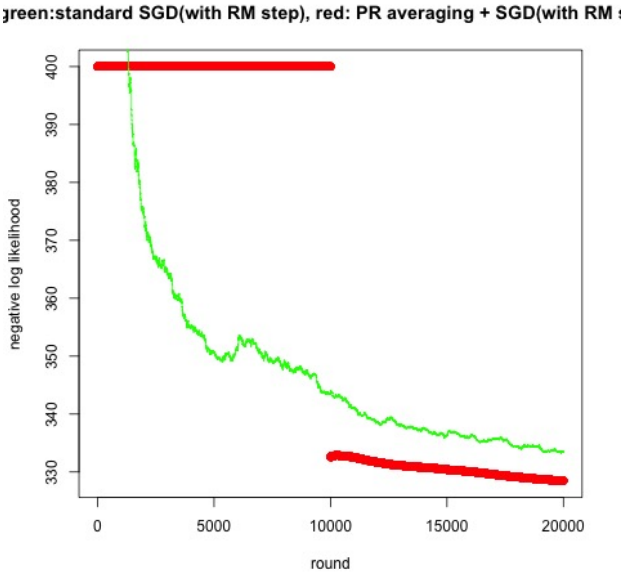


Figure 4: Negative log likelihood

2 Exercise 3

2.1 (Line Search, Exercise3.1.r)

2.1.1 (A)

The wolfe conditions are summarized by two equations:

condition(1):

$$f(\beta_k + \alpha * p_k <= f(\beta) + c_1 * \alpha * {p_k}^T * f(\beta_k)$$

condition(2):

$$\nabla f(\beta_k + \alpha * p_k)^T * p_k >= c_2 * \nabla f(\beta_k)^T * p_k$$

where where $0 < c_1 < 1$ and $c_1 < c_2 < 1$

The first condition guarantees that the chosen step size α leads to a sufficient decrease of the objective function, whereas the second condition forces a step size that's not too small, as a tiny step size will always satisfy condition(1).

However, in practice, instead of testing condition(2), we could keep decreasing alpha gradually until the first time condition(1) is satisfied, such that α cannot be trivially small.
Hence the algorithm is:

- initialize α, pick ρ and c in range (0,1)
- repeat until condition(1) is first satisfied:
  $\alpha = \alpha * \rho$
- use α as the step size

2.1.2 (B)

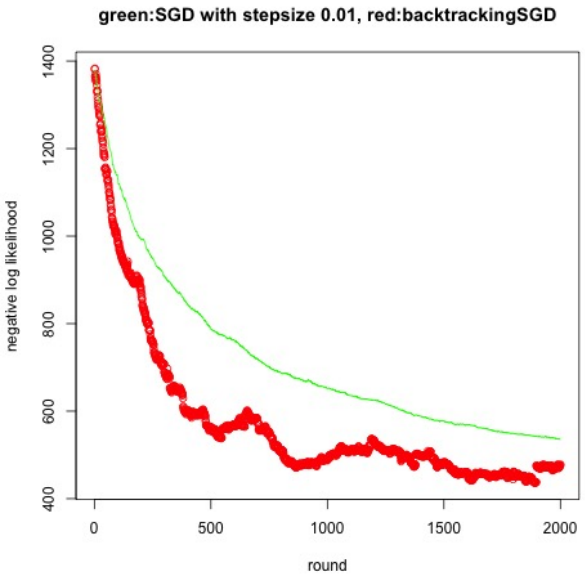


Figure 5: Negative log likelihood

As can be seen, compared to using a fixed step size, backtracking line search fits much faster.

2.2 (Quasi-Newton, Exercise3.2.r)

2.2.1 (A)

Secont condition:

$$B_{k+1} * s_k = y_k$$

where $s_k = x_{k+1} - x_k$ and $y_k = \nabla l_{k+1} - \nabla l_k$

The derivation of this condition can be found on page 24 of the textbook. It's derived from the second-order Taylor approximation of the derivative. In practice we update the inverse of B_{k+1} instead for better efficiency. To implement this, I used the following procedure:

- 1. Initialize H_1 as the identity matrix, and β_1 as a 0s
- 2. For each iteration k:
  - compute $\nabla l(\beta_k)$ if not previously computed
  - use $-H_k * \nabla l(\beta_k)$ as the direction
  - apply backtracking line search to find the step size
  - $\beta_{k+1} = \beta_k + stepsize * direction$
  - $s_k = \beta_{k+1} - \beta_k$
  - compute $\nabla l(\beta_{k+1})$
  - $y_k = \nabla l_{k+1} - \nabla l_k$

– $H_{k+1} = (I - \rho_k * s_k * y_k^T) * H_k * (I - \rho_k * s_k * y_k^T)$, where $\rho_k = \frac{1}{y_k^T * s_k}$

2.2.2 (B)

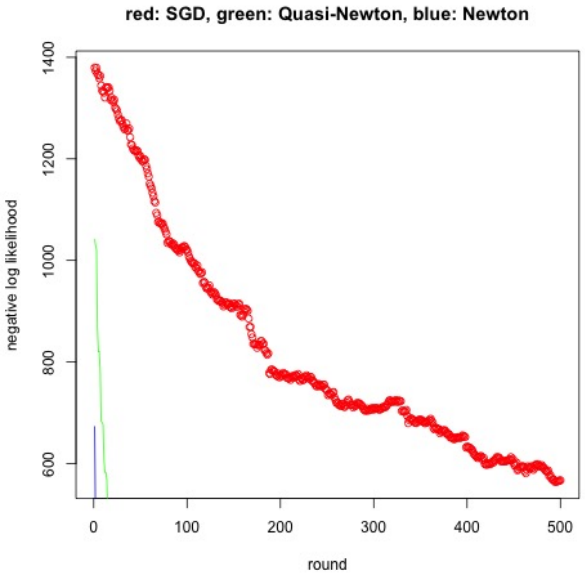


Figure 6: Negative log likelihood

We can see from this picture that both Newton's method and Quasi-newton's method converges faster than SGD. However, each single iteration of Newton's method and Quasi-newton's method takes much longer because they need to traverse the entire dataset. To show the comparison between Newton's method and Quasi-newton's method more clearly, a plot without SGD is shown below.

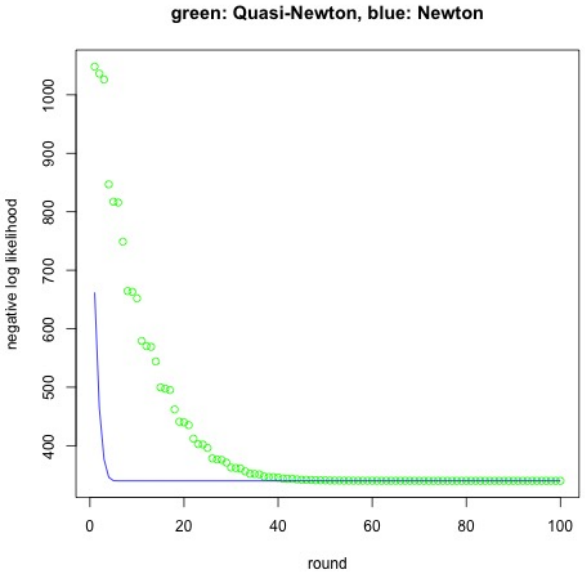


Figure 7: Negative log likelihood

This plot implied the trade-off of Quasi-newton's method, i.e. it's more computationally efficient but converges slightly slower than Newton's method.

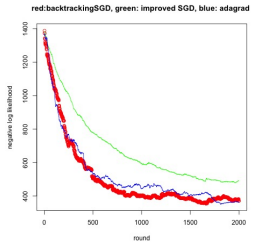
3 Exercise 4

3.1 Improved backtracking SGD and Adagrad, Exercise 4.1.r

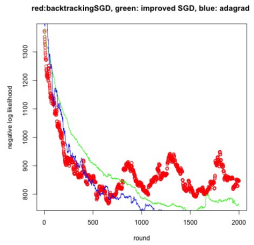
For section(A) I am not sure if I am doing correctly. If I follow exactly the instructions, I end up with huge step size and the network doesn't even converge, possibly because the step size for the average gradient is equivalent to the subsample size times the step size for the sum of gradient. Instead of doing so, what I did was to divide the step size for the sum of gradient by the subsample size, and I got decent decreasing curve in this manner.

For adagrad, I initialized the accumulated gradients as 0.001. This leads to a huge step when β_i is first updated, but it prevents the program from encountering NAN problems.

Below are the plot of negative log likelihood for a normal case and a sparse case. In the sparse case, only 20% of the entries in X is nonzero.



(a) non-sparse X



(b) sparsity = 0.8 in X

Figure 8: red: backtracking SGD, green: improved backtracking SGD in (A), blue: AdaGrad

As expected, Adagrad converges fast in the sparse case. Surprisingly the method from (A) also works pretty well in the sparse case.

3.2 Complete version, Exercise4.2.cpp

In this section I implemented a SGD with the following features:

- implementation in Cpp with Rcpp and Eigen
- L2 regularization of β and lazy update of the regularization
- sparsity support

With the regularization weight $\lambda = 0.0000005$, $stepsize = 0.005$ and running over three epochs, my training and testing prediction accuracy are 97.9% and 97.6% respectively. This took about 10 seconds after the data has been loaded.

```
> microbenchmark(mymain(X[,train],Y[train], X[,test],Y[test]), times = 1)
0.979474 0.9755
Unit: seconds
                                          expr      min       lq     mean   median       uq      max neval
 mymain(X[, train], Y[train], X[, test], Y[test]) 10.14429 10.14429 10.14429 10.14429 10.14429 10.14429     1
```

Figure 9: prediction accuracy and benchmark