Comparision of Optimization Methods in Machine Learning

Tanmay Gupta

12/4/2018

Introduction

In this paper, I will be evaluating a few optimization techniques that are used in the context of Machine Learning. In our Introduction to Optimization Class, we have covered the mathematical basis of L-BFGS, DFP, Newton Method, and the Conjugate Gradient method among a number of other techniques. It is a well known fact that these numerical techniques are widely used in the world of machine learning. In a world where data is becoming more abundant and cheaper by the day, the data rules which algorithm is best for a particular problem. In light of that, the prupose of this paper will be to evaluate the different techniques on the basis of the time it takes them to minimize a given loss function on 3 types of data the details of which I will be covering in the coming few pages. I will be evaluating the performance of the Binary Logistic Regression classifier, a model that is popular for its effectiveness in classifying data into categories.

Optimizers Evaluated

- **Conjugate Gradient (CG):** Conjugate Gradient which was implemented using scipy.optimize.fmin_cg. Scipy.optimize is a library in python that specializes in optimization of functions.
- **Truncated Newton (TNC):** Truncated Newton as implemented fmin_tnc function of the library.
- **BFGS:** Broyden–Fletcher–Goldfarb–Shanno method, as implemented in fmin_bfgs.
- **L-BFGS:** Limited-memory BFGS as implemented in fmin_l_bfgs_b.
- **Newton-CG (NCG):** Newton Conjugate Gradient as implemented in fmin_ncg.

Data Used

The data that I will be working with is a simulated data set, with 10000 rows and 1001 columns, including a constant term, as often used in regression. I will be preparing 3 seperate sets of data with varying correlation among the columns of data, ranging from near 0 to near 0.8 in some extreme cases. The idea is to evaluate the performance of different optimizers for different types of data that we can find in the real world.

The following code generates the data that we will be using for this paper.

```
n_samples, n_features = 10 ** 4, 10 ** 3
np.random.seed(0)
X = np.random.randn(n_samples, n_features)
w = np.random.randn(n_features)
y = np.sign(X.dot(w))
X += 0.8 * np.random.randn(n_samples, n_features) # add noise
X+= corr # this makes it correlated by adding a constant term
X = np.hstack((X, np.ones((X.shape[0], 1)))) # add a column of ones for intercept
```

Functions Used

The optimization code is primarily written in Python while the code to present the visualizations are written in R and document was created using R Markdown.

In the code below, we will be defining a few functions that we will be using for the purpose of Optimization. I have defined 3 functions:

Phi: To represent the logistic regression function. The function is the standard logistic regression function.

$$\phi(t) = \frac{1}{1 + e^{-t}}$$

```
def phi(t):
    # logistic regression function
    idx = t > 0
    out = np.empty(t.size, dtype=np.float)
    out[idx] = 1. / (1 + np.exp(-t[idx]))
    exp_t = np.exp(t[~idx])
    out[~idx] = exp_t / (1. + exp_t)
    return out
```

Loss: The purpose of any optimization algorithm is to minimize a certain function. In the context of machine learning, the purpose is then to minimize the loss function of a certain algorithm with respect to the coefficients of the model given some data. In our case, we will be minimizing the log-loss function, which takes the form:

$$\log(\phi(t)) = \begin{cases} -\log(1 + e^{-t}) & t > 0 \\ t - \log(t + e^{t}) & t \leq 0 \end{cases}$$

```
def loss(w, X, y, alpha):
    # loss function to be optimized. In our case, this will be the logistic loss function
    z = X.dot(w)
    yz = y * z
    idx = yz > 0
    out = np.zeros_like(yz)
    out[idx] = np.log(1 + np.exp(-yz[idx]))
    out[~idx] = (-yz[~idx] + np.log(1 + np.exp(yz[~idx])))
    out = out.sum() + .5 * alpha * w.dot(w)
    return out
```

Gradient: This function evaluates the gradient of the loss function and is used in nearly all the optimization techniques in this paper. This function, takes the form of:

$$\nabla \log(\phi(t)) = yX^T \cdot (\phi(y(X \cdot w)) - 1) + \alpha w$$

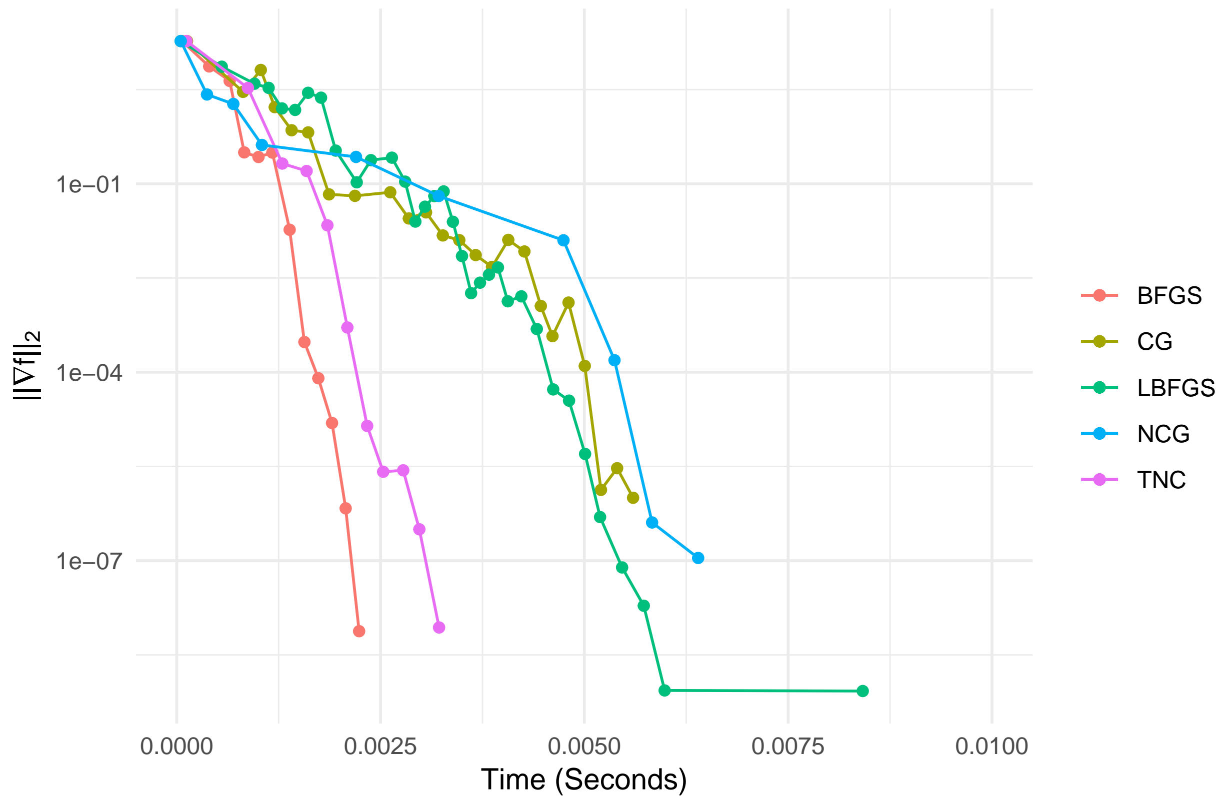
```
def gradient(w, X, y, alpha):
    # gradient of the loss function
    z = X.dot(w)
    z = phi(y * z)
    z0 = (z - 1) * y
    grad = X.T.dot(z0) + alpha * w
    return grad
```

Setting up Initial Experiment

I will be setting up an initial experiment with a 5×5 matrix with logistic regression to calibrate the optimizers and check if they are performing optimally.

```
CG
Optimization terminated successfully.
         Current function value: 1.704291
         Iterations: 23
         Function evaluations: 50
         Gradient evaluations: 48
```

```
L-BFGS
        Current Function Value: 1.70429083405
        Iternations:: 33
        Function Evaluations: 70
BFGS
Optimization terminated successfully.
         Current function value: 1.704291
         Iterations: 11
         Function evaluations: 13
         Gradient evaluations: 13
Newton-CG
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 1.704291
         Iterations: 9
         Function evaluations: 81
         Gradient evaluations: 129
         Hessian evaluations: 0
TNC
        Current Function Value: 1.70429083405
        Iterations: 11
        Function Evaluations: 66
```



Here, we can see, that as expected, BFGS algorithm gives us the best performance while the other optimizers aren't too far behind. We can see, however, that the L-BFGS algorithm does not perform as well in the sense that we can see it takes much longer for L-BFGS to reach the same level of ||
$nablaf||_2$ as compared to the other optimizers.

Case 1 - No Correlation within Data

In this specific case, we will be evaluating the performance of the optimizers when there is little to no correlation in the data.

```
corr = 0.
if corr == 0.:
    n_samples, n_features = 10 ** 4, 10 ** 3
    np.random.seed(0)
    X = np.random.randn(n_samples, n_features)
    w = np.random.randn(n_features)
    y = np.sign(X.dot(w))
    X += 0.8 * np.random.randn(n_samples, n_features) # add noise
    X+= corr # this makes it correlated by adding a constant term
    X = np.hstack((X, np.ones((X.shape[0], 1)))) # add a column of ones for intercept
    alpha = 1.

    # conjugate gradient
    print('CG')
    timings_cg = []
    precision_cg = []
    w0 = np.zeros(X.shape[1])
    start = datetime.now()
    def callback(x0):
        prec = linalg.norm(gradient(x0, X, y, alpha), 2)
        precision_cg.append(prec)
        timings_cg.append((datetime.now() - start).total_seconds())
    callback(w0)
    w = optimize.fmin_cg(loss, w0, fprime=gradient, args=(X, y, alpha), gtol=1e-6,
                         callback=callback, maxiter=200)

    # L-BFGS
    print('L-BFGS')
    timings_lbfgs = []
    precision_lbfgs = []
    w0 = np.zeros(X.shape[1])
    start = datetime.now()
    def callback(x0):
        prec = linalg.norm(gradient(x0, X, y, alpha), 2)
        precision_lbfgs.append(prec)
        timings_lbfgs.append((datetime.now() - start).total_seconds())
    callback(w0)
    out = optimize.fmin_l_bfgs_b(loss, w0, fprime=gradient, args=(X, y, alpha),
                                 pgtol=1e-10, maxiter=200, maxfun=250, factr=1e-30,
                                 callback=callback, disp = 98)
    print("\t\tCurrent Function Value: {x}".format(x = out[1]))
    print("\t\tIternations:: {x}".format(x = out[2]['nit']))
    print("\t\tFunction Evaluations: {x}".format(x = out[2]['funcalls']))

    # BFGS
    print('BFGS')
    timings_bfgs = []
    precision_bfgs = []
    w0 = np.zeros(X.shape[1])
```

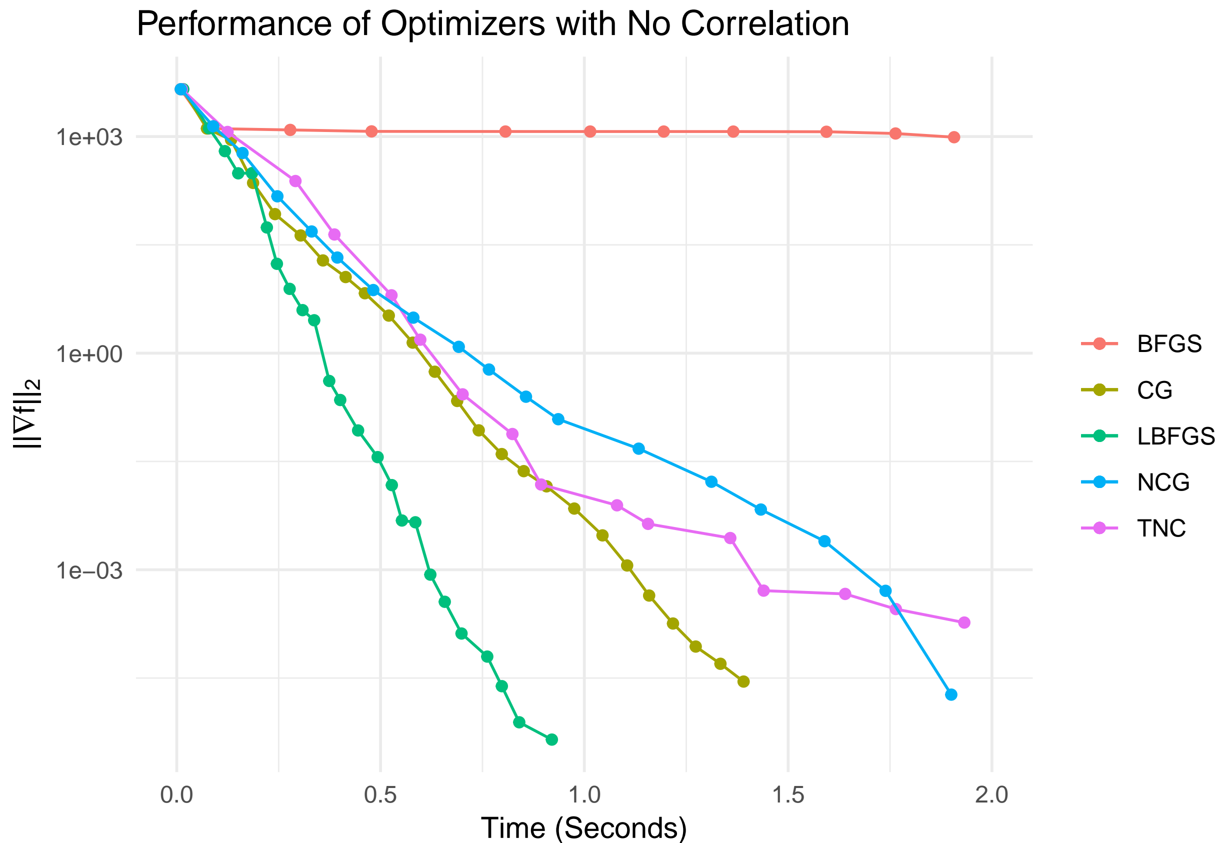
```python
    start = datetime.now()
    def callback(x0):
        prec = linalg.norm(gradient(x0, X, y, alpha), 2)
        precision_bfgs.append(prec)
        timings_bfgs.append((datetime.now() - start).total_seconds())
    callback(w0)
    out = optimize.fmin_bfgs(loss, w0, fprime=gradient, args=(X, y, alpha), gtol=1e-10,
                             maxiter=50, callback=callback)

    # Newton-CG
    print('Newton-CG')
    timings_ncg = []
    precision_ncg = []
    w0 = np.zeros(X.shape[1])
    start = datetime.now()
    def callback(x0):
        prec = linalg.norm(gradient(x0, X, y, alpha), 2)
        precision_ncg.append(prec)
        timings_ncg.append((datetime.now() - start).total_seconds())
    callback(w0)
    out = optimize.fmin_ncg(loss, w0, fprime=gradient, args=(X, y, alpha), avextol=1e-8,
                            callback=callback, maxiter=40)

    # Truncated Newton
    print('TNC')
    timings_tnc = []
    precision_tnc = []
    w0 = np.zeros(X.shape[1])
    start = datetime.now()
    def callback(x0):
        prec = linalg.norm(gradient(x0, X, y, alpha), 2)
        precision_tnc.append(prec)
        timings_tnc.append((datetime.now() - start).total_seconds())
    callback(w0)
    out = optimize.fmin_tnc(loss, w0, fprime=gradient, args=(X, y, alpha), callback=callback,
                            ftol = 1e-14, pgtol = 1e-17, xtol = 1e-10)
    print("\t\tCurrent Function Value: {x}".format(x = loss(out[0], X, y, alpha)))
    print("\t\tIterations: {x}".format(x = len(precision_tnc)))
    print("\t\tFunction Evaluations: {x}".format(x = out[1]))
```

```
CG
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 3848.590812
         Iterations: 24
         Function evaluations: 49
         Gradient evaluations: 48
L-BFGS
        Current Function Value: 3848.59081206
        Iternations:: 23
        Function Evaluations: 28
BFGS
Warning: Maximum number of iterations has been exceeded.
         Current function value: 3896.356013
         Iterations: 50
```

```
         Function evaluations: 76
         Gradient evaluations: 76
Newton-CG
Optimization terminated successfully.
         Current function value: 3848.590812
         Iterations: 17
         Function evaluations: 18
         Gradient evaluations: 96
         Hessian evaluations: 0
TNC
        Current Function Value: 3848.59081206
        Iterations: 19
        Function Evaluations: 79
```

The condition number for this system matrix is 12.1683226.

We can visualize the performance of the optimizers using R. The graph of their performance is given below, where I have plotted the norm of gradient, $||\nabla f||_2$ vs Time in Seconds taken to reach that value.



From the above plot, it is evident that L-BFGS method gives us the best and the fastest result for completely uncorrelated data. While other optimizers such as TNC, NCG, and CG also perform at a similar or comparable rate, we can see that BFGS gives us rather disappointing results. It was surprising to see the poor performance of BFGS when compared to L-BFGS, one would think that they should perfrom similarly but the poor performance may be due to the memory requirement for estimating the Hessian of a function with a 1001 variables as input.

Next, we will add some correlation to the data and see how the performance of these optimizers changes.

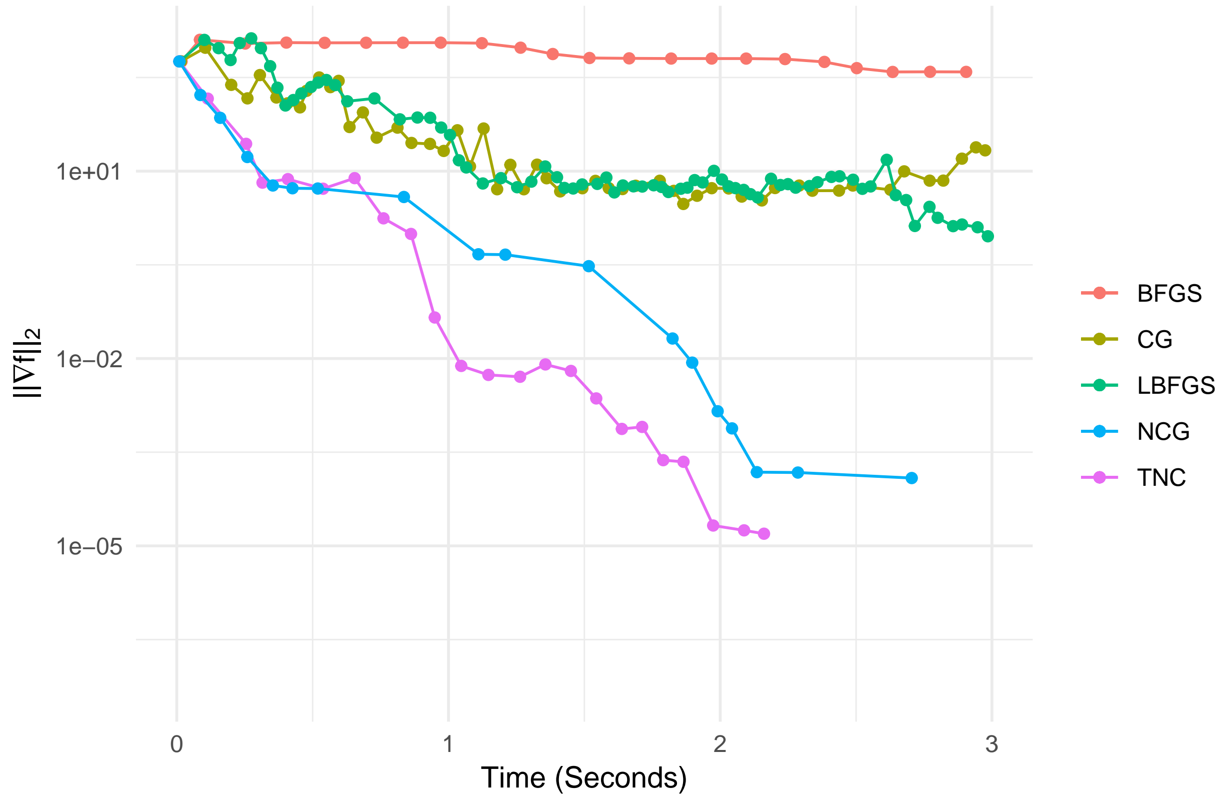
Case 2 - Little Correlation within Data

In this specific case, we will be evaluating the performance of the optimizers when there is little correlation in the data. Following are the results for the logistic regression using similar code as above.

```
CG
Warning: Maximum number of iterations has been exceeded.
         Current function value: 3851.737877
         Iterations: 200
         Function evaluations: 511
         Gradient evaluations: 511
L-BFGS
        Current Function Value: 3851.73722757
        Iternations:: 146
        Function Evaluations: 175
BFGS
Warning: Maximum number of iterations has been exceeded.
         Current function value: 3923.452454
         Iterations: 50
         Function evaluations: 85
         Gradient evaluations: 85
Newton-CG
Optimization terminated successfully.
         Current function value: 3851.737228
         Iterations: 20
         Function evaluations: 23
         Gradient evaluations: 420
         Hessian evaluations: 0
TNC
        Current Function Value: 3851.73722757
        Iterations: 23
        Function Evaluations: 110
```

The condition number for this system matrix is 47.4125047.

We can visualize the performance of the optimizers using R. The graph of their performance is given below, where I have plotted the norm of gradient, $||\nabla f||_2$ vs Time in Seconds taken to reach that value.



In this case, we can see that our top performers from the previous case have taken a back-seat and TNC and NCG have performed better than most other cases. BFGS continues to perfrom poorly while L-BFGS has proven slow in this regard as well.

In the last case that I will be analysing today, we will consider data that is heavily correlated within itself.

Case 3 - Heavy Correlation within Data

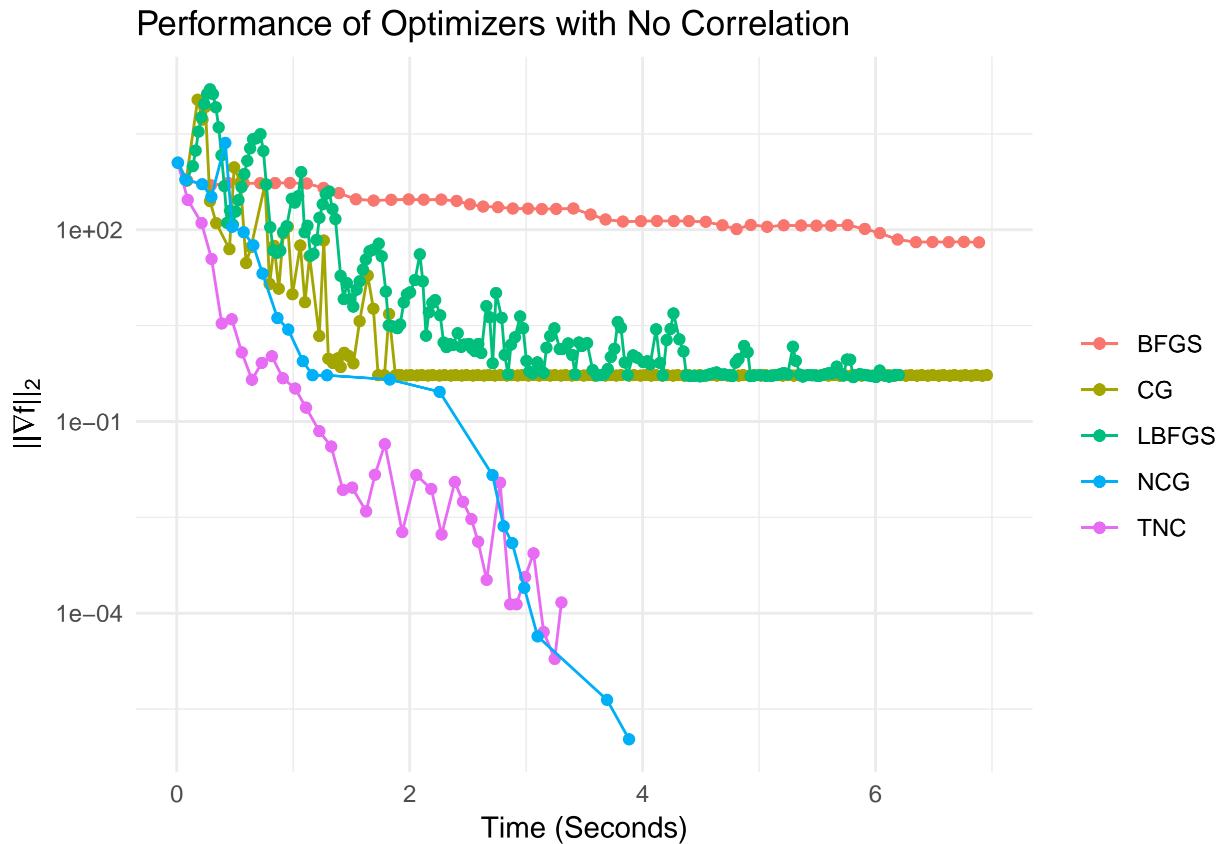
In this specific case, we will be evaluating the performance of the optimizers when there is heavy correlation in the data.

```
CG
Warning: Maximum number of iterations has been exceeded.
         Current function value: 3856.905869
         Iterations: 200
         Function evaluations: 506
         Gradient evaluations: 506
L-BFGS
        Current Function Value: 3856.90577325
        Iternations:: 200
        Function Evaluations: 236
BFGS
Warning: Maximum number of iterations has been exceeded.
         Current function value: 3900.789613
         Iterations: 50
         Function evaluations: 88
```

```
         Gradient evaluations: 88
Newton-CG
Optimization terminated successfully.
         Current function value: 3856.768202
         Iterations: 22
         Function evaluations: 29
         Gradient evaluations: 334
         Hessian evaluations: 0
TNC
        Current Function Value: 3856.76820177
        Iterations: 37
        Function Evaluations: 175
```

The condition number for this system matrix is 2404.5492682.

We can visualize the performance of the optimizers using R. The graph of their performance is given below, where I have plotted the norm of gradient, $||\nabla f||_2$ vs Time in Seconds taken to reach that value.



Finally, for the last case, we can see that the performance of TNC and NCG are somewhat comparable but L-BFGS, BFGS, and CG perfrom extremely poorly in this case. It is therefore certain that given the case when the data has heavy correlation, L-BFGS, BFGS, and CG optimizers should be avoided.

Mathematical Analysis

In this section, we will now do a brief mathematical analysis to understand the result of the computations that we made in the above sections.

We began our analysis with a small test using a simple 5×5 system of equations to check how the algorithms perform in an ideal state - a reasonable number of rows and columns with a low condition number. We can see that the BFGS algorithm performs the best in this case, as was expected since researchers today agree that the BFGS algorithm method is "best" in practice (Notes 13, pg 213) which can be attributed to the low dimensionality of the system being computed. But this property changes in the test example that I have used in this paper.

In the first data sample, the condition number of the matrix is 12.17 which is relatively small for a matrix that large. This number, however, is accompanied by heavy dimensionality - 1001 columns! BFGS is a good candidate for optimization in low-dimensional problems, which is why, it fails to perform well in this case. And with that condition number increasing in every other sample of data that we've used, we would expect the BFGS algorithm to continue perform poorly within this paradigm, which is reflected in the results.

The L-BFGS algorithm is the best performer in this case because this combines the properties of the BFGS algorithmic performance with a well-conditioned matrix and the L-BFGS property of performing better in high dimenional problems by storing a limited number of vectors for computation. In some papers, it has been noted that L-BFGS can be considered an extension of the Conjugate Gradient Method which uses fewer evaluations to get a similar result - which can be confirmed in our evaluation results above. The L-BFGS algorithm fails to perform well in data matrices which are ill-conditioned, as expected.

In the case of Conjugate Gradient Methods, we know that the implementaion performace of the algorithm depends directly on the condition of the data matrix. We can see that the performance of the Conjugate Gradient is competitive in the first sample with a well-conditioned matrix but that deteriorates consistently for cases with ill-conditioning.

A Truncated Newton Method is an algorithm that performs comparibly to the L-BFGS but uses an implemention of the Conjugate Gradient algorithm. In this case, the total effort is measured by the cumlative sum of CG steps and to get good performance, we need to tune the CG stopping criterion, one that will give us enough steps to find a good descent direction. Due to this criterion, the TNC performs sub-optimally in the case where CG performs at its best - in the first case. However, in later stages, we can see that the TNC outperforms other method and performs optimally with the Newton-CG in the last case with heavy ill-conditioning.

Lastly, we will discuss the performance of the Newton-CG. Given my limited knowledge in this algorithm, these methods are based on Newton iterations, coupled with conjugate-gradient iterations to solve the resulting linear Newton-correction equation. The performance of this method is sub-optimal in the first case where the there is little to no correlation in the data but the method outperforms other algorithms in the other instances.

Final Thoughts

We can see from the implementation these methods that the dimensionality and the condition number play an important role in determining the performance of an optimizer in the context of Logistic Regression. We have compared some classic methods with some new and advanced methods and seen how each behave in a number of different hypothetical cases.

References

- Ducci, John. Truncated Newton Method - Convex Optimization II Lecture
- Liu, Dong, and Jorge Nocedal. "ON LIMITED MEMORY BFGS METHOD FOR LARGE SCALE OPTIMIZATION." Department of Electrical Engineering and Computer Science, Northwestern University.

- Yang, Jianke. “Newton-Conjugate-Gradient Methods for Solitary Wave Computations.” Journal of Computational Physics, 2009.
- Yousef, Saad. “Iterative Methods for Sparse Linear Systems.” Society for Industrial and Applied Mathematics.