HW4_Question_4b

March 11, 2018

Group: Austin Wang, Joe Higgins, Lawrence Moore

```
In [1]: import numpy as np
        e = np.exp(1)
```

0.1 Data

As we did in Homework 3, we slightly alter the data points so that the data is no longer linearly separable. For the particular dataset from Homework 2 and Homework 3, this means subtracting 0.01 from both coordinates of the positive class (0,0) point, and adding 0.01 to both coordinates of the negative class (0,0) point.

```
In [2]: pos_data = np.matrix([
            [ -0.01, -0.01],
            [ 1, 0],
            [ 0, 1]
        ])
        neg_data = np.matrix([
            [ 0.01, 0.01],
            [-1, 0],
            [ 0,-1]
        ])

        # Add shift variable
        pos_data = np.hstack((pos_data, np.ones((3,1))))
        neg_data = np.hstack((neg_data, np.ones((3,1))))
```

0.2 Functions

Below are the implementations of the log-logistic-loss function gradient and Hessian. The gradient will be used in both the Quasi-Newton and Newton methods, while the Hessian will be used in only the Newton method.

```
In [3]: # Gradient of the log-logistic-loss function
        def grad(x):

            pos_data_sum = \
                e**(-1*np.dot(pos_data[0,:], x))/(1 + e**(-1*np.dot(pos_data[0,:], x))) * \
```

```
            pos_data[0,:] + e**(-1*np.dot(pos_data[1,:], x))/ \
            (1 + e**(-1*np.dot(pos_data[1,:], x))) * pos_data[1,:] + \
            e**(-1*np.dot(pos_data[2,:], x))/(1 + e**(-1*np.dot(pos_data[2,:], x))) * \
            pos_data[2,:]

        neg_data_sum = \
            e**(np.dot(neg_data[0,:], x))/(1 + e**(np.dot(neg_data[0,:], x))) * \
            neg_data[0,:] + e**(np.dot(neg_data[1,:], x))/ \
            (1 + e**(np.dot(neg_data[1,:], x))) * neg_data[1,:] + \
            e**(np.dot(neg_data[2,:], x))/(1 + e**(np.dot(neg_data[2,:], x))) * \
            neg_data[2,:]

        return np.transpose(-1*pos_data_sum + neg_data_sum)

    # Hessian of the log-logistic-loss function
    def hess(x):
        output_hess = np.zeros((3, 3))
        for j in range(3):
            pos_sum = np.zeros((1,3))
            neg_sum = np.zeros((1,3))
            for i in range(3):
                num = e**(-1 * np.dot(pos_data[i, :], x)) \
                      * pos_data[i, j]
                den = (1 + e**(-1 * np.dot(pos_data[i, :], x)))**2
                pos_sum += (num/den)*pos_data[i, :]

                num = e**(np.dot(neg_data[i, :], x)) \
                      * neg_data[i, j]
                den = (1 + e**(np.dot(neg_data[i, :], x)))**2
                neg_sum += (num/den)*neg_data[i, :]
            output_hess[j,:] = pos_sum + neg_sum
        return output_hess
```

0.3 Quasi-Newton Method

Below we implement the BFGS Quasi-Newton method:

```
In [4]: # Initialize algorithm variables
        alpha_k = 0.5 # Learning rate
        x_0 = np.matrix([
            [0],
            [0],
            [0]
        ])
        check = 1000
        max_iter = 10000
        k = 0
        x_k1 = x_0
```

```
        B_k = np.eye(3, 3)

        # Perform BFGS algorithm
        while check > 10**-8 and k < max_iter:
            k += 1
            x_k = x_k1
            p_k = np.linalg.solve(B_k, -1 * grad(x_k))
            s_k = alpha_k * p_k
            x_k1 = x_k + s_k
            alpha_k = alpha_k
            y_k = grad(x_k1) - grad(x_k)

            middle = np.dot(y_k, np.transpose(y_k)) / \
                np.dot(np.transpose(y_k), s_k)
            BS = np.dot(B_k, s_k)
            last = np.dot(BS, np.transpose(BS)) / \
                    np.dot(np.transpose(s_k), BS)

            B_k = B_k + middle - last
            check = np.linalg.norm(grad(x_k))
```

```
In [5]: print('Solution converged to: ')
        print(x_k1)
        print('Number of Iterations Necessary: ', k)
```

```
Solution converged to:
[[5.24199464]
 [5.24199464]
 [0.        ]]
Number of Iterations Necessary:  33
```

0.4 Newton Method

Our implementation of the Newton method, which *does* use the Hessian is as follows:

```
In [6]: x_0 = np.matrix([
            [0],
            [0],
            [0]
        ])
        check = 1000
        max_iter = 10000
        k = 0
        x_k = x_0

        while check > 10**-8 and k < max_iter:

            x_k1 = x_k - np.linalg.inv(hess(x_k)) * grad(x_k)
```

```
            check = np.linalg.norm(x_k1 - x_k)
            x_k = x_k1
            k = k+1
```

```
In [7]: print('Solution converged to: ')
        print(x_k1)
        print('Number of Iterations Necessary: ', k)
```

```
Solution converged to:
[[5.24199496]
 [5.24199496]
 [0.        ]]
Number of Iterations Necessary:  9
```

0.5 Conclusion

In the experiments above, we found that the Newton method convergence (9 iterations) was significantly faster than the Quasi-Newton method convergence (33 iterations). This is not surprising, as intuitively we are using more information about the objective function in the Newton method by providing the entire Hessian. When compared to the methods used in Homework 3, we find that the algorithms from fastest to slowest are:

1. Newton Method (9 iterations)
2. BB Method (12 iterations)
3. Quasi-Newton BFGS Method (33 iterations)
4. Conjugate Direction (371 iterations)
5. Accelerated Steepest Descent (805 iterations)
6. Steepest Descent (2557 iterations)