Kai Wong (704-451-679)

CS 145, Fall 2018

Homework #1

1. **Linear Regression**
   a. Closed-form solution

```
C:\Users\kyles\Desktop\CS145\hw1\HW1 Programming\LinearRegression
λ python linearRegression.py 0 0
Learning Algorithm Type:  0
Is normalization used:  0
Beta:  [-0.0862246   0.05340575  0.65803045  0.41731923 -0.01772481  0.30069864
  1.02871152  0.48383363  0.26685697  0.04573456  0.31944742  1.14776959
  0.29366213  0.41491543  0.85180482 -0.05950309  0.47235562  0.46198106
  0.00497427  0.0205398   0.41310473  0.98508025  0.15573467  0.8618602
  0.41974331 -0.06893699  0.33317496  0.27766637 -0.04184791 -0.23599504
  0.15020297  0.37745027  0.80256455  0.16053288  0.2744667   0.63461071
  0.74135259  0.56079776  0.94058723 -0.0432542   0.80803615  0.93967722
  0.12225161 -0.19933624  0.09398732  0.11412993  0.35479619  0.78582876
  0.38900433  0.11804526  0.67618837  0.70377377  0.05526258 -0.24019005
  0.87339793 -0.01381723  0.83138416  0.90569236  0.39980648  0.25235308
  0.69692397 -0.00949757  0.17676599  0.45822485  0.02743899  1.16718165
  0.04176352  1.01993881  0.56015024 -0.29761224  0.3177761   0.55781578
  1.1376088   0.55190283  0.4099807   0.91987238  1.34076835  0.53297825
  0.63648277  0.22140583  0.21469531 -0.00609269  0.82898663  0.46891532
 -0.25571565  0.1972989   1.38639797  0.87219453  0.65782257  0.54983464
  1.11698567  0.94267463  0.79030138  0.30055848  0.53288973  0.22873689
  0.86702876  0.98591924  0.08132528  0.30834368  0.70121488]
Training MSE:  3.4349195199049083
Test MSE:  4.396097860818799
```

Batch gradient descent

```
C:\Users\kyles\Desktop\CS145\hw1\HW1 Programming\LinearRegression
λ python linearRegression.py 1 0
Learning Algorithm Type:  1
Is normalization used:  0
Beta:  [ 0.40665898  0.05538594  0.44875029  0.05100071  0.38740305  0.11616919
  0.49596903  0.44585549  0.59759108  0.0410644   0.32267505  0.94053663
  0.50645872  0.28910674  0.79949857  0.29775249  0.62553201  0.38607749
  0.37280177  0.47810871  0.33472418  0.9113348   0.28872488  0.23658669
  0.7322789   0.12112573  0.10987849  0.15933707  0.51123543  0.47846995
  0.26759894  0.6202136   0.7956479   0.26033819  0.45834834  0.80399073
  0.78590368  0.15737526  0.27317199 -0.04921114  0.33136187  0.88645783
  0.23728425  0.14035175  0.37421499  0.24962977  0.17585142  0.19178849
  0.56462694  0.37478985  0.86305927  0.59652507  0.07932675  0.2173946
  0.8600279   0.34577032  0.67575142  0.81308816  0.68207839  0.22974359
  0.19564246  0.49166481  0.42356683  0.64269335  0.33695839  0.59626066
  0.28410297  0.9388493   0.53505475  0.33807749  0.2761005   0.48921808
  0.86429593  0.30440474  0.15791317  0.24934126  0.90132394  0.25306673
  0.64283351  0.60656828  0.62571507  0.36629387  0.27134952  0.50028716
  0.32845946  0.22069671  1.03075786  0.75023935  0.63707979  0.59075876
  0.60495274  0.29990051  0.72975734  0.13299885  0.38069159  0.34283049
  0.42430495  0.73355314  0.02107     0.29442261  0.48210969]
Training MSE:  4.152947462435229
Test MSE:  4.606043239084392
```

Stochastic gradient descent

```
C:\Users\kyles\Desktop\CS145\hw1\HW1 Programming\LinearRegression
λ python linearRegression.py 2 0
Learning Algorithm Type:  2
Is normalization used:  0
Beta:  [-0.06680393  0.08707394  0.63170074  0.44763911 -0.0695943   0.30606493
  1.04465698  0.54325128  0.28683367  0.08285257  0.23261116  1.16360627
  0.2746125   0.42200669  0.88175804 -0.06649516  0.46393972  0.47238395
  0.01991683  0.05781866  0.47538018  0.93438569  0.18232677  0.81101105
  0.44098078 -0.0588913   0.33283299  0.28347963 -0.01277878 -0.16055005
  0.18523592  0.34023806  0.67154608  0.19815612  0.32819601  0.67653493
  0.75743775  0.57557415  0.8966588  -0.05739598  0.79283683  0.95765923
  0.07441623 -0.18243319 -0.00755561  0.14576484  0.30051976  0.80751638
  0.40415136  0.12251418  0.69546426  0.65235662  0.10175816 -0.28420656
  0.82652542  0.03289948  0.82943105  0.96188668  0.42583134  0.26284621
  0.64889644 -0.01635652  0.1803895   0.48379174  0.00694008  1.17896778
  0.03467876  1.02202044  0.53292779 -0.33447088  0.31778476  0.56221107
  1.1275696   0.48971292  0.45959362  0.92691827  1.41909099  0.55694937
  0.58527946  0.20600487  0.23838747 -0.00557941  0.82976213  0.43354329
 -0.31539166  0.12429697  1.39738909  0.88137512  0.6693406   0.58002618
  1.18265974  1.00623641  0.79102965  0.26707063  0.56824698  0.22227386
  0.98164914  1.05418381  0.13041708  0.26526567  0.6985279 ]
Training MSE:  3.562687948225144
Test MSE:  4.544200377053288
```

The Test MSEs were relatively close for all three methods, but the closed-form solution performed the best (consistently) out of the optimization methods. This is to be expected for a small dataset such as this where the inverse of X^TX is relatively inexpensive to compute. The consistency and performance can be attributed to the use of a single, overall equation to determine Beta values, compared to the iterative approaches used in the gradient descent methods which begin with randomized Betas.

For the learned weights, there is no variation for the closed-form solution since again, a single overall equation is used to compute the Betas. For the gradient descent methods, this was not the case as different instances of the program would generate different learned weights. This is likely because the initial Beta values are randomized, hence the Beta would converge differently depending on its starting value.

In terms of execution time, there was a noticeable difference where using batch gradient descent was significantly slower than using stochastic gradient descent. This is to be expected since in batch GD, each Beta update during training only takes place after each epoch.

b. Closed-form solution

```
C:\Users\kyles\Desktop\CS145\hw1\HW1 Programming\LinearRegression
λ python linearRegression.py 0 1
Learning Algorithm Type:  0
Is normalization used:  1
Beta:  [ 2.27729720e+01  1.53267685e-01  1.85400036e-01  1.20001101e-01
 -5.02894960e-03  8.91855522e-02  2.85477509e-01  1.40249729e-01
  7.58001703e-02  1.29653087e-02  9.40114997e-02  3.31501951e-01
  8.48405150e-02  1.19998020e-01  2.42101087e-01 -1.70904428e-02
  1.37119556e-01  1.35350218e-01  1.41619004e-03  5.96423043e-03
  1.15830867e-01  2.84837752e-01  4.40248244e-02  2.49185633e-01
  1.20285952e-01 -1.97966211e-02  9.78939759e-02  8.05403060e-02
 -1.21241111e-02 -6.77059821e-02  4.42940642e-02  1.07814670e-01
  2.27982170e-01  4.72154203e-02  7.98729034e-02  1.82957097e-01
  2.10609705e-01  1.62079663e-01  2.74455584e-01 -1.24456123e-02
  2.32346197e-01  2.68821067e-01  3.49745502e-02 -5.73174263e-02
  2.74558199e-02  3.22366923e-02  1.03219840e-01  2.23792899e-01
  1.12445398e-01  3.34223468e-02  1.96611852e-01  2.04171370e-01
  1.61259528e-02 -7.12316220e-02  2.51757075e-01 -3.88735810e-03
  2.31055679e-01  2.65481860e-01  1.14239087e-01  7.19519080e-02
  2.03225977e-01 -2.77922653e-03  5.10043840e-02  1.31478537e-01
  7.74623329e-03  3.36781203e-01  1.19518825e-02  2.98145298e-01
  1.64253970e-01 -8.57326109e-02  9.04810592e-02  1.57878654e-01
  3.30578812e-01  1.58142457e-01  1.17519641e-01  2.66603450e-01
  3.90619100e-01  1.54573813e-01  1.82230684e-01  6.25165215e-02
  6.11873098e-02 -1.74345404e-03  2.34361003e-01  1.35158424e-01
 -7.34879378e-02  5.72871764e-02  4.02966409e-01  2.50642329e-01
  1.87572968e-01  1.57855445e-01  3.18225717e-01  2.66144412e-01
  2.29911686e-01  8.53225833e-02  1.56706806e-01  6.57087894e-02
  2.52781623e-01  2.90068806e-01  2.33284776e-02  9.01229905e-02
  2.03998476e-01]
Training MSE:  3.4349195199049083
Test MSE:  4.396097860818446
```

Batch gradient descent

```
C:\Users\kyles\Desktop\CS145\hw1\HW1 Programming\LinearRegression
λ python linearRegression.py 1 1
Learning Algorithm Type:  1
Is normalization used:  1
Beta:  [ 2.26277376e+01  1.62279716e-01  1.72496078e-01  1.22810174e-01
 -2.03124264e-02  9.11941437e-02  2.98449951e-01  1.51961678e-01
  7.34109662e-02  2.39856826e-02  1.06133380e-01  3.52579400e-01
  1.03345405e-01  1.38990277e-01  2.38307791e-01  3.44941227e-03
  1.29474494e-01  1.60075653e-01  9.51040078e-03  9.12958236e-03
  1.21069582e-01  2.78626976e-01  6.37579790e-02  2.43896241e-01
  1.16739354e-01 -5.93947184e-03  1.03126870e-01  9.04542081e-02
  2.25288706e-02 -4.59327490e-02  2.46513223e-02  1.22713489e-01
  2.45141135e-01  4.62323514e-02  9.33032424e-02  1.87636557e-01
  2.17514492e-01  1.57697755e-01  3.02120206e-01  5.95106813e-03
  2.63785549e-01  2.71016989e-01  2.67044765e-02 -4.71369084e-02
  3.79563959e-02  5.13962162e-02  1.03774530e-01  2.29020789e-01
  9.84034107e-02  4.82860981e-02  2.08212451e-01  1.92771887e-01
  4.53650744e-02 -5.69576788e-02  2.37870939e-01  1.07061652e-02
  2.40520382e-01  2.80922803e-01  1.16760578e-01  8.68335725e-02
  2.10576399e-01  1.13643268e-02  4.36603379e-02  1.34285892e-01
  4.23597391e-02  3.47391279e-01  2.45384138e-02  2.95234638e-01
  1.82758120e-01 -9.24295238e-02  9.29191670e-02  1.61155702e-01
  3.48686241e-01  1.73427910e-01  1.03982784e-01  2.64497123e-01
  4.03695285e-01  1.45837319e-01  1.93545489e-01  6.12032819e-02
  5.39446975e-02 -9.77968209e-04  2.31030677e-01  1.50564876e-01
 -7.06232312e-02  7.92916559e-02  4.13631242e-01  2.44528731e-01
  2.02354117e-01  1.83355514e-01  3.28029041e-01  2.58089995e-01
  2.48345607e-01  9.09709763e-02  1.49456371e-01  8.10492029e-02
  2.73347774e-01  2.82227721e-01  2.95706078e-02  9.65340815e-02
  1.97349924e-01]
Training MSE:  3.4668953255043884
Test MSE:  4.393893739834045
```

Stochastic gradient descent

```
C:\Users\kyles\Desktop\CS145\hw1\HW1 Programming\LinearRegression
λ python linearRegression.py 2 1
Learning Algorithm Type:  2
Is normalization used:  1
Beta:  [ 2.29123621e+01  2.40062177e-01  2.48105436e-01  1.24519968e-01
 -1.27076309e-01  1.43465290e-01  4.02761401e-01  1.11323359e-01
  1.00145559e-01 -1.74648258e-02  1.43125095e-01  2.66541462e-01
  9.57843725e-02  2.13915756e-01  2.74680436e-01 -1.38196567e-01
 -2.12147413e-02  4.40167570e-02  4.15823063e-03  1.00915831e-01
  5.69813349e-02  3.61881724e-01  1.33814212e-01  2.29673920e-01
  1.89480510e-01 -1.09164224e-01 -1.04684698e-01  1.21858701e-01
 -5.56724767e-02 -1.79435703e-01  4.74446525e-02  7.50333397e-02
  1.18168085e-01  1.20880159e-01 -3.60343151e-02  1.62568555e-01
  8.14943109e-02  5.12890801e-02  2.77182797e-01 -2.60816954e-02
  2.66998942e-01  2.93698845e-01 -4.84698941e-03  1.38106262e-02
  1.23803837e-01 -4.48286228e-02  1.68345840e-01  2.46425502e-01
  1.96073313e-01  4.30531086e-02  3.25691382e-01  1.74909650e-01
  1.57935072e-01 -1.10026020e-01  2.71359907e-01 -4.22966310e-02
  1.32712154e-01  4.03921608e-01  1.64946266e-01  1.78825736e-01
  2.21739748e-01 -3.23012238e-02  1.39852730e-01  9.32695138e-02
  4.24640352e-03  3.28309651e-01 -1.04851805e-01  5.49783542e-01
  1.52278254e-01  7.26040838e-02  1.80956869e-01  9.56742688e-02
  2.25677140e-01  8.04580866e-02  1.77563798e-01  1.93384133e-01
  4.19082974e-01  3.58012513e-01  1.75665230e-01  1.03767265e-01
 -7.05884646e-02 -9.02770489e-03  2.59705040e-01  2.68506622e-01
 -1.36830203e-01  6.46786949e-03  3.79848413e-01  2.83693143e-01
  1.38012541e-01  2.96422444e-01  3.53101385e-01  3.41489975e-01
  1.54652325e-01 -1.81068842e-03  2.46698551e-01 -1.35980594e-01
  1.65735360e-01  2.58413476e-01 -1.72113164e-01  1.85520089e-01
  1.90199762e-01]
Training MSE:  4.235506513640409
Test MSE:  5.08569515750769
```

Applying z-score normalization changed the learned Betas for all 3 optimization methods. This is expected since normalization is meant to change (by standardizing) the raw values of each feature x. Hence, the methods would converge to a different set of Betas since the values of the data and feature parameters are now different.

For the closed-form solution, normalization did not affect both Training and Test MSEs. This is expected since the entire feature and data (both training and test) set is uniformly standardized, so the computed Beta would also be scaled accordingly => The same equation should not give a different prediction.

For batch gradient descent, there was an improvement in both performance (lower MSE) as well as consistency. This is because normalization improves (regularizes) the topology of the loss function by ensuring that the parameters are centered onto the same scale. Unnormalized data may place more emphasis on certain parameter gradients, hence larger parameter gradients could dominate each update, resulting in divergent behavior. This could not only increase the time needed for convergence, but also result in convergence to less optimum beta values.

For stochastic gradient descent, there was improved stability in Test MSE values (less variance across different program instances) but there was a slight decrease in the model's accuracy (Test MSEs of 4.6 could be observed without normalization, but with normalization Test MSE was consistent in the 5.0-5.1 range). This could be because in un-normalized data, there might be outlier values that result in biased estimates of the true gradient during each Beta update despite performing shuffling of input data per iteration within a training epoch. As a result, lower Test MSEs may be a non-deterministic consequence of random inputs each iteration (shuffling) and outlier data values that create a bias. Similarly, SGD without normalization can also give higher Test MSEs well above the consistent 5.0-5.1 range seen in SGD on normalized data.

2. Logistic Regression and Model Selection

a. Batch gradient descent

```
C:\Users\kyles\Desktop\CS145\hw1\HW1 Programming\LogisticRegression
λ python logisticRegression.py 0 0
Learning Algorithm Type:  0
Is normalization used:  0
average logL for iteration 0: 13.759099820105785
average logL for iteration 1000: 0.02092969617727125
average logL for iteration 2000: 0.020891302190242756
average logL for iteration 3000: 0.020865455721394685
average logL for iteration 4000: 0.020848844462198632
average logL for iteration 5000: 0.020838626633848778
average logL for iteration 6000: 0.020832532443439113
average logL for iteration 7000: 0.02082896811838793
average logL for iteration 8000: 0.020826907910883416
average logL for iteration 9000: 0.020825725332010003
Beta:  [ 8.66798931 -9.20361177 -5.0125613  -6.0495453  -0.70946458]
Training avgLogL:  0.023499797705119018
Test accuracy:  0.9890510948905109
```

Newton Raphson method

```
C:\Users\kyles\Desktop\CS145\hw1\HW1 Programming\LogisticRegression
λ python logisticRegression.py 1 0
Learning Algorithm Type:  1
Is normalization used:  0
average logL for iteration 0: 0.11172894405790369
average logL for iteration 1000: 0.018429477101347836
average logL for iteration 2000: 0.01842947710134781
average logL for iteration 3000: 0.0184294771013478
average logL for iteration 4000: 0.018429477101347836
average logL for iteration 5000: 0.018429477101347815
average logL for iteration 6000: 0.01842947710134781
average logL for iteration 7000: 0.018429477101347815
average logL for iteration 8000: 0.01842947710134781
average logL for iteration 9000: 0.018429477101347836
Python 3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> ^Z

now exiting InteractiveConsole...
Beta:  [ 7.31317701 -7.70705368 -4.15787617 -5.21346398 -0.58583733]
Training avgLogL:  0.018429477101347784
Test accuracy:  0.9890510948905109
```

The test accuracy was the same (possibly because both methods converged to the objective function's global minimum), but the learned weights were different. More importantly, execution time when using the Newton-Raphson method was significantly longer than when using batch gradient descent. This is because instead of using a learning rate hyperparameter, the Newton-Raphson method uses the inverse hessian of the objective function (the curvature of the loss function topology) to take a more direct route towards the minimum of the function. Doing so may result in faster convergence compared to batch gradient descent, but computing the inverse hessian is computationally expensive using numerical methods. In addition, Newton's method requires the entire solution space of the function to be convex, which may not be the case for most modern ML problems.

b. I implemented a new function def getBeta_RegularizedBatchGradient() which took the same arguments as the original def getBeta_BatchGradient() but computed a regularized derivative of the negative log likelihood function instead.

The new first derivative now has a bias term $\lambda \sum_{j=0}^{P} 2\beta_j$, and is also divided by n, the number of training samples. The variable λ is a new hyperparameter that controls regularization strength which I set to 1 for simplicity.

I added the option to run > python logisticRegression.py 2 0 which would use this new regularized batch gradient descent algorithm, and I got the following experimental results:

```
C:\Users\kyles\Desktop\CS145\hw1\HW1 Programming\LogisticRegression
λ python logisticRegression.py 2 0
Learning Algorithm Type:  2
Is normalization used:  0
average logL for iteration 0: 2.4066638700077556
average logL for iteration 1000: 0.11432508491836023
average logL for iteration 2000: 0.08198495673348262
average logL for iteration 3000: 0.06966943390931862
average logL for iteration 4000: 0.06242415749457248
average logL for iteration 5000: 0.057411900408816656
average logL for iteration 6000: 0.053650777890022504
average logL for iteration 7000: 0.050688650869525856
average logL for iteration 8000: 0.04827992608249172
average logL for iteration 9000: 0.04627631982179383
Beta:  [ 1.68838846 -1.93873721 -1.16398326 -1.31095727 -0.27838572]
Training avgLogL:  0.044583123232740926
Test accuracy:  0.9890510948905109
```

I noticed that both training loss and test accuracy did not improve – test accuracy was identical to using unregularized batch gradient descent and training loss even had a slight increase.

This is expected since regularization does not improve performance on the same data set but is intended to improve the model's generalizability. Technically, the model should have better performance on new, unseen data which I did not test. As mentioned during lecture, regularization will add bias to the model if it has high variance (overfitting) and does so by adding a bias term to the objective function that increases as the values of the learned betas increase. Doing so modifies the original optimization problem by adding an additional constraint.