20F-COMSCI145 HW1

DANNING LIU YU

TOTAL POINTS

96 / 100

QUESTION 1

Linear regression 50 pts

1.1 Closed form solution 10 / 10

✓ **- 0 pts** Correct

1.2 Batch gradient solution 9 / 10

✓ **- 1 pts** bgd not converged

1.3 Stochastic gradient 10 / 10

✓ **- 0 pts** Correct

1.4 Questions 17 / 20

✓ **- 3 pts** good explaination or mistakes in Q3

QUESTION 2

Logistic regression 50 pts

2.1 Batch gradient descent 15 / 15

✓ **- 0 pts** Correct

2.2 Newton Raphson 15 / 15

✓ **- 0 pts** Correct

2.3 Questions 20 / 20

✓ **- 0 pts** Correct

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

If you can successfully run the code above, there will be no problem for environment setting.

1. Linear regression

This workbook will walk you through a linear regression example.

In [84]:
```
from hw1code.linear_regression import LinearRegression

lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.csv')
# As a sanity check, we print out the size of the training data (1000, 100) and trainin
print('Training data shape: ', lm.train_x.shape)
print('Training labels shape:', lm.train_y.shape)
```

```
Training data shape:  (1000, 100)
Training labels shape: (1000,)
```

1.1 Closed form solution

In this section, complete the `getBeta` function in `linear_regression.py` which use the close for solution of $\hat{\beta}$.

Train you model by using `lm.train('0')` function.

Print the training error and the testing error using `lm.predict` and `lm.compute_mse` given.

In [92]:
```
from hw1code.linear_regression import LinearRegression

lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.csv')
training_error= 0
testing_error= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.normalize() # Normalize data
beta = lm.train('0')

predict_train_y = lm.predict(lm.train_x, beta)
training_error = lm.compute_mse(predict_train_y, lm.train_y)

predict_test_y = lm.predict(lm.test_x, beta)
testing_error = lm.compute_mse(predict_test_y, lm.test_y)
#=======================#
#   END YOUR CODE HERE   #
#=======================#
print('Training error is: ', training_error)
print('Testing error is: ', testing_error)
```

```
Learning Algorithm Type:  0
Training error is:  0.08693886675396784
Testing error is:  0.11017540281675804
```

1.2 Batch gradient descent

1.1 Closed form solution 10 / 10

✓ - 0 pts Correct

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

If you can successfully run the code above, there will be no problem for environment setting.

1. Linear regression

This workbook will walk you through a linear regression example.

In [84]:
```
from hw1code.linear_regression import LinearRegression

lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.csv')
# As a sanity check, we print out the size of the training data (1000, 100) and trainin
print('Training data shape: ', lm.train_x.shape)
print('Training labels shape:', lm.train_y.shape)
```

```
Training data shape:  (1000, 100)
Training labels shape: (1000,)
```

1.1 Closed form solution

In this section, complete the `getBeta` function in `linear_regression.py` which use the close for solution of $\hat{\beta}$.

Train you model by using `lm.train('0')` function.

Print the training error and the testing error using `lm.predict` and `lm.compute_mse` given.

In [92]:
```
from hw1code.linear_regression import LinearRegression

lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.csv')
training_error= 0
testing_error= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.normalize() # Normalize data
beta = lm.train('0')

predict_train_y = lm.predict(lm.train_x, beta)
training_error = lm.compute_mse(predict_train_y, lm.train_y)

predict_test_y = lm.predict(lm.test_x, beta)
testing_error = lm.compute_mse(predict_test_y, lm.test_y)
#=======================#
#   END YOUR CODE HERE   #
#=======================#
print('Training error is: ', training_error)
print('Testing error is: ', testing_error)
```

```
Learning Algorithm Type:  0
Training error is:  0.08693886675396784
Testing error is:  0.11017540281675804
```

1.2 Batch gradient descent

In this section, complete the `getBetaBatchGradient` function in `linear_regression.py` which compute the gradient of the objective fuction.

Train you model by using `lm.train('1')` function.

Print the training error and the testing error using `lm.predict` and `lm.compute_mse` given.

In [95]:
```
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.csv')
training_error= 0
testing_error= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.normalize() # Normalize data
beta = lm.train('1')

predict_train_y = lm.predict(lm.train_x, beta)
training_error = lm.compute_mse(predict_train_y, lm.train_y)

predict_test_y = lm.predict(lm.test_x, beta)
testing_error = lm.compute_mse(predict_test_y, lm.test_y)
#=======================#
#   END YOUR CODE HERE   #
#=======================#
print('Training error is: ', training_error) # fix typo: accuracy -> error
print('Testing error is: ', testing_error)
```

```
Learning Algorithm Type:  1
Training error is:  0.0956626816930878
Testing error is:  0.12915886805147245
```

1.3 Stochastic gadient descent

In this section, complete the `getBetaStochasticGradient` function in `linear_regression.py` , which use an estimated gradient of the objective function.

Train you model by using `lm.train('2')` function.

Print the training error and the testing error using `lm.predict` and `lm.compute_mse` given.

In [94]:
```
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.csv')
training_error= 0
testing_error= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.lr = 0.0005 # need to decrease learning rate, or else overflow occurs without data n
lm.normalize() # Normalize data
beta = lm.train('2')

predict_train_y = lm.predict(lm.train_x, beta)
training_error = lm.compute_mse(predict_train_y, lm.train_y)

predict_test_y = lm.predict(lm.test_x, beta)
testing_error = lm.compute_mse(predict_test_y, lm.test_y)
#=======================#
```

1.2 Batch gradient solution 9 / 10

✓ **- 1 pts** bgd not converged

In this section, complete the `getBetaBatchGradient` function in `linear_regression.py` which compute the gradient of the objective fuction.

Train you model by using `lm.train('1')` function.

Print the training error and the testing error using `lm.predict` and `lm.compute_mse` given.

In [95]:
```
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.csv')
training_error= 0
testing_error= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.normalize() # Normalize data
beta = lm.train('1')

predict_train_y = lm.predict(lm.train_x, beta)
training_error = lm.compute_mse(predict_train_y, lm.train_y)

predict_test_y = lm.predict(lm.test_x, beta)
testing_error = lm.compute_mse(predict_test_y, lm.test_y)
#=======================#
#   END YOUR CODE HERE   #
#=======================#
print('Training error is: ', training_error) # fix typo: accuracy -> error
print('Testing error is: ', testing_error)
```

```
Learning Algorithm Type:  1
Training error is:  0.0956626816930878
Testing error is:  0.12915886805147245
```

1.3 Stochastic gadient descent

In this section, complete the `getBetaStochasticGradient` function in `linear_regression.py` , which use an estimated gradient of the objective function.

Train you model by using `lm.train('2')` function.

Print the training error and the testing error using `lm.predict` and `lm.compute_mse` given.

In [94]:
```
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.csv')
training_error= 0
testing_error= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.lr = 0.0005 # need to decrease learning rate, or else overflow occurs without data n
lm.normalize() # Normalize data
beta = lm.train('2')

predict_train_y = lm.predict(lm.train_x, beta)
training_error = lm.compute_mse(predict_train_y, lm.train_y)

predict_test_y = lm.predict(lm.test_x, beta)
testing_error = lm.compute_mse(predict_test_y, lm.test_y)
#=======================#
```

```
#   END YOUR CODE HERE   #
#========================#
print('Training error is: ', training_error)
print('Testing error is: ', testing_error)
```

```
Learning Algorithm Type:  2
Training error is:  0.08699734697688297
Testing error is:  0.10998019498187048
```

Questions:

1. Compare the MSE on the testing dataset for each version. Are they the same? Why or why not?
2. Apply z-score normalization for eachh featrure and comment whether or not it affect the three algorithm.
3. Ridge regression is adding an L2 regularization term to the original objective function of mean squared error. The objective function become following:

$$J(\beta) = \frac{1}{2n}\sum_i \left(x_i^T\beta - y_i\right)^2 + \frac{\lambda}{2n}\sum_j \beta_j^2,$$

   where $\lambda \leq 0$, which is a hyper parameter that controls the trade off. Take the derivative of this provided objective function and derive the closed form solution for β.

Your answer here:

Problem 1

The MSEs for each version are as follows:

Closed form:
Training error is: 0.08693886675396784
Testing error is: 0.11017540281675801

Batch Gradient Descent:
Training error is: 0.08694495274449923
Testing error is: 0.1102627661478727

Stochastic Gradient Descent:
(Note: learning rate was set to 0.0005, as if it is set to 0.005, the calculation diverges. Also, since this is based on randomly picking points, the results will slightly differ each time the code is run)
Learning Algorithm Type: 2
Training error is: 0.09365641087228466
Testing error is: 0.11363831746544562

None of the three solutions are the same. We know that the closed form is the most accurate solution (limited only by floating point precision). The gradient descent results are different from the closed form because they are numerical approximations, and thus are not perfectly accurate. Comparing batch and stochastic gradient descent results to the closed form shows that batch gradient descent is more accurate than stochastic gradient descent, which makes sense, as batch gradient uses all the points in the dataset while SGD randomly picks points to use and thus has more noise in its final result. Batch gradient descent differed from the closed form by ~0.00002 and

1.3 Stochastic gradient **10 / 10**

✓ **- 0 pts** **Correct**

```
#   END YOUR CODE HERE   #
#========================#
print('Training error is: ', training_error)
print('Testing error is: ', testing_error)
```

```
Learning Algorithm Type:  2
Training error is:  0.08699734697688297
Testing error is:  0.10998019498187048
```

Questions:

1. Compare the MSE on the testing dataset for each version. Are they the same? Why or why not?
2. Apply z-score normalization for eachh featrure and comment whether or not it affect the three algorithm.
3. Ridge regression is adding an L2 regularization term to the original objective function of mean squared error. The objective function become following:

$$J(\beta) = \frac{1}{2n}\sum_i \left(x_i^T\beta - y_i\right)^2 + \frac{\lambda}{2n}\sum_j \beta_j^2,$$

   where $\lambda \leq 0$, which is a hyper parameter that controls the trade off. Take the derivative of this provided objective function and derive the closed form solution for β.

Your answer here:

Problem 1

The MSEs for each version are as follows:

Closed form:
Training error is: 0.08693886675396784
Testing error is: 0.11017540281675801

Batch Gradient Descent:
Training error is: 0.08694495274449923
Testing error is: 0.1102627661478727

Stochastic Gradient Descent:
(Note: learning rate was set to 0.0005, as if it is set to 0.005, the calculation diverges. Also, since this is based on randomly picking points, the results will slightly differ each time the code is run)
Learning Algorithm Type: 2
Training error is: 0.09365641087228466
Testing error is: 0.11363831746544562

None of the three solutions are the same. We know that the closed form is the most accurate solution (limited only by floating point precision). The gradient descent results are different from the closed form because they are numerical approximations, and thus are not perfectly accurate. Comparing batch and stochastic gradient descent results to the closed form shows that batch gradient descent is more accurate than stochastic gradient descent, which makes sense, as batch gradient uses all the points in the dataset while SGD randomly picks points to use and thus has more noise in its final result. Batch gradient descent differed from the closed form by ~0.00002 and

~0.0001 for training and test data, respectively, while stochastic gradient descent differed from closed form by ~0.006 and ~0.003 for training and test data, respectively.

Problem 2
The MSEs for each version after normalizing the data (by calling `self.normalize()` before `self.train()`):

Closed form:
Training error is: 0.08693886675396784
Testing error is: 0.11017540281675804

Batch Gradient Descent:
Training error is: 0.0956626816930878
Testing error is: 0.12915886805147245

Stochastic Gradient Descent:
(Note: learning rate was set to 0.0005, as if it is set to 0.005, the calculation diverges. Also, since this is based on randomly picking points, the results will slightly differ each time the code is run)
Training error is: 0.08700262160334213
Testing error is: 0.10994697412903368

The results are nearly the same, with the closed form normalized error matching the non-normalized error, while the two gradient descent results are very similar to their respective non-normalized results. This makes sense, as normalization is simply a linear transformation of data, and thus the new linear regression model will incorporate it into its parameters. The minor differences are most likely due randomness and the fact that some precision is lost when the data is transformed.

Problem 3

$$\mathbf{J}(\boldsymbol{\beta}) = \frac{1}{2n}\left(\sum_i \left(\mathbf{x}_i^T\boldsymbol{\beta} - y_i\right)^2 + \lambda\sum_j \beta_j^2\right)$$

$$\mathbf{J}(\boldsymbol{\beta}) = \frac{1}{2n}\left(\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_2^2\right) = \frac{1}{2n}\left((\mathbf{X}\boldsymbol{\beta} - \mathbf{y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\right)$$

$$\mathbf{J}(\boldsymbol{\beta}) = \frac{1}{2n}\left((\boldsymbol{\beta}^T\mathbf{X}^T - \mathbf{y}^T)(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\right)$$

$$\mathbf{J}(\boldsymbol{\beta}) = \frac{1}{2n}\left(\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\boldsymbol{\beta} + \mathbf{y}^T\mathbf{y} + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\right)$$

Now note that $\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y}$ is a scalar and the transpose of a scalar is itself, so we can make the following transformation:

$$\left(\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y}\right)^T = \mathbf{y}^T\left(\boldsymbol{\beta}^T\mathbf{X}^T\right)^T = \mathbf{y}^T\mathbf{X}\boldsymbol{\beta}$$

$$\mathbf{J}(\boldsymbol{\beta}) = \frac{1}{2n}\left(\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - 2\left(\mathbf{X}^T\mathbf{y}\right)^T\boldsymbol{\beta} + \mathbf{y}^T\mathbf{y} + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\right)$$

$$\frac{\partial\mathbf{J}(\boldsymbol{\beta})}{\partial\boldsymbol{\beta}} = \frac{1}{2n}\left(2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - 2\mathbf{X}^T\mathbf{y} + 2\lambda\boldsymbol{\beta}\right) = \frac{1}{n}\left(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \mathbf{X}^T\mathbf{y} + \lambda\boldsymbol{\beta}\right)$$

Now set this derivative to 0 to minimize the objective to find the optimal value for $\boldsymbol{\beta}$:

$$\frac{\partial\mathbf{J}(\boldsymbol{\beta})}{\partial\boldsymbol{\beta}} = 0 = \frac{1}{n}\left(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \mathbf{X}^T\mathbf{y} + \lambda\boldsymbol{\beta}\right)$$

$$\mathbf{X}^T\mathbf{y} = \boldsymbol{\beta}\left(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + \lambda\mathbf{I}\right)$$

$$\boldsymbol{\beta} = \left(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

2. Logistic regression

This workbook will walk you through a logistic regression example.

In [24]:
```
from hw1code.logistic_regression import LogisticRegression

lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-test.cs
# As a sanity chech, we print out the size of the training data (1000, 5) and training
print('Training data shape: ', lm.train_x.shape)
print('Training labels shape:', lm.train_y.shape)
```

```
Training data shape:  (1000, 5)
Training labels shape: (1000,)
```

2.1 Batch gradient descent

In this section, complete the `getBeta_BatchGradient` in `logistic_regression.py`, which compute the gradient of the log likelihoood function.

Complete the `compute_avglogL` function in `logistic_regression.py` for sanity check.

1.4 Questions 17 / 20

✓ **- 3 pts good explaination or mistakes in Q3**

Problem 3

$$\mathbf{J}(\boldsymbol{\beta}) = \frac{1}{2n}\left(\sum_i \left(\mathbf{x}_i^T\boldsymbol{\beta} - y_i\right)^2 + \lambda\sum_j \beta_j^2\right)$$

$$\mathbf{J}(\boldsymbol{\beta}) = \frac{1}{2n}\left(\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_2^2\right) = \frac{1}{2n}\left((\mathbf{X}\boldsymbol{\beta} - \mathbf{y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\right)$$

$$\mathbf{J}(\boldsymbol{\beta}) = \frac{1}{2n}\left((\boldsymbol{\beta}^T\mathbf{X}^T - \mathbf{y}^T)(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\right)$$

$$\mathbf{J}(\boldsymbol{\beta}) = \frac{1}{2n}\left(\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\boldsymbol{\beta} + \mathbf{y}^T\mathbf{y} + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\right)$$

Now note that $\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y}$ is a scalar and the transpose of a scalar is itself, so we can make the following transformation:

$$\left(\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y}\right)^T = \mathbf{y}^T\left(\boldsymbol{\beta}^T\mathbf{X}^T\right)^T = \mathbf{y}^T\mathbf{X}\boldsymbol{\beta}$$

$$\mathbf{J}(\boldsymbol{\beta}) = \frac{1}{2n}\left(\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - 2\left(\mathbf{X}^T\mathbf{y}\right)^T\boldsymbol{\beta} + \mathbf{y}^T\mathbf{y} + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}\right)$$

$$\frac{\partial\mathbf{J}(\boldsymbol{\beta})}{\partial\boldsymbol{\beta}} = \frac{1}{2n}\left(2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - 2\mathbf{X}^T\mathbf{y} + 2\lambda\boldsymbol{\beta}\right) = \frac{1}{n}\left(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \mathbf{X}^T\mathbf{y} + \lambda\boldsymbol{\beta}\right)$$

Now set this derivative to 0 to minimize the objective to find the optimal value for $\boldsymbol{\beta}$:

$$\frac{\partial\mathbf{J}(\boldsymbol{\beta})}{\partial\boldsymbol{\beta}} = 0 = \frac{1}{n}\left(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \mathbf{X}^T\mathbf{y} + \lambda\boldsymbol{\beta}\right)$$

$$\mathbf{X}^T\mathbf{y} = \boldsymbol{\beta}\left(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + \lambda\mathbf{I}\right)$$

$$\boldsymbol{\beta} = \left(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

2. Logistic regression

This workbook will walk you through a logistic regression example.

In [24]:
```
from hw1code.logistic_regression import LogisticRegression

lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-test.cs
# As a sanity chech, we print out the size of the training data (1000, 5) and training
print('Training data shape: ', lm.train_x.shape)
print('Training labels shape:', lm.train_y.shape)
```

```
Training data shape:  (1000, 5)
Training labels shape: (1000,)
```

2.1 Batch gradient descent

In this section, complete the `getBeta_BatchGradient` in `logistic_regression.py`, which compute the gradient of the log likelihoood function.

Complete the `compute_avglogL` function in `logistic_regression.py` for sanity check.

Train you model by using `lm.train('0')` function.

And print the training and testing accuracy using `lm.predict` and `lm.compute_accuracy` given.

In [75]:
```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-test.cs
training_accuracy= 0
testing_accuracy= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.normalize() # if data is not normalized, overflow occurs
beta = lm.train('0')
# print(beta)

predict_train_y = lm.predict(lm.train_x, beta)
training_accuracy = lm.compute_accuracy(predict_train_y, lm.train_y)

predict_test_y = lm.predict(lm.test_x, beta)
testing_accuracy = lm.compute_accuracy(predict_test_y, lm.test_y)

#=======================#
#   END YOUR CODE HERE   #
#=======================#
print('Training accuracy is: ', training_accuracy)
print('Testing accuracy is: ', testing_accuracy)
```

```
average logL for iteration 0: -0.6143836332678481
average logL for iteration 1000: -0.5118493716923482
average logL for iteration 2000: -0.4886334504768928
average logL for iteration 3000: -0.4791415756020926
average logL for iteration 4000: -0.4738742439413998
average logL for iteration 5000: -0.4705447170409768
average logL for iteration 6000: -0.46829210228844503
average logL for iteration 7000: -0.4666963765362078
average logL for iteration 8000: -0.4655242354941027
average logL for iteration 9000: -0.4646364845289794
Training avgLogL:  -0.46394669216828555
Training accuracy is:  0.799
Testing accuracy is:  0.7514910536779325
```

2.2 Newton Raphhson

In this section, complete the `getBeta_Newton` in `logistic_regression.py`, which make use of both first and second derivative.

Train you model by using `lm.train('1')` function.

Print the training and testing accuracy using `lm.predict` and `lm.compute_accuracy` given.

In [74]:
```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-test.cs
training_accuracy= 0
testing_accuracy= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.normalize()
```

2.1 Batch gradient descent 15 / 15

✓ - 0 pts Correct

Train you model by using `lm.train('0')` function.

And print the training and testing accuracy using `lm.predict` and `lm.compute_accuracy` given.

In [75]:
```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-test.cs
training_accuracy= 0
testing_accuracy= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.normalize() # if data is not normalized, overflow occurs
beta = lm.train('0')
# print(beta)

predict_train_y = lm.predict(lm.train_x, beta)
training_accuracy = lm.compute_accuracy(predict_train_y, lm.train_y)

predict_test_y = lm.predict(lm.test_x, beta)
testing_accuracy = lm.compute_accuracy(predict_test_y, lm.test_y)

#=======================#
#   END YOUR CODE HERE   #
#=======================#
print('Training accuracy is: ', training_accuracy)
print('Testing accuracy is: ', testing_accuracy)
```

```
average logL for iteration 0: -0.6143836332678481
average logL for iteration 1000: -0.5118493716923482
average logL for iteration 2000: -0.4886334504768928
average logL for iteration 3000: -0.4791415756020926
average logL for iteration 4000: -0.4738742439413998
average logL for iteration 5000: -0.4705447170409768
average logL for iteration 6000: -0.46829210228844503
average logL for iteration 7000: -0.4666963765362078
average logL for iteration 8000: -0.4655242354941027
average logL for iteration 9000: -0.4646364845289794
Training avgLogL:  -0.46394669216828555
Training accuracy is:  0.799
Testing accuracy is:  0.7514910536779325
```

2.2 Newton Raphhson

In this section, complete the `getBeta_Newton` in `logistic_regression.py`, which make use of both first and second derivative.

Train you model by using `lm.train('1')` function.

Print the training and testing accuracy using `lm.predict` and `lm.compute_accuracy` given.

In [74]:
```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-test.cs
training_accuracy= 0
testing_accuracy= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.normalize()
```

```
beta = lm.train('1')
# print(beta)

predict_train_y = lm.predict(lm.train_x, beta)
training_accuracy = lm.compute_accuracy(predict_train_y, lm.train_y)

predict_test_y = lm.predict(lm.test_x, beta)
testing_accuracy = lm.compute_accuracy(predict_test_y, lm.test_y)
#=======================#
#   END YOUR CODE HERE   #
#=======================#
print('Training accuracy is: ', training_accuracy)
print('Testing accuracy is: ', testing_accuracy)
```

```
average logL for iteration 0: -0.5490333349721084
average logL for iteration 500: -0.49621656724560864
average logL for iteration 1000: -0.47492585568212775
average logL for iteration 1500: -0.4660544141314995
average logL for iteration 2000: -0.46243177001304364
average logL for iteration 2500: -0.46099448340010934
average logL for iteration 3000: -0.460438178231231
average logL for iteration 3500: -0.46022672404107146
average logL for iteration 4000: -0.4601473282342417
average logL for iteration 4500: -0.46011775260794563
average logL for iteration 5000: -0.46010679040991975
average logL for iteration 5500: -0.4601027398769609
average logL for iteration 6000: -0.4601012460622463
average logL for iteration 6500: -0.46010069579506097
average logL for iteration 7000: -0.4601004932409696
average logL for iteration 7500: -0.46010041871285146
average logL for iteration 8000: -0.4601003912980606
average logL for iteration 8500: -0.46010038121528235
average logL for iteration 9000: -0.4601003775073364
average logL for iteration 9500: -0.4601003761438175
Training avgLogL:  -0.4601003756430145
Training accuracy is:  0.797
Testing accuracy is:  0.7534791252485089
```

Questions:

1. Compare the accuracy on the testing dataset for each version. Are they the same? Why or why not?
2. Regularization. Similar to linear regression, an regularization term could be added to logistic regression. The objective function becomes following:

$$J(\beta) = -\frac{1}{n}\sum_i \left(y_i x_i^T \beta - \log\left(1+\exp\{x_i^T\beta\}\right)\right) + \lambda \sum_j \beta_j^2,$$

where $\lambda \leq 0$, which is a hyper parameter that controls the trade off. Take the derivative $\frac{\partial J(\beta)}{\partial \beta_j}$ of this provided objective function and provide the batch gradient descent update.

Your answer here:

Problem 1

They are almost the same, with batch gradient descent having an accuracy of 0.7515 for the test data and Newton-Rhapson having an accuracy of 0.7535 for the same test data. This is only a difference of 0.002, which is quite small. This makes sense, as both are numerical methods that

2.2 Newton Raphson **15 / 15**

✓ **- 0 pts** **Correct**

```
beta = lm.train('1')
# print(beta)

predict_train_y = lm.predict(lm.train_x, beta)
training_accuracy = lm.compute_accuracy(predict_train_y, lm.train_y)

predict_test_y = lm.predict(lm.test_x, beta)
testing_accuracy = lm.compute_accuracy(predict_test_y, lm.test_y)
#=======================#
#   END YOUR CODE HERE   #
#=======================#
print('Training accuracy is: ', training_accuracy)
print('Testing accuracy is: ', testing_accuracy)
```

```
average logL for iteration 0: -0.5490333349721084
average logL for iteration 500: -0.49621656724560864
average logL for iteration 1000: -0.47492585568212775
average logL for iteration 1500: -0.4660544141314995
average logL for iteration 2000: -0.46243177001304364
average logL for iteration 2500: -0.46099448340010934
average logL for iteration 3000: -0.460438178231231
average logL for iteration 3500: -0.46022672404107146
average logL for iteration 4000: -0.4601473282342417
average logL for iteration 4500: -0.46011775260794563
average logL for iteration 5000: -0.46010679040991975
average logL for iteration 5500: -0.4601027398769609
average logL for iteration 6000: -0.4601012460622463
average logL for iteration 6500: -0.46010069579506097
average logL for iteration 7000: -0.4601004932409696
average logL for iteration 7500: -0.46010041871285146
average logL for iteration 8000: -0.4601003912980606
average logL for iteration 8500: -0.46010038121528235
average logL for iteration 9000: -0.4601003775073364
average logL for iteration 9500: -0.4601003761438175
Training avgLogL:  -0.4601003756430145
Training accuracy is:  0.797
Testing accuracy is:  0.7534791252485089
```

Questions:

1. Compare the accuracy on the testing dataset for each version. Are they the same? Why or why not?
2. Regularization. Similar to linear regression, an regularization term could be added to logistic regression. The objective function becomes following:

$$J(\beta) = -\frac{1}{n}\sum_i \left(y_i x_i^T \beta - \log\left(1+\exp\{x_i^T\beta\}\right)\right) + \lambda \sum_j \beta_j^2,$$

where $\lambda \leq 0$, which is a hyper parameter that controls the trade off. Take the derivative $\frac{\partial J(\beta)}{\partial \beta_j}$ of this provided objective function and provide the batch gradient descent update.

Your answer here:

Problem 1

They are almost the same, with batch gradient descent having an accuracy of 0.7515 for the test data and Newton-Rhapson having an accuracy of 0.7535 for the same test data. This is only a difference of 0.002, which is quite small. This makes sense, as both are numerical methods that

iteratively converge upon the same solution (the greatest log-likelihood). However, the batch gradient descent method did not seem to converge within 10000 iterations, as the average logL value kept changing, while the Newton-Raphson method did after around 4000 iterations, which makes sense, as it is an 2nd order approximation.

Problem 2

$$\mathbf{J}(\boldsymbol{\beta}) = -\frac{1}{n}\sum_i \left(y_i \mathbf{x}_i^T \boldsymbol{\beta} - \log\left(1+\exp\left(\mathbf{x}_i^T \boldsymbol{\beta}\right)\right)\right) + \lambda \sum_j \beta_j^2$$

$$\frac{\partial \mathbf{J}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\frac{1}{n}\sum_i \left(\mathbf{x}_i y_i - \frac{\frac{\partial}{\partial \boldsymbol{\beta}}\left(1+\exp\left(\mathbf{x}_i^T \boldsymbol{\beta}\right)\right)}{1+\exp\left(\mathbf{x}_i^T \boldsymbol{\beta}\right)}\right) + 2\lambda\boldsymbol{\beta}$$

$$\frac{\partial \mathbf{J}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\frac{1}{n}\sum_i \left(\mathbf{x}_i y_i - \frac{\mathbf{x}_i \exp\left(\mathbf{x}_i^T \boldsymbol{\beta}\right)}{1+\exp\left(\mathbf{x}_i^T \boldsymbol{\beta}\right)}\right) + 2\lambda\boldsymbol{\beta}$$

$$\frac{\partial \mathbf{J}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\frac{1}{n}\sum_i \left(\mathbf{x}_i \left(y_i - \sigma\left(\mathbf{x}_i^T \boldsymbol{\beta}\right)\right)\right) + 2\lambda\boldsymbol{\beta}$$

$$\boldsymbol{\beta}^{(n+1)} = \boldsymbol{\beta}^{(n)} + \eta\left(2\lambda\boldsymbol{\beta}^{(n)} - \frac{1}{n}\sum_i \left(\mathbf{x}_i \left(y_i - \sigma\left(\mathbf{x}_i^T \boldsymbol{\beta}^{(n)}\right)\right)\right)\right)$$

2.3 Visualize the decision boundary on a toy dataset

In this subsection, you will use the same implementation for another small dataset with each datapoint x with only two features (x_1, x_2) to visualize the decision boundary of logistic regression model.

In [2]:
```
from hw1code.logistic_regression import LogisticRegression

lm=LogisticRegression(verbose = False)
lm.load_data('./data/logistic-regression-toy.csv','./data/logistic-regression-toy.csv')
# As a sanity chech, we print out the size of the training data (99,2) and training lab
print('Training data shape: ', lm.train_x.shape)
print('Training labels shape:', lm.train_y.shape)
```

```
Training data shape:  (99, 2)
Training labels shape: (99,)
```

In the following block, you can apply the same implementation of logistic regression model (either in 2.1 or 2.2) to the toy dataset. Print out the $\hat{\beta}$ after training and accuracy on the train set.

In [3]:
```
training_accuracy= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.normalize()
beta = lm.train('0')
print(beta)
predict_train_y = lm.predict(lm.train_x, beta)
training_accuracy = lm.compute_accuracy(predict_train_y, lm.train_y)

#=======================#
```

```
#   END YOUR CODE HERE   #
#========================#
print('Training accuracy is: ', training_accuracy)
```

```
Training avgLogL:  -0.32938172655990533
[-0.0398782   1.41612141  1.97753997]
Training accuracy is:  0.8888888888888888
```

Next, we try to plot the decision boundary of your learned logistic regression classifier. Generally, a decision boundary is the region of a space in which the output label of a classifier is ambiguous. That is, in the given toy data, given a datapoint $x = (x_1, x_2)$ on the decision boundary, the logistic regression classifier cannot decide whether $y = 0$ or $y = 1$.

Question

Is the decision boundary for logistic regression linear? Why or why not?

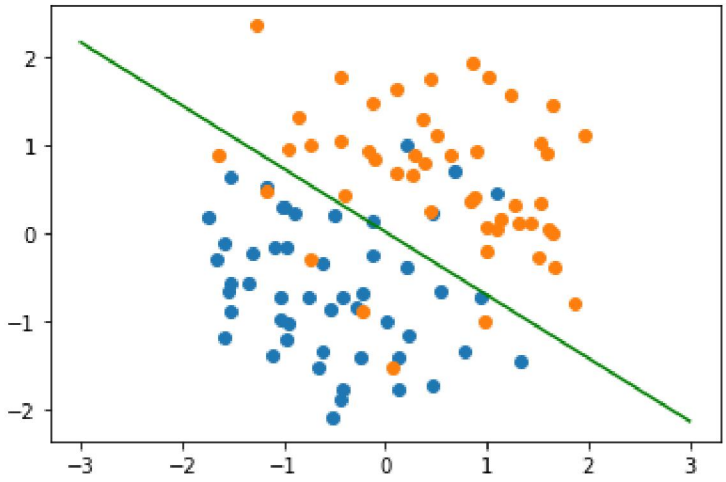
Your answer here:

The decision boundary for logistic regression is linear. The decision boundary is the set of points where $\sigma(a) = 0.5$. When $\sigma(a) = 0.5$, then $a = 0$, which means that $a = x^T\beta = 0$. For the case when there are two features, it becomes the equation $b_0 + x_1b_1 + x_2b_2$, which is a linear equation with respect to x_1 and x_2.

Draw the decision boundary in the following cell. Note that the code to plot the raw data points are given. You may need `plt.plot` function (see here).

In [15]:
```
# scatter plot the raw data
df = pd.concat([lm.train_x, lm.train_y], axis=1)
groups = df.groupby("y")
for name, group in groups:
    plt.plot(group["x1"], group["x2"], marker="o", linestyle="", label=name)

# plot the decision boundary on top of the scattered points
#========================#
# STRART YOUR CODE HERE  #
#========================#
x1_vals = np.arange(-3, 3, 0.001)
x2_vals = -1 * (x1_vals * beta[1] + beta[0]) / beta[2]
plt.plot(x1_vals, x2_vals, "g-")
#========================#
#   END YOUR CODE HERE   #
#========================#
plt.show()
```



End of Homework 1 :)

After you've finished the homework, please print out the entire `ipynb` notebook and two `py` files into one PDF file. Make sure you include the output of code cells and answers for questions. Prepare submit it to GradeScope.

2.3 Questions **20 / 20**

✓ **- 0 pts** **Correct**