CS145 Howework 1

Important Note: HW1 is due on **11:59 PM PT, Oct 19 (Monday, Week 3)**. Please submit through GradeScope (you will receive an invite to Gradescope for CS145 Fall 2020.).

Print Out Your Name and UID

Name: Rui Deng, UID: 205123245

Before You Start

You need to first create HW1 conda environment by the given `cs145hw1.yml` file, which provides the name and necessary packages for this tasks. If you have `conda` properly installed, you may create, activate or deactivate by the following commands:

```
conda env create -f cs145hw1.yml
conda activate hw1
conda deactivate
```

OR

```
conda env create --name NAMEOFYOURCHOICE -f cs145hw1.yml
conda activate NAMEOFYOURCHOICE
conda deactivate
```

To view the list of your environments, use the following command:

```
conda env list
```

More useful information about managing environments can be found here (https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html).

You may also quickly review the usage of basic Python and Numpy package, if needed in coding for matrix operations.

In this notebook, you must not delete any code cells in this notebook. If you change any code outside the blocks that you are allowed to edit (between `STRART/END YOUR CODE HERE`), you need to highlight these changes. You may add some additional cells to help explain your results and observations.

In [1]:

```
import numpy as np
import pandas as pd
import sys
import random as rd
import matplotlib.pyplot as plt
%load_ext autoreload
%autoreload 2
```

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 [2]:

```
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
 training labels (1000,)
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 [1]:

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

beta = lm.train('0')
predicted_train_y = lm.predict(lm.train_x, beta)
training_error = lm.compute_mse(predicted_train_y, lm.train_y)
predicted_test_y = lm.predict(lm.test_x, beta)
testing_error = lm.compute_mse(predicted_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: ', 0.110175402816758)
```

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 [2]:

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

beta = lm.train('1')
predicted_train_y = lm.predict(lm.train_x, beta)
training_error = lm.compute_mse(predicted_train_y, lm.train_y)
predicted_test_y = lm.predict(lm.test_x, beta)
testing_error = lm.compute_mse(predicted_test_y, lm.test_y)

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

```
('Learning Algorithm Type: ', '1')
('Training accuracy is: ', 0.08694023606971792)
('Testing accuracy is: ', 0.11021514372539926)
```

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 [4]:

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

beta = lm.train('2')
predicted_train_y = lm.predict(lm.train_x, beta)
training_error = lm.compute_mse(predicted_train_y, lm.train_y)
predicted_test_y = lm.predict(lm.test_x, beta)
testing_error = lm.compute_mse(predicted_test_y, lm.test_y)

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

```
('Learning Algorithm Type: ', '2')
('Training accuracy is: ', 0.09862058021123295)
('Testing accuracy is: ', 0.11634315938682402)
```

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 $\lambda \geq 0$, 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:

Please type your answer here!

1. MSE for closed form solution: 0.110175402816758

for batch gradient descent: 0.11021514372539926

for stochastic gradient descent (with learning rate 0.0005): 0.11634315938682402

They are not the same. For closed form solution, the MSE is the same no matter how many times we run the algorithm. This is because we are calculating beta from a pre-defined formula and there is no randomness in closed form solution. However, there are small differences in MSE if we run the gradient descent method for multiple times, since we initialize beta randomly and also use random mini-batch for stochastic gradient descent. Thus, with such randomness, MSE from the last two methods cannot be the same with MSE in closed form solution.

Still, all of the three methods give us similar error, since they all converge to the optimal solution.

1. After normalization: for closed form solution: 0.110175402816758

for batch gradient descent: 0.13744440164075847

for stochastic gradient descent (with learning rate 0.0005): 0.13721370635089097

Normalization does not change the MSE for closed form solution, but it does increase the MSE of testing set for both gradient descent methods. This is because we are using the mean and standard deviation of the training set to normalize the testing set, and the statistics for the training set may not be appropriate for the testing set.

1. Derivative:

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

If we write this in vector form, we have:

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

Set this derivative to 0, and we get:

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

as our new closed form solution.

2. Logistic regression

This workbook will walk you through a logistic regression example.

In [3]:

```
from hw1code.logistic_regression import LogisticRegression

lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-test.csv')
# As a sanity chech, we print out the size of the training data (1000, 5) and training labels (1000,)
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 gradiend 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 [19]:

```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-test.csv')
training_accuracy= 0
testing_accuracy= 0
#=========================#
# STRART YOUR CODE HERE  #
#=========================#
beta = lm.train('0')
predicted_train_y = lm.predict(lm.train_x, beta)
training_accuracy = lm.compute_accuracy(predicted_train_y, lm.train_y)
predicted_test_y = lm.predict(lm.test_x, beta)
testing_accuracy = lm.compute_accuracy(predicted_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: -0.575495420086
average logL for iteration 1000: -0.504071168498
average logL for iteration 2000: -0.484364560378
average logL for iteration 3000: -0.475661811496
average logL for iteration 4000: -0.47087006589
average logL for iteration 5000: -0.467925360099
average logL for iteration 6000: -0.465993392131
average logL for iteration 7000: -0.46466668148
average logL for iteration 8000: -0.463722801457
average logL for iteration 9000: -0.463031297563
('Training avgLogL: ', -0.46251212811749887)
('Training accuracy is: ', 0.8)
('Testing accuracy is: ', 0.7415506958250497)
```

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 [26]:

```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-test.csv')
training_accuracy= 0
testing_accuracy= 0
#=========================#
# STRART YOUR CODE HERE  #
#=========================#
beta = lm.train('1')
predicted_train_y = lm.predict(lm.train_x, beta)
training_accuracy = lm.compute_accuracy(predicted_train_y, lm.train_y)
predicted_test_y = lm.predict(lm.test_x, beta)
testing_accuracy = lm.compute_accuracy(predicted_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: -0.620851256358
average logL for iteration 500: -0.527671233888
average logL for iteration 1000: -0.489163287119
average logL for iteration 1500: -0.472294578524
average logL for iteration 2000: -0.465045797813
average logL for iteration 2500: -0.46204601611
average logL for iteration 3000: -0.460848349572
average logL for iteration 3500: -0.460383332502
average logL for iteration 4000: -0.460206285828
average logL for iteration 4500: -0.460139749201
average logL for iteration 5000: -0.460114951079
average logL for iteration 5500: -0.460105756944
average logL for iteration 6000: -0.460102359119
average logL for iteration 6500: -0.460101105889
average logL for iteration 7000: -0.460100644216
average logL for iteration 7500: -0.460100474267
average logL for iteration 8000: -0.460100411734
average logL for iteration 8500: -0.460100388732
average logL for iteration 9000: -0.460100380272
average logL for iteration 9500: -0.46010037716
('Training avgLogL: ', -0.46010037601754294)
('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 $\lambda \geq 0$, 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:

Please type your answer here!

1. Testing accuracy for gradient descent: 0.7415506958250497 for Newton Raphson: 0.7534791252485089 Newton Raphson has slightly higher accuracy than batch gradient descent, probably because that it adjusts learning rate throughout the optimization process through the hessian matrix instead of using a pre-defined learning rate.
2. Derivative

$$\frac{\partial J(\beta)}{\partial \beta_j} = -\frac{1}{n}\sum_i x_{ij}\left(y_i - p_i(\beta)\right) + 2\lambda\beta_j$$

, and we define p_i as

$$p_i = \frac{\exp\{x_i^T\beta\}}{1+\exp\{x_i^T\beta\}}$$

Then, we have the update rule as:

$$\beta^{new} = \beta^{old} + \frac{\eta}{n}\sum_i x_i\left(y_i - p_i(\beta)\right) + 2\eta\lambda\beta^{old}$$

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 [27]:

```
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 labels (99,)
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 [28]:

```
training_accuracy= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
beta = lm.train('0')
print(beta)
predicted_train_y = lm.predict(lm.train_x, beta)
training_accuracy = lm.compute_accuracy(predicted_train_y, lm.train_y)
#=======================#
#   END YOUR CODE HERE   #
#=======================#
print('Training accuracy is: ', training_accuracy)
```

```
('Training avgLogL: ', -0.32937413231151724)
[-0.04119331  1.41642477  1.97907353]
('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:

Please type your answer here!

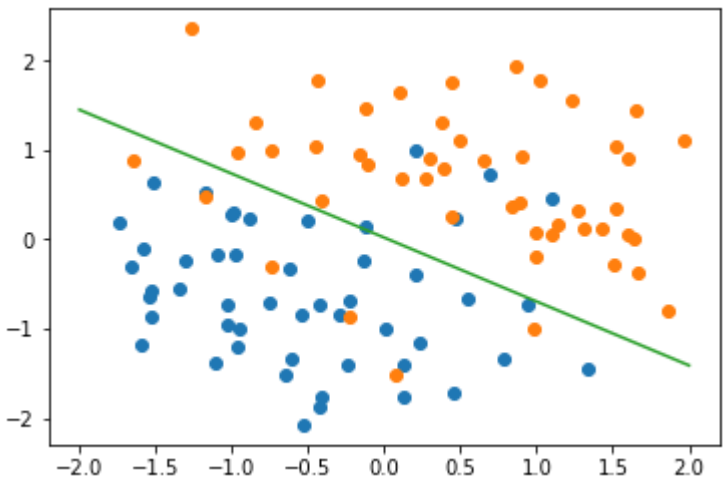
The boundary is linear because if $X\beta > 0$, our sigmoid function will give us probability > 0.5 thus belong to label 1; and vice versa. Thus, our decision boundary is $X\beta = 0$, which is a hyperplane in higher dimension and a line in 2D.

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 (https://matplotlib.org/tutorials/introductory/pyplot.html)).

In [30]:

```
# 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  #
#========================#
xs = np.linspace(-2, 2, 100)
ys = - beta[1] / beta[2] * xs - beta[0] / beta[2]
plt.plot(xs, ys)
#========================#
#   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.

```
import pandas as pd
import numpy as np
import sys
import random as rd

#insert an all-one column as the first column
def addAllOneColumn(matrix):
    n = matrix.shape[0] #total of data points
    p = matrix.shape[1] #total number of attributes

    newMatrix = np.zeros((n,p+1))
    newMatrix[:,1:] = matrix
    newMatrix[:,0] = np.ones(n)

    return newMatrix

# Reads the data from CSV files, converts it into Dataframe and returns x and y
 dataframes
def getDataframe(filePath):
    dataframe = pd.read_csv(filePath)
    y = dataframe['y']
    x = dataframe.drop('y', axis=1)
    return x, y

# train_x and train_y are numpy arrays
# function returns value of beta calculated using (0) the formula beta =
 (X^T*X)^ -1)*(X^T*Y)
def getBeta(train_x, train_y):
    n = train_x.shape[0] #total of data points
    p = train_x.shape[1] #total number of attributes

    beta = np.zeros(p)
    #========================#
    # STRART YOUR CODE HERE  #
    #========================#

    beta = np.linalg.inv(np.matmul(train_x.transpose(), train_x))
    beta = np.matmul(beta, np.matmul(train_x.transpose(), train_y))

    #========================#
    #   END YOUR CODE HERE   #
    #========================#
    return beta

# train_x and train_y are numpy arrays
# lr (learning rate) is a scalar
# function returns value of beta calculated using (1) batch gradient descent
def getBetaBatchGradient(train_x, train_y, lr, num_iter):
    beta = np.random.rand(train_x.shape[1])

    n = train_x.shape[0] #total of data points
```

```
    p = train_x.shape[1] #total number of attributes

    beta = np.random.rand(p)

    #update beta interatively
    for iter in range(0, num_iter):
       deriv = np.zeros(p)
       for i in range(n):
           #========================#
           # STRART YOUR CODE HERE  #
           #========================#

            deriv = deriv + train_x[i].dot(train_x[i].transpose().dot(beta) -
             train_y[i])

           #========================#
           #   END YOUR CODE HERE   #
           #========================#
       deriv = deriv / n
       beta = beta - deriv.dot(lr)
    return beta

# train_x and train_y are numpy arrays
# lr (learning rate) is a scalar
# function returns value of beta calculated using (2) stochastic gradient
 descent
def getBetaStochasticGradient(train_x, train_y, lr):
    n = train_x.shape[0] #total of data points
    p = train_x.shape[1] #total number of attributes

    beta = np.random.rand(p)

    epoch = 100
    for iter in range(epoch):
        indices = list(range(n))
        rd.shuffle(indices)
        for i in range(100):
            idx = indices[i]
           #========================#
           # STRART YOUR CODE HERE  #
           #========================#

            grad = (train_y[idx] - train_x[idx].dot(beta)) * train_x[idx]
            beta = beta + lr * grad

           #========================#
           #   END YOUR CODE HERE   #
           #========================#
    return beta
```

```
# Linear Regression implementation
class LinearRegression(object):
    # Initializes by reading data, setting hyper-parameters, and forming linear
     model
    # Forms a linear model (learns the parameter) according to type of beta (0
     - closed form, 1 - batch gradient, 2 - stochastic gradient)
    # Performs z-score normalization if z_score is 1
    def __init__(self,lr=0.005, num_iter=1000):
        self.lr = lr
        self.num_iter = num_iter
        self.train_x = pd.DataFrame()
        self.train_y = pd.DataFrame()
        self.test_x = pd.DataFrame()
        self.test_y = pd.DataFrame()
        self.algType = 0
        self.isNormalized = 0

    def load_data(self, train_file, test_file):
        self.train_x, self.train_y = getDataframe(train_file)
        self.test_x, self.test_y = getDataframe(test_file)

    def normalize(self):
        # Applies z-score normalization to the dataframe and returns a
         normalized dataframe
        self.isNormalized = 1
        means = self.train_x.mean(0)
        std = self.train_x.std(0)
        self.train_x = (self.train_x - means).div(std)
        self.test_x = (self.test_x - means).div(std)

    # Gets the beta according to input
    def train(self, algType):
        self.algType = algType

        if self.isNormalized == 0:
            self.normalize()

        newTrain_x = addAllOneColumn(self.train_x.values) #insert an all-one
         column as the first column
        print('Learning Algorithm Type: ', algType)

        if(algType == '0'):
            beta = getBeta(newTrain_x, self.train_y.values)
            #print('Beta: ', beta)

        elif(algType == '1'):
            beta = getBetaBatchGradient(newTrain_x, self.train_y.values,
             self.lr, self.num_iter)
            #print('Beta: ', beta)
        elif(algType == '2'):
```

```
            beta = getBetaStochasticGradient(newTrain_x, self.train_y.values,
             self.lr * 0.1)
            #print('Beta: ', beta)
        else:
            print('Incorrect beta_type! Usage: 0 - closed form solution, 1 -
             batch gradient descent, 2 - stochastic gradient descent')


        return beta

    # Predicts the y values on given data and learned beta
    def predict(self,x, beta):
        newTest_x = addAllOneColumn(x)
        self.predicted_y = newTest_x.dot(beta)
        return self.predicted_y


    # predicted_y and y are the predicted and actual y values respectively as
     numpy arrays
    # function returns the mean squared error (MSE) value for the test dataset
    def compute_mse(self,predicted_y, y):
        mse = np.sum((predicted_y - y)**2)/predicted_y.shape[0]
        return mse
```

```
# -*- coding: utf-8 -*-

import pandas as pd
import numpy as np
import sys
import random as ra


# insert an all-one column as the first column
def addAllOneColumn(matrix):
    n = matrix.shape[0]  # total of data points
    p = matrix.shape[1]  # total number of attributes

    newMatrix = np.zeros((n, p + 1))
    newMatrix[:, 0] = np.ones(n)
    newMatrix[:, 1:] = matrix

    return newMatrix


# Reads the data from CSV files, converts it into Dataframe and returns x and y
 dataframes
def getDataframe(filePath):
    dataframe = pd.read_csv(filePath)
    y = dataframe['y']
    x = dataframe.drop('y', axis=1)
    return x, y


# sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))


# compute average logL
def compute_avglogL(X, y, beta):
    eps = 1e-50
    n = y.shape[0]
    avglogL = 0
    # =========================#
    # STRART YOUR CODE HERE  #
    # =========================#
    for i in range(n):
        avglogL += y[i] * X[i].T.dot(beta) - np.log(1 +
         np.exp(X[i].T.dot(beta)))
    avglogL /= n
    # =========================#
    #   END YOUR CODE HERE   #
    # =========================#
    return avglogL
```

```
# train_x and train_y are numpy arrays
# lr (learning rate) is a scalar
# function returns value of beta calculated using (0) batch gradient descent
def getBeta_BatchGradient(train_x, train_y, lr, num_iter, verbose):
    beta = np.random.rand(train_x.shape[1])

    n = train_x.shape[0]  # total of data points
    p = train_x.shape[1]  # total number of attributes

    grad = 0
    beta = np.random.rand(p)
    # update beta interatively
    for iter in range(0, num_iter):
        # ========================#
        # STRART YOUR CODE HERE  #
        # ========================#
        grad = train_x.T.dot(train_y - sigmoid(train_x.dot(beta)))
        grad /= n
        beta = beta + grad.dot(lr)
        # ========================#
        #   END YOUR CODE HERE   #
        # ========================#
        if (verbose == True and iter % 1000 == 0):
            avgLogL = compute_avglogL(train_x, train_y, beta)
            print('average logL for iteration {}: {} \t').format(iter, avgLogL)
    return beta


# train_x and train_y are numpy arrays
# function returns value of beta calculated using (1) Newton-Raphson method
def getBeta_Newton(train_x, train_y, num_iter, verbose):
    n = train_x.shape[0]  # total of data points
    p = train_x.shape[1]  # total number of attributes

    grad = np.zeros(p)
    beta = np.random.rand(p)
    for iter in range(0, num_iter):
        # ========================#
        # STRART YOUR CODE HERE  #
        # ========================#
        sig = sigmoid(train_x.dot(beta))
        grad = train_x.T.dot(train_y - sig)
        grad /= n
        hessian = train_x.T.dot(np.diag(sig * (1-sig))).dot(train_x)
        beta = beta + np.linalg.inv(hessian).dot(grad)
        # ========================#
        #   END YOUR CODE HERE   #
        # ========================#
        if (verbose == True and iter % 500 == 0):
            avgLogL = compute_avglogL(train_x, train_y, beta)
```

```
            print('average logL for iteration {}: {} \t').format(iter, avgLogL)
    return beta


# Logistic Regression implementation
class LogisticRegression(object):
    # Initializes by reading data, setting hyper-parameters
    # Learns the parameter using (0) Batch gradient (1) Newton-Raphson
    # Performs z-score normalization if isNormalized is 1
    # Print intermidate training loss if verbose = True
    def __init__(self, lr=0.005, num_iter=10000, verbose=True):
        self.lr = lr
        self.num_iter = num_iter
        self.verbose = verbose
        self.train_x = pd.DataFrame()
        self.train_y = pd.DataFrame()
        self.test_x = pd.DataFrame()
        self.test_y = pd.DataFrame()
        self.algType = 0
        self.isNormalized = 0

    def load_data(self, train_file, test_file):
        self.train_x, self.train_y = getDataframe(train_file)
        self.test_x, self.test_y = getDataframe(test_file)

    def normalize(self):
        # Applies z-score normalization to the dataframe and returns a
         normalized dataframe
        self.isNormalized = 1
        data = np.append(self.train_x, self.test_x, axis=0)
        means = data.mean(0)
        std = data.std(0)
        self.train_x = (self.train_x - means).div(std)
        self.test_x = (self.test_x - means).div(std)

    # Gets the beta according to input
    def train(self, algType):
        self.algType = algType

        if self.isNormalized == 0:
            self.normalize()

        newTrain_x = addAllOneColumn(self.train_x.values)  # insert an all-one
         column as the first column
        if (algType == '0'):
            beta = getBeta_BatchGradient(newTrain_x, self.train_y.values,
             self.lr, self.num_iter, self.verbose)
            # print('Beta: ', beta)

        elif (algType == '1'):
```

```
            beta = getBeta_Newton(newTrain_x, self.train_y.values,
             self.num_iter, self.verbose)
            # print('Beta: ', beta)
        else:
            print('Incorrect beta_type! Usage: 0 - batch gradient descent, 1 -
             Newton-Raphson method')

        train_avglogL = compute_avglogL(newTrain_x, self.train_y.values, beta)
        print('Training avgLogL: ', train_avglogL)

        return beta

    # Predict on given data x with learned parameter beta
    def predict(self, x, beta):
        newTest_x = addAllOneColumn(x)
        self.predicted_y = (sigmoid(newTest_x.dot(beta)) >= 0.5)
        return self.predicted_y

    # predicted_y and y are the predicted and actual y values respectively as
     numpy arrays
    # function returns the accuracy
    def compute_accuracy(self, predicted_y, y):
        acc = float(np.sum(predicted_y == y)) / predicted_y.shape[0]
        return acc
```