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: Ali Mirabzadeh , UID: 305179067

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

More useful information about managing environments can be found [here (https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html)](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
import matplotlib.image as mpimg
%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-regress
ion-test.csv')
# As a sanity check, we print out the size of the training data (1000, 1
00) 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 [3]:
```
from hw1code.linear_regression import LinearRegression

lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regress
ion-test.csv')
training_error= 0
testing_error= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#
beta = lm.train('0')
train_predict = lm.predict(lm.train_x.values, beta)
test_predict = lm.predict(lm.test_x.values, beta)
training_error =  lm.compute_mse(train_predict,lm.train_y.values)
testing_error = lm.compute_mse(test_predict, lm.test_y.values)
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Training error is: ', training_error)
print('Testing error is: ', testing_error)

## below for normalizing
lm.normalize()
beta_normalized = lm.train('0')
train_predict_n = lm.predict(lm.train_x.values, beta_normalized)
test_predict_n = lm.predict(lm.test_x.values, beta_normalized)
training_error_n =  lm.compute_mse(train_predict_n,lm.train_y.values)
testing_error_n = lm.compute_mse(test_predict_n, lm.test_y.values)
print('Normalized training error is: ', training_error_n)
print('Normalized testing error is: ', testing_error_n)
```

```
Learning Algorithm Type:  0
Training error is:  0.08693886675396784
Testing error is:  0.11017540281675804
Learning Algorithm Type:  0
Normalized training error is:  0.08693886675396784
Normalized 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 [4]:
```python
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regress
ion-test.csv')
training_error= 0
testing_error= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#
beta = lm.train('1')
train_predict = lm.predict(lm.train_x.values, beta)
test_predict = lm.predict(lm.test_x.values, beta)
training_error =  lm.compute_mse(train_predict,lm.train_y.values)
testing_error = lm.compute_mse(test_predict, lm.test_y.values)
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Training error is: ', training_error)
print('Testing error is: ', testing_error)

## below for normalizing
lm.normalize()
beta_normalized = lm.train('1')
train_predict_n = lm.predict(lm.train_x.values, beta_normalized)
test_predict_n = lm.predict(lm.test_x.values, beta_normalized)
training_error_n =  lm.compute_mse(train_predict_n,lm.train_y.values)
testing_error_n = lm.compute_mse(test_predict_n, lm.test_y.values)
print('Normalized training error is: ', training_error_n)
print('Normalized testing error is: ', testing_error_n)
```

```
Learning Algorithm Type:  1
Training error is:  0.08693919081192665
Testing error is:  0.11019432186477264
Learning Algorithm Type:  1
Normalized training error is:  0.09654351096262996
Normalized testing error is:  0.12865993449113894
```

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 [5]:
```python
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regress
ion-test.csv')
training_error= 0
testing_error= 0
#========================#
# START YOUR CODE HERE  #
#========================#
beta = lm.train('2')
train_predict = lm.predict(lm.train_x.values, beta)
test_predict = lm.predict(lm.test_x.values, beta)
training_error =  lm.compute_mse(train_predict,lm.train_y.values)
testing_error = lm.compute_mse(test_predict, lm.test_y.values)
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Training error is: ', training_error)
print('Testing error is: ', testing_error)

## below for normalizing
lm.normalize()
beta_normalized = lm.train('2')
train_predict_n = lm.predict(lm.train_x.values, beta_normalized)
test_predict_n = lm.predict(lm.test_x.values, beta_normalized)
training_error_n =  lm.compute_mse(train_predict_n,lm.train_y.values)
testing_error_n = lm.compute_mse(test_predict_n, lm.test_y.values)
print('Normalized training error is: ', training_error_n)
print('Normalized testing error is: ', testing_error_n)
```

```
Learning Algorithm Type:  2
Training error is:  91.10508480511325
Testing error is:  96.50500424208748
Learning Algorithm Type:  2
Normalized training error is:  31.219733063437648
Normalized testing error is:  29.420724030847268
```

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:

1. With some small precision difference, they are the same. We see the error rates to vary because thse are different models. The stochastic gradient descent has the largest error rate because our dataset is not large enough for this method to be efficent. More precise methods like batch gradient descent and linear regression are more effective.
2. You can see each section (1.1, 1.2, 1.3) as I have written the code for it and see the printed result. With the normalized data we can see an increase in error rate for batch gradient descent, no change in error rate for linear regression and a decrease in error rate for stochastic gradient descent.
3.

$$\beta = \left(x^T x + \lambda\right)^- 1 + \left(x^T y\right)$$

2. Logistic regression

This workbook will walk you through a logistic regression example.

In [6]:
```
from hw1code.logistic_regression import LogisticRegression

lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-reg
ression-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 [7]:
```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-reg
ression-test.csv')
training_accuracy= 0
testing_accuracy= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#
lm.normalize()
beta = lm.train('0')
train_predict = lm.predict(lm.train_x.values, beta)
test_predict = lm.predict(lm.test_x.values, beta)
training_accuracy =  lm.compute_accuracy(train_predict,lm.train_y.values
)
testing_accuracy = lm.compute_accuracy(test_predict, lm.test_y.values)
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Training accuracy is: ', training_accuracy)
print('Testing accuracy is: ', testing_accuracy)
```

```
average logL for iteration 0: -0.5561796575778383
average logL for iteration 1000: -0.4615656578257467
average logL for iteration 2000: -0.4615656578257467
average logL for iteration 3000: -0.4615656578257467
average logL for iteration 4000: -0.4615656578257467
average logL for iteration 5000: -0.4615656578257467
average logL for iteration 6000: -0.4615656578257467
average logL for iteration 7000: -0.4615656578257467
average logL for iteration 8000: -0.4615656578257467
average logL for iteration 9000: -0.4615656578257467
Training avgLogL:  -0.4615656578257467
Training accuracy is:  0.798
Testing accuracy is:  0.7495029821073559
```

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 [8]:
```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-reg
ression-test.csv')
training_accuracy= 0
testing_accuracy= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#
lm.normalize()
beta = lm.train('1')
train_predict = lm.predict(lm.train_x.values, beta)
test_predict = lm.predict(lm.test_x.values, beta)
training_accuracy =  lm.compute_accuracy(train_predict,lm.train_y.values
)
testing_accuracy = lm.compute_accuracy(test_predict, lm.test_y.values)
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Training accuracy is: ', training_accuracy)
print('Testing accuracy is: ', testing_accuracy)
```

```
average logL for iteration 0: -0.7139731229909018
average logL for iteration 500: -0.5558529778896625
average logL for iteration 1000: -0.49990618171705625
average logL for iteration 1500: -0.4766183024021117
average logL for iteration 2000: -0.46678224234711174
average logL for iteration 2500: -0.4627302012028891
average logL for iteration 3000: -0.4611124973709076
average logL for iteration 3500: -0.46048365917256584
average logL for iteration 4000: -0.46024395049229505
average logL for iteration 4500: -0.4601537801568773
average logL for iteration 5000: -0.4601201520274992
average logL for iteration 5500: -0.46010767881536613
average logL for iteration 6000: -0.4601030679263344
average logL for iteration 6500: -0.46010136699582815
average logL for iteration 7000: -0.46010074033149045
average logL for iteration 7500: -0.46010050963239185
average logL for iteration 8000: -0.4601004247433812
average logL for iteration 8500: -0.4601003935162356
average logL for iteration 9000: -0.460100382031068
average logL for iteration 9500: -0.46010037780733604
Training avgLogL:  -0.4601003762559442
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:

1. They are the same with a very small difference in their precisions. And it makes sense as they both have the same optimal point;both algorithms are guarenteeed to converge.
2.

$$\beta_{new} = \beta_{old} + \frac{\partial J(\beta)}{\partial \beta_j}$$
$$\frac{\partial J(\beta)}{\partial \beta_j} = (XY - X\sigma(\beta) + 2\lambda\beta)$$

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 [9]:
```
from hw1code.logistic_regression import LogisticRegression

lm=LogisticRegression(verbose = False)
lm.load_data('./data/logistic-regression-toy.csv','./data/logistic-regre
ssion-toy.csv')
# As a sanity chech, we print out the size of the training data (99,2) a
nd 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 [10]:
```
training_accuracy= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#
lm.normalize()
beta = lm.train('1')
train_predict = lm.predict(lm.train_x.values, beta)
training_accuracy =  lm.compute_accuracy(train_predict,lm.train_y.values
)
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Training accuracy is: ', training_accuracy)
print('Beta: ', beta)
```

```
Training avgLogL:  -0.3291474312957121
Training accuracy is:  0.8888888888888888
Beta:  [-0.04717577  1.46005896  2.06586134]
```

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?

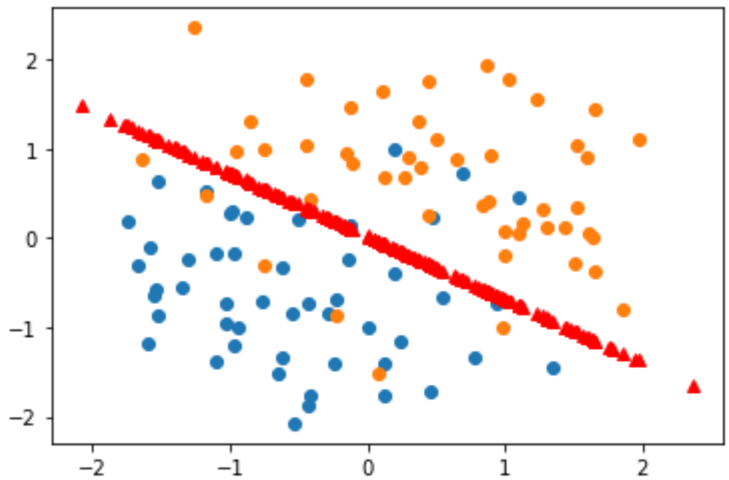
Your answer here:

Yes, because the probablity can be written as a linear function p = beta.dot(x) which is linear as we are classifying the two features and the line seperates them

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

In [14]:
```
# 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=n
ame)

# plot the decision boundary on top of the scattered points
#==========================#
# STRART YOUR CODE HERE  #
#==========================#
db = -(beta[0] + beta[1] * df[['x1','x2']]) / beta[2]
plt.plot(X, db, 'r^')
#==========================#
#   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.

```python
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  #
    #========================#
    # 1. Calculate the transpose
    # 2. first_term = (X^T*X)^ -1)
    # 3. second_term = (X^T*Y)
    # 4. calculate beta = first_term * second_term
    train_x_transpose = np.transpose(train_x)
    first_term = np.linalg.inv(np.matmul(train_x_transpose, train_x))
    second_term = np.matmul(train_x_transpose, train_y)
    beta = np.matmul(first_term, second_term)
    #========================#
    #   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)
```

```
    for iter in range(0, num_iter):
       deriv = np.zeros(p)
       for i in range(n):
           #========================#
           # STRART YOUR CODE HERE  #
           #========================#
            x_i = train_x[i]
            y_i = train_y[i]
            x_i_transpose = np.transpose(x_i)
            A = np.matmul(x_i_transpose, beta)
            deriv += x_i * (A - 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(n):
            idx = indices[i]
           #========================#
           # STRART YOUR CODE HERE  #
           #========================#
            y_i = train_y[idx]
            x_i = train_x[idx]
            x_i_transpose = np.transpose(x_i)
            A = np.matmul(x_i_transpose, beta)
            beta = beta + (lr * (y_i - A) * x_i)
           #========================#
           #   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.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
        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'):
            self.lr = 0.0005  #lr to 0.0005 so that the beta does converge
            beta = getBetaStochasticGradient(newTrain_x, self.train_y.values, self.lr)
            #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 of all test points
    # Outputs the predicted y values to the text file named "logistic-regression-output_algType_isNormalized" inside "output" folder
    # Computes MSE
    def predict(self,x, beta):
        newTest_x = addAllOneColumn(x)
        self.predicted_y = newTest_x.dot(beta)
        return self.predicted_y


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

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

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[:,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):

        x_transpose = np.transpose(X[i])
        x_transpose_dot_beta = np.dot(x_transpose, beta)
        first_term = y[i] * x_transpose_dot_beta
        second_term = 1 + np.exp(x_transpose_dot_beta)
        avglogL += first_term - np.log(second_term)

    avglogL = 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


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

            beta_transpose_dot_x = np.dot(np.transpose(beta), train_x[i])
            sigmoid_res = sigmoid(beta_transpose_dot_x)
            diff = train_y[i] - sigmoid_res
            gradient = np.dot(diff, train_x[i])
            beta +=  gradient * lr
        #========================#
        #   END YOUR CODE HERE   #
        #========================#
        if(verbose == True and iter % 1000 == 0):
            avgLogL = compute_avglogL(train_x, train_y, beta)
            print(f'average logL for iteration {iter}: {avgLogL} \t')
    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

    beta = np.random.rand(p)
    ########## Please Fill Missing Lines Here ##########
    for iter in range(0, num_iter):
        #========================#
        # STRART YOUR CODE HERE  #
        #========================#
        beta_XT = np.dot(beta, np.transpose(train_x))
        sigmoid_res = sigmoid(beta_XT)
        diff = train_y - sigmoid_res
        # first deriv
        first_deriv = np.dot(diff, train_x)
        # second deriv
        prob_mul = sigmoid_res * (1 - sigmoid_res)
        x_mul = np.array([x*y for (x,y) in zip(train_x, prob_mul)])
        second_deriv = -1 * np.dot(np.transpose(x_mul), train_x)
        beta -= np.dot(np.linalg.inv(second_deriv), first_deriv)/n
        #========================#
        #   END YOUR CODE HERE   #
        #========================#
        if(verbose == True and iter % 500 == 0):
            avgLogL = compute_avglogL(train_x, train_y, beta)
            print(f'average logL for iteration {iter}: {avgLogL} \t')
    return beta



# Linear Regression implementation
class LogisticRegression(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 -  batch gradient, 1 - Newton-Raphson)
    # Performs z-score normalization if isNormalized is 1
    # Print intermediate 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
        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

    # Predicts the y values of all test points
    # Outputs the predicted y values to the text file named "logistic-regression-output_algType_isNormalized" inside "output" folder
```

```
    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 prints the accuracy
    def compute_accuracy(self,predicted_y, y):
        acc = np.sum(predicted_y == y)/predicted_y.shape[0]
        return acc

```