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: Kevin Li, UID: 405200619

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 `START/END YOUR CODE HERE`), you need to highlight these changes. You may add some additional cells to help explain your results and observations.

In [19]:
```
import numpy as np
import pandas as pd
import sys
import random as rd
import matplotlib.pyplot as plt
%load_ext autoreload
%autoreload 2
```

```
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 [20]:
```
from hw1code.linear_regression import LinearRegression

lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.
# As a sanity check, we print out the size of the training data (1000, 100) and t
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 [21]:
```
from hw1code.linear_regression import LinearRegression

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

beta = lm.train('0')
y_train_hat = lm.predict(lm.train_x, beta)
y_test_hat = lm.predict(lm.test_x, beta)

training_error = lm.compute_mse(y_train_hat, lm.train_y)
testing_error = lm.compute_mse(y_test_hat, 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:  0.08693886675396781
Testing error is:  0.11017540281675806
```

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 [22]:
```
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.
training_error= 0
testing_error= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#

beta = lm.train('1')
y_train_hat = lm.predict(lm.train_x, beta)
y_test_hat = lm.predict(lm.test_x, beta)

training_error = lm.compute_mse(y_train_hat, lm.train_y)
testing_error = lm.compute_mse(y_test_hat, 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:  0.08693902047792586
Testing accuracy is:  0.11018845441523527
```

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 [23]:
```python
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.
training_error= 0
testing_error= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#

lm.lr = 0.0005
beta = lm.train('2')
y_train_hat = lm.predict(lm.train_x, beta)
y_test_hat = lm.predict(lm.test_x, beta)

training_error = lm.compute_mse(y_train_hat, lm.train_y)
testing_error = lm.compute_mse(y_test_hat, 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:  0.09531517836589815
Testing accuracy is:  0.11959236638904801
```

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:

In [24]:
```
# Running normalized versions
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.
print(f'Feature average std: {lm.train_x.describe().loc["std"].mean()}\n')
lm.normalize()

def run_alg(a):
    beta = lm.train(a)
    training_error = lm.compute_mse(lm.predict(lm.train_x, beta), lm.train_y)
    testing_error = lm.compute_mse(lm.predict(lm.test_x, beta), lm.test_y)
    print('Training error is: ', training_error)
    print('Testing error is: ', testing_error, '\n')

for a in ('0', '1', '2'):
    run_alg(a)
```

```
Feature average std: 2.3522292509877913

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

Learning Algorithm Type:  1
Training error is:  0.10018132194244624
Testing error is:  0.13900607116647945

Learning Algorithm Type:  2
Training error is:  0.10992817207625191
Testing error is:  0.13117156993305695
```

1. The MSE is slightly different but close for each version. They're different because they each take different paths to optimizing beta, but they all end up in a similar place because (it seems like) this problem is probably convex i.e. only has one optimum solution. To be precise, the first closed form solution should actually get the beta with the minimum error while BGD and SGD simply run for a certain number of iterations, then stop wherever they are. Since they involve choosing a random beta and SGD randomly samples from the training dataset, they will perform slightly randomly (although asymptotically they should still converge on the optimal beta)
2. Z-scoring the features has no effect on the closed form solution but does slightly affect the gradient descent algorithms. This is because linear regression simply assigns a weight (β_i) to each feature; if the feature's size is scaled, then the respective weight will be scaled appropriately to compensate. The "zeroing" of the features is also accounted for due to the addition of the bias term (which we see is added via the addAllOneColumn function). For the gradient descent algorithms, since they actually use the feature values to calculate their gradients and are affected by the feature values, we see mild effects on the model. One reason this happens is because Z-scoring (normalizing) is done to make each feature "just as important" as each other to avoid quickly converging on a solution that only uses some features. For example, if the standard deviation of x_1 is massive and the standard deviation of x_2 is tiny and we had the true solution be $y = x_1 + x_2$, then we would be able to converge on a "pretty good" solution by simply guessing $y = x_1$ i.e. $b_1 = 1, b_2 = 0$. Note that this does not neccessarily mean normalization improves performance! In fact in our case, it seems

to hurt performance (although it would be nice to do something more like k-fold validation to investigate more thoroughly). It also helps with avoiding overflow which we saw affect SGD (previously without normalization, we needed to reduce the learning rate to avoid overflow).

3. The solution ends up being $\beta = (X^T X + \lambda I)^{-1} X^T y$

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

We know what the left side (original loss) turns into since it's the same as the original loss function $J(\beta)$, so we get the following

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

Taking the derviative of the right side, we get

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

We set the derviative equal to zero and do some rearranging and get

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

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

2. Logistic regression

This workbook will walk you through a logistic regression example.

In [25]:
```
from hw1code.logistic_regression import LogisticRegression

lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-t
# As a sanity chech, we print out the size of the training data (1000, 5) and tra
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 [26]:
```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-t
training_accuracy= 0
testing_accuracy= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#

lm.normalize()

beta = lm.train('0')
y_train_hat = lm.predict(lm.train_x, beta)
y_test_hat = lm.predict(lm.test_x, beta)

training_accuracy = lm.compute_accuracy(y_train_hat, lm.train_y)
testing_accuracy = lm.compute_accuracy(y_test_hat, lm.test_y)

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

```
average logL for iteration 0: -0.4893882425713696
average logL for iteration 1000: -0.460100375350853
average logL for iteration 2000: -0.460100375350853
average logL for iteration 3000: -0.460100375350853
average logL for iteration 4000: -0.460100375350853
average logL for iteration 5000: -0.460100375350853
average logL for iteration 6000: -0.460100375350853
average logL for iteration 7000: -0.460100375350853
average logL for iteration 8000: -0.460100375350853
average logL for iteration 9000: -0.460100375350853
Training avgLogL:  -0.460100375350853
Training accuracy is:  0.797
Testing accuracy is:  0.7534791252485089
```

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 [27]:
```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regression-t
training_accuracy= 0
testing_accuracy= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#

lm.normalize()

beta = lm.train('1')
y_train_hat = lm.predict(lm.train_x, beta)
y_test_hat = lm.predict(lm.test_x, beta)

training_accuracy = lm.compute_accuracy(y_train_hat, lm.train_y)
testing_accuracy = lm.compute_accuracy(y_test_hat, lm.test_y)

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

```
average logL for iteration 0: -0.4905626329298569
average logL for iteration 500: -0.460100375350853
average logL for iteration 1000: -0.460100375350853
average logL for iteration 1500: -0.460100375350853
average logL for iteration 2000: -0.460100375350853
average logL for iteration 2500: -0.460100375350853
average logL for iteration 3000: -0.460100375350853
average logL for iteration 3500: -0.460100375350853
average logL for iteration 4000: -0.460100375350853
average logL for iteration 4500: -0.460100375350853
average logL for iteration 5000: -0.460100375350853
average logL for iteration 5500: -0.460100375350853
average logL for iteration 6000: -0.460100375350853
average logL for iteration 6500: -0.460100375350853
average logL for iteration 7000: -0.460100375350853
average logL for iteration 7500: -0.460100375350853
average logL for iteration 8000: -0.460100375350853
average logL for iteration 8500: -0.460100375350853
average logL for iteration 9000: -0.460100375350853
average logL for iteration 9500: -0.460100375350853
Training avgLogL:  -0.460100375350853
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:

1. They are the same since the both converged on the same (probably very close to optimal) solution. Newton-Raphson is simply a different optimizer and can therefore have different properties (i.e. speed, ability to find global optimum, precision, etc) but if they both find the global optimum, they will obviously have the same results
2. The batch gradient update ends up being $\beta_j^{t+1} = B_j^t + \eta\left(-2\lambda\beta_j + \frac{1}{n}\sum_i x_{ij}(c_i - p_i(\beta))\right)$

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

Like before, we know the derivative of the left part of the equation since it's the regular log likelihood loss function and we covered the derivative in class, so we gradient

$$\frac{\partial J}{\partial \beta_j} = -\frac{1}{n}\sum_{i=1}^n x_{ij}(y_i - p_i(\beta)) + \frac{\partial}{\partial \beta_j}\lambda\sum_j \beta_j^2$$
$$\frac{\partial J}{\partial \beta_j} = -\frac{1}{n}\sum_{i=1}^n x_{ij}(y_i - p_i(\beta)) + 2\lambda\beta_j$$

Now that we know the derivative, gradient descent becomes as simple as

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

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 [29]:
```
from hw1code.logistic_regression import LogisticRegression

lm=LogisticRegression(verbose = False)
lm.load_data('./data/logistic-regression-toy.csv','./data/logistic-regression-toy
# As a sanity chech, we print out the size of the training data (99,2) and traini
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 [30]:
```
training_accuracy= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#

lm.normalize()
beta = lm.train('1')
y_train_hat = lm.predict(lm.train_x, beta)
y_test_hat = lm.predict(lm.test_x, beta)

training_accuracy = lm.compute_accuracy(y_train_hat, lm.train_y)
testing_accuracy = lm.compute_accuracy(y_test_hat, lm.test_y)

print(f'Beta: {beta}')

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

```
Training avgLogL:  -0.329147431295712
Beta: [-0.04717577  1.46005896  2.06586134]
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:

Yes.

We know the decision boundary for logistic regression is when $\sigma(x^T\beta) = 0.5$. Then, we know that

$$\sigma(x^T\beta) = 0.5$$
$$\frac{1}{1+e^{-x^T\beta}} = 0.5$$
$$1 = e^{-x^T\beta}$$
$$0 = -x^T\beta$$

This is the equation for a hyperplane - the decision boundary is linear

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)).

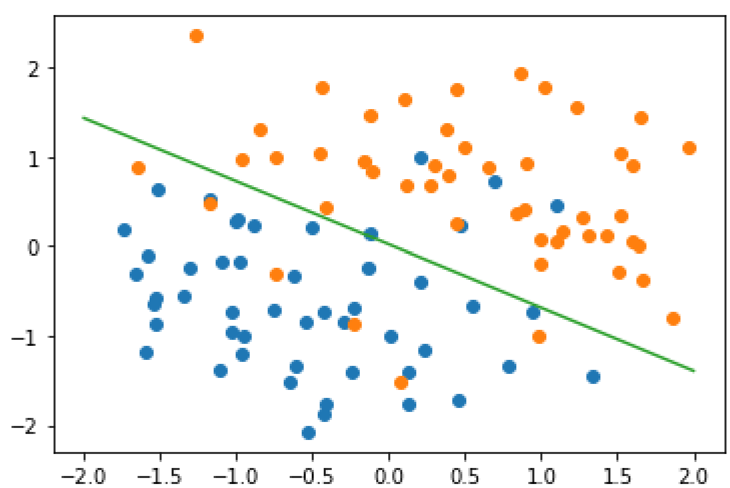
In [35]:
```
# scatter plot the raw data
df = pd.concat([lm.train_x, lm.train_y], axis=1)
groups = df.groupby("y")
for name, group in groups:
    plt.plot(group["x1"], group["x2"], marker="o", linestyle="", label=name)

# plot the decision boundary on top of the scattered points
#========================#
# STRART YOUR CODE HERE  #
#========================#

x = np.array([-2, 2])
y = (beta[0] + beta[1] * x) / -beta[2]

plt.plot(x, y)

#========================#
#   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  #
    #========================#
    beta = np.linalg.inv(train_x.T @ train_x) @ (train_x.T @ 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 += train_x[i] * (train_x[i] @ 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(n):
            idx = indices[i]
           #========================#
           # STRART YOUR CODE HERE  #
           #========================#
            beta += lr * train_x[idx] * (train_y[idx] - train_x[idx] @ beta)
           #========================#
           #   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)
```

```python

    # 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'):
            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 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



```

```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  #
    #========================#
    avglogL = np.sum(y * (X @ beta) - np.log(1 + np.exp(X @ beta))) / 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):
        #========================#
```

```
        # START YOUR CODE HERE  #
        #=======================#

        beta += lr * (train_x.T @ (train_y - sigmoid(train_x @ beta)))

        #=======================#
        #   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)
    for iter in range(0, num_iter):
        #=======================#
        # START YOUR CODE HERE  #
        #=======================#

        y_pred = sigmoid(train_x @ beta)
        grad = train_x.T @ (train_y - sigmoid(train_x @ beta))

        neg_hessian = train_x.T @ np.diag(y_pred * (1 - y_pred)) @ train_x
        beta += np.linalg.inv(neg_hessian) @ grad

        #=======================#
        #   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



# 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
        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 = np.sum(predicted_y == y)/predicted_y.shape[0]
        return acc
```