CS145 Howework 1

Important Note: The submission deadline for all homeworks are one week from its release date. HW1 is due on **11:59 PM PT, April 19 (Wednesday)**. Please submit through GradeScope (you will receive an invitation for CS145 Spring 2023).

Before You Start

You need to first create HW1 conda environment using `cs145hw1.yml`. This file provides the env name and necessary packages for this tasks. If you have `conda` installed, you may create, activate and deactivate an environment using the following commands:

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

Here are some references about conda: conda.

You should 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 will need to highlight these changes. You may add additional cells to help explain your results and observations.

In [37]:
```
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 example will walk you through three optimization algorithms for linear regression.

In [38]:
```
from hw1code.linear_regression import LinearRegression

lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-
# As a sanity check, we print out the size of the training data (1000, 100)
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 compute the close form solution of $\hat{\beta}$.

To train you modelm use `lm.train('0')` function.

Compute the training error and the testing error using `lm.predict` and `lm.compute_mse`.

In [39]:
```
from hw1code.linear_regression import LinearRegression

lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-
training_error= 0
testing_error= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
## hint, for training error, you should get something around 0.08

beta = lm.train('0')
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)

#=======================#
#   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:  0.11017540281675801
```

1.2.Batch gradient descent

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

To train you model, use `lm.train('1')` function.

Compute the training error and the testing error using `lm.predict` and `lm.compute_mse`.

In [40]:
```
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-
training_error= 0
testing_error= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#

beta = lm.train('1')
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)

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

```
Learning Algorithm Type:  1
Training error is:  0.08694107160099752
Testing error is:  0.11012995921188702
```

1.3.Stochastic gadient descent

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

To train you model, use `lm.train('2')` function.

Compute the training error and the testing error using `lm.predict` and `lm.compute_mse`.

In [41]:
```
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-
training_error= 0
testing_error= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#

beta = lm.train('2')
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)

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

```
Learning Algorithm Type:  2
Training error is:  0.08693897185738789
Testing error is:  0.11018508040566348
```

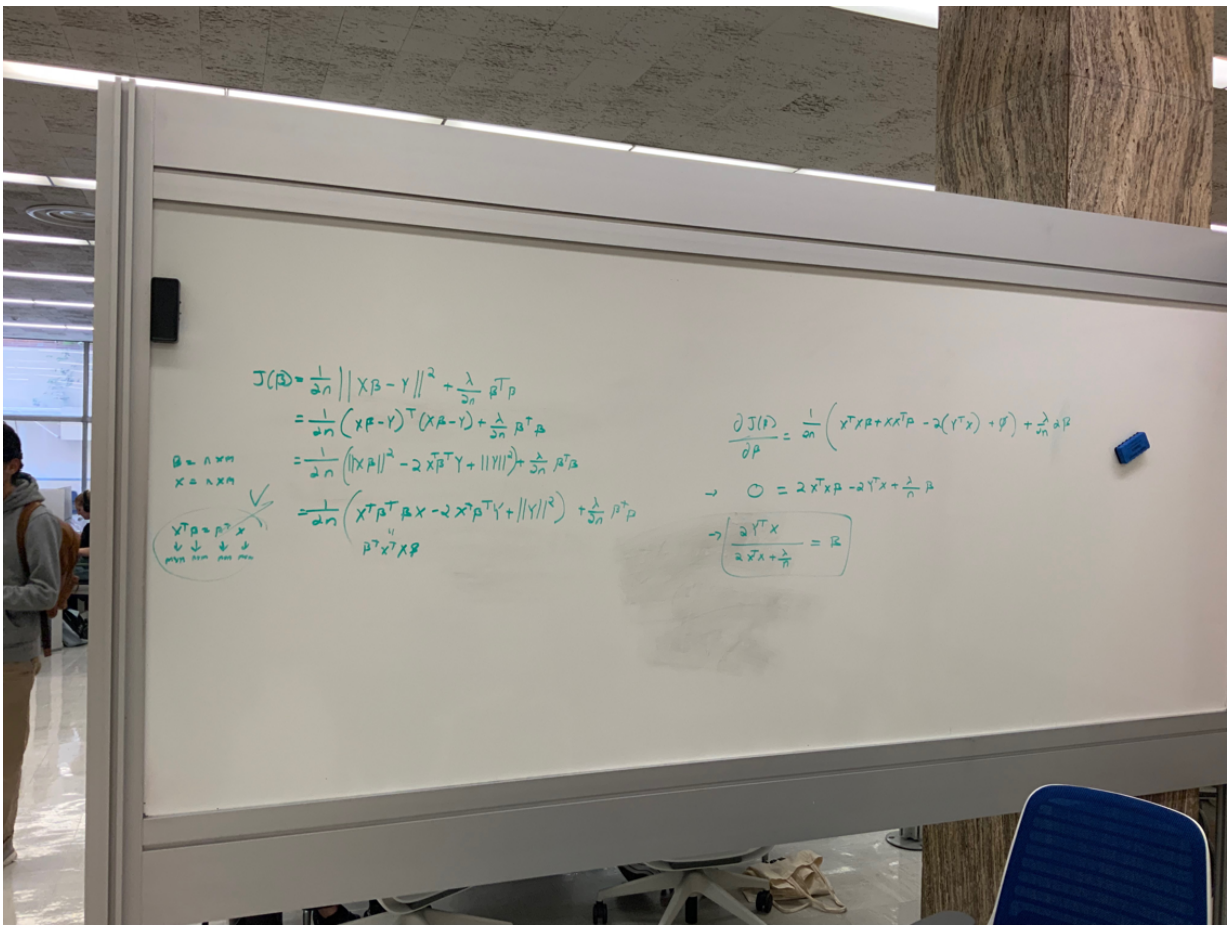
Questions:

1. Ridge regression adds an L2 regularization term to the original objective function. The objective function becomes the following:

$$J(\beta) = \frac{1}{2n}||X\beta - Y||^2 + \frac{\lambda}{2n}\beta^T\beta,$$

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

Your answer here:



$$\beta = 2Y^T X * (2X^T X + \frac{\lambda}{n})^{-1}$$

2. Logistic regression

This example will walk you through algorithms for logistic regression

In [42]:
```
from hw1code.logistic_regression import LogisticRegression

lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regress
# As a sanity chech, we print out the size of the training data (1000, 5) ar
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 computes the gradient of the log likelihoood function.

Complete the `compute_avglogL` function in `logistic_regression.py` for sanity check, you should get something around 0.46.

To train you model, use `lm.train('0')` function.

Compute the training and testing accuracy using `lm.predict` and `lm.compute_accuracy`.

In [43]:
```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regress
training_accuracy= 0
testing_accuracy= 0
lm.normalize()  # apply z-score normalization to the data
#=======================#
# STRART YOUR CODE HERE  #
#=======================#

beta = lm.train('0')
training_accuracy = lm.compute_accuracy(lm.predict(lm.train_x, beta), lm.tra
testing_accuracy = lm.compute_accuracy(lm.predict(lm.test_x, beta), lm.test_

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

```
average logL for iteration 0: 0.49356102428226734
average logL for iteration 1000: 0.4601003753508533
average logL for iteration 2000: 0.4601003753508533
average logL for iteration 3000: 0.4601003753508533
average logL for iteration 4000: 0.4601003753508533
average logL for iteration 5000: 0.4601003753508533
average logL for iteration 6000: 0.4601003753508533
average logL for iteration 7000: 0.4601003753508533
average logL for iteration 8000: 0.4601003753508533
average logL for iteration 9000: 0.4601003753508533
Training avgLogL:  0.4601003753508533
Training accuracy is:  0.797
Testing accuracy is:  0.7534791252485089
```

2.2 Newton Raphhson

In this section, complete the `getBeta_Newton` function in `logistic_regression.py`, which makes use of both first and second derivatives.

To train you model, use `lm.train('1')` function.

Compute the training and testing accuracy using `lm.predict` and `lm.compute_accuracy`.

In [45]:
```
lm=LogisticRegression()
lm.load_data('./data/logistic-regression-train.csv','./data/logistic-regress
training_accuracy= 0
testing_accuracy= 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
lm.normalize()
beta = lm.train('1')
training_accuracy = lm.compute_accuracy(lm.predict(lm.train_x, beta), lm.tra
testing_accuracy = lm.compute_accuracy(lm.predict(lm.test_x, beta), lm.test_

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

```
average logL for iteration 0: 0.4773881437850127
average logL for iteration 500: 0.4601003753508532
average logL for iteration 1000: 0.4601003753508532
average logL for iteration 1500: 0.46010037535085313
average logL for iteration 2000: 0.4601003753508532
average logL for iteration 2500: 0.46010037535085313
average logL for iteration 3000: 0.46010037535085313
average logL for iteration 3500: 0.4601003753508533
average logL for iteration 4000: 0.4601003753508532
average logL for iteration 4500: 0.4601003753508532
average logL for iteration 5000: 0.4601003753508533
average logL for iteration 5500: 0.46010037535085324
average logL for iteration 6000: 0.46010037535085313
average logL for iteration 6500: 0.4601003753508533
average logL for iteration 7000: 0.4601003753508533
average logL for iteration 7500: 0.4601003753508532
average logL for iteration 8000: 0.4601003753508533
average logL for iteration 8500: 0.4601003753508532
average logL for iteration 9000: 0.4601003753508533
average logL for iteration 9500: 0.4601003753508533
Training avgLogL:  0.46010037535085324
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?

Your answer here:

The answers are the same, in terms of accuracy; however, the average log loss differs for every 1000 iterations because of the way the algorithms try to reach a local minimum. Batch gradient descent does it in small incrementations, whereas Newton will require far fewer steps for convergence because it takes bigger steps.

2.3 Visualize the decision boundary on a toy dataset

In this subsection, you will use the above implementation for another small dataset where each datapoint x only has 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-regressic
# As a sanity chech, we print out the size of the training data (99,2) and t
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
lm.normalize()
#========================#
# STRART YOUR CODE HERE  #
#========================#
beta = lm.train('0')
training_accuracy = lm.compute_accuracy(lm.predict(lm.train_x, beta), lm.tra

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

```
Training avgLogL:  0.329147431295712
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. We know that the decision boundary lies where the probability of set x is 0.5.

using the sigmoid

$$\frac{1}{1+e^{-\theta x}} = 0.5$$

doing some algebra

$$1 = e^{-\theta x}$$

taking the log of both sides

$$-\theta x = 0$$

which 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).

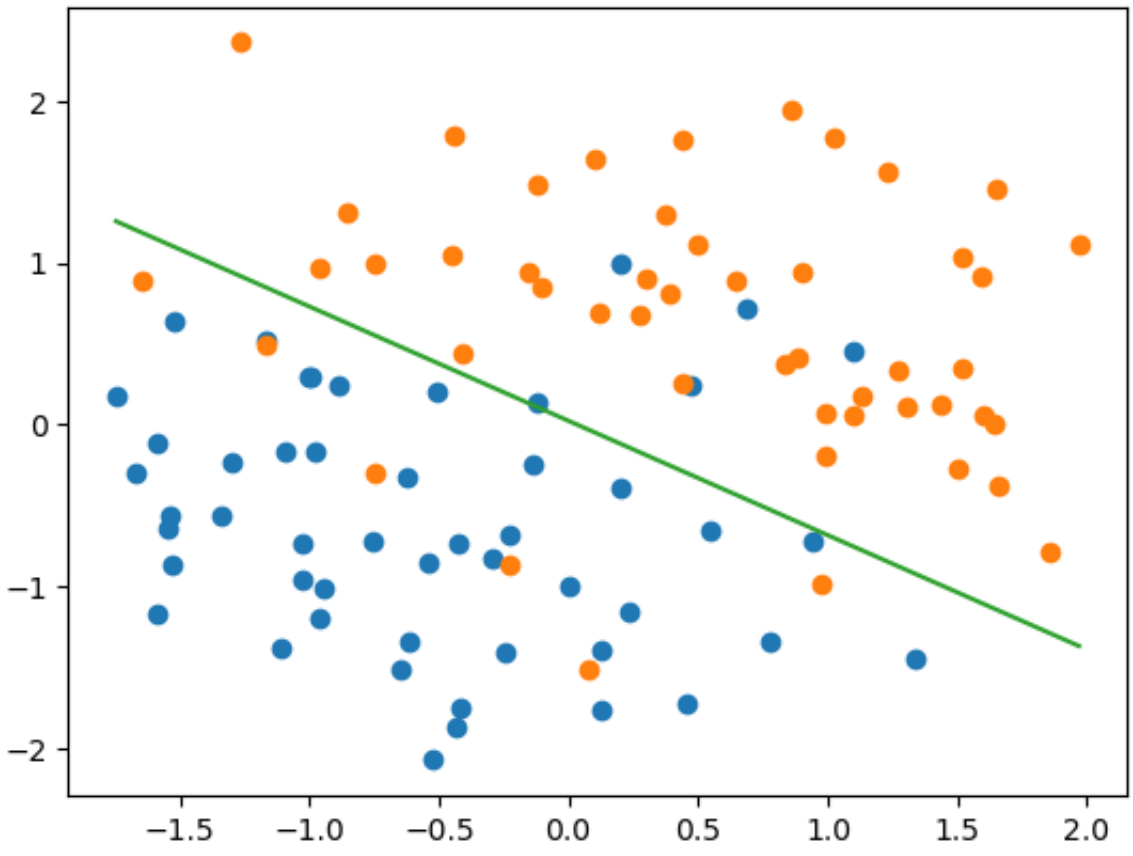
In [29]:
```
# scatter plot the raw data
df = pd.concat([lm.train_x, lm.train_y], axis=1)
groups = df.groupby("y")
for name, group in groups:
    plt.plot(group["x1"], group["x2"], marker="o", linestyle="", label=name)

# plot the decision boundary on top of the scattered points
x1_vec = np.linspace(lm.train_x["x1"].min(),lm.train_x["x1"].max(),2)  ## x
#========================#
# STRART YOUR CODE HERE  #
#========================#

x2 = -(beta[0] + beta[1]*x1_vec) / beta[2]
plt.plot(x1_vec, x2)


# z = beta[0] + beta[1]x1 + beta[2]x2
# x2 = (z - beta[0] - beta[1]x1) / beta[2]
# z = 0 => probability is 0.5 which is boundary

#========================#
#   END YOUR CODE HERE   #
#========================#
plt.show()
```



End of Homework 1