# 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](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.

```
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 example will walk you through three optimization algorithms for linear regression.

```
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 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`.

```
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
#========================#
# START YOUR CODE HERE  #
#========================#
## hint, for training error, you should get something around 0.08
lm.normalize()
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
getBeta p: 101
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`.

```
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.csv')
training_error= 0
testing_error= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#
lm.normalize()
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:  0.09863293671476969
Testing error is:  0.13408550691262822
```

## 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`.

```
lm=LinearRegression()
lm.load_data('./data/linear-regression-train.csv','./data/linear-regression-test.csv')
training_error= 0
testing_error= 0
#========================#
# STRART YOUR CODE HERE  #
#========================#
lm.normalize()
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:  0.09918731854480163
Testing error is:  0.1337090695005499
```

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 $\beta$.

## Your answer here:

https://drive.google.com/file/d/1HVBWWUYsW8VAhC9__33toGDcJxoUxCcJ/view?usp=share_link

# 2. Logistic regression

This example will walk you through algorithms for logistic regression

```
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 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`.

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

train_pred = lm.predict(lm.train_x, beta)
training_error = lm.compute_accuracy(train_pred, lm.train_y)
test_pred = lm.predict(lm.test_x, beta)
testing_error = lm.compute_accuracy(test_pred, lm.test_y)
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Training accuracy is: ', training_accuracy)
print('Testing accuracy is: ', testing_accuracy)
#logl vals are similar to those on piazza but I dont understand why the accuracies are 0
```

```
beta shape is: (6,)
x shape is: (1000, 6)
y shape is: (1000,)
average logL for iteration 0: -0.5392168671552291
average logL for iteration 1000: -0.46010037535085313
average logL for iteration 2000: -0.46010037535085313
```

```
average logL for iteration 3000: -0.46010037535085313
average logL for iteration 4000: -0.46010037535085313
average logL for iteration 5000: -0.46010037535085313
average logL for iteration 6000: -0.46010037535085313
average logL for iteration 7000: -0.46010037535085313
average logL for iteration 8000: -0.46010037535085313
average logL for iteration 9000: -0.46010037535085313
Training avgLogL:  -0.4601003753508529
Training accuracy is:  0
Testing accuracy is:  0
```

## 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`.

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

train_pred = lm.predict(lm.train_x, beta)
training_error = lm.compute_accuracy(train_pred, lm.train_y)
test_pred = lm.predict(lm.test_x, beta)
testing_error = lm.compute_accuracy(test_pred, lm.test_y)
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Training accuracy is: ', training_accuracy)
print('Testing accuracy is: ', testing_accuracy)
```

```
average logL for iteration 0: -inf
/home/josh/CS145/hw1_2023/hw1code/logistic_regression.py:41: RuntimeWarning: overflow encountered in exp
  avglogL = avglogL + y[iter]*(xit @ beta) - np.log(1 + np.exp(xit @ beta))
/home/josh/CS145/hw1_2023/hw1code/logistic_regression.py:29: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-z))
average logL for iteration 500: -inf
average logL for iteration 1000: -inf
average logL for iteration 1500: -inf
average logL for iteration 2000: -inf
average logL for iteration 2500: -inf
average logL for iteration 3000: -inf
average logL for iteration 3500: -inf
average logL for iteration 4000: -inf
average logL for iteration 4500: -inf
average logL for iteration 5000: -inf
average logL for iteration 5500: -inf
average logL for iteration 6000: -inf
average logL for iteration 6500: -inf
average logL for iteration 7000: -inf
average logL for iteration 7500: -inf
average logL for iteration 8000: -inf
average logL for iteration 8500: -inf
average logL for iteration 9000: -inf
average logL for iteration 9500: -inf
Training avgLogL:  -inf
Training accuracy is:  0
Testing accuracy is:  0
```

### Questions:

1. Compare the accuracy on the testing dataset for each version. Are they the same? Why or why not?

## Your answer here:

I could not get the functions to work =(

## 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.

```
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.

```
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.train_y)
print(beta)
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Training accuracy is: ', training_accuracy)
#why does this work here but not in the previous cells?
```

```
beta shape is: (3,)
x shape is: (99, 3)
y shape is: (99,)
Training avgLogL:  -0.3291474312957121
[-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?

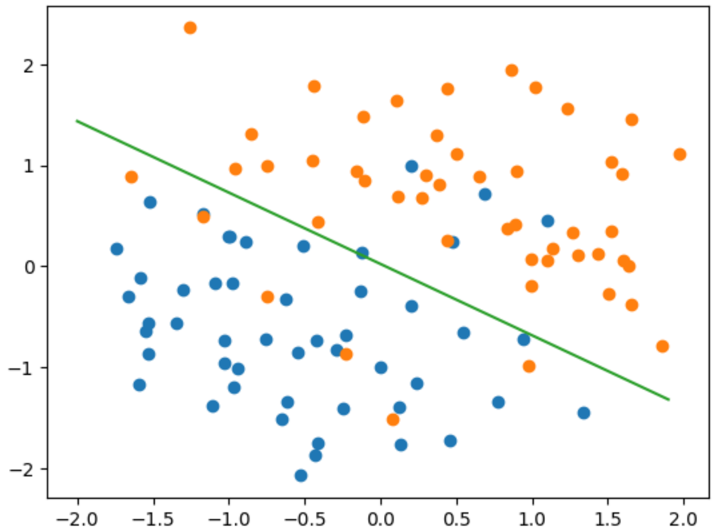
## Your answer here:

Yes as the decision boundary where the sigmoid function is equal to .5. This occurs when the input to the sigmoid function is 0 as e^0 is one so 1/(1+e^0) is 1/2.

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

```
# 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 axis of the boundary is also given
#========================#
# STRART YOUR CODE HERE  #
#========================#
x1 = np.arange(-2, 2, 0.1)
#y = b0 +b1x1 + b2x2
#since decision boundary at y=0, and we are iterating through x1, we find x2 for the decision boundary
x2 = -(x1*beta[1] + beta[0])/beta[2]
plt.plot(x1, x2)
#========================#
#   END YOUR CODE HERE   #
#========================#
plt.show()
```




## End of Homework 1