GENERAL EXPLANATIONS

i. First, import the necessary libraries

1. Numpy
2. Pandas
3. Matplotlib

ii. Run main.py to get the outputs

Note : Before loading the dataset, I have directly added column headers x1, x2 and y on the csv files for better understanding and viewing (in both train and test dataset).

The code starts by importing all the necessary libraries required as stated above. Then all the required functions have been defined.

The main part of code starts by loading the datasets I have added bias terms in both the training data and test data as shown below(Create a seperate ones column)

```
train_data = pd.read_csv("FMLA1Q1Data_train.csv")
test_data = pd.read_csv("FMLA1Q1Data_test.csv")
train_data["x0"] = 1
test_data["x0"] = 1
```

Next, I have splitted the features and labels as X and y for both train and test datasets In the following subdivision, explanation for each part of the question is given

PART 1

The equation for linear regression is

```
y = w0 + w1 * x1 + w2 * x2
```

and the corresponding solution for w in matrix form is

```
w* = (X * X.T) * X * Y
```

X.T = transpose of X matrix

```
def analytical_solution(X, y):
    w = np.linalg.inv(X.T @ X) @ X.T @ y
    w = np.array(w)
    w.reshape(-1,1)
    return w
```

This is the analytical solution function used as per the formula. Please note that as I have taken X as a n*d matrix, So corresponding changes have been made (I replaced X by X.T)

PART 2

CODE EXPLANATION

```
w_init = np.zeros(X_train.shape[1])
epochs = 100
error_function = np.zeros(epochs)

learning_rates = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10]
best_l = 0
weight_error = np.inf
for l in learning_rates :
    w = gradient_descent(X_train, y_train, w_init, l, epochs, w_ml,
error_function)
    if mse(w, w_ml) < weight_error :
        weight_error = mse(w, w_ml)
        best_l = l

w = gradient_descent(X_train, y_train, w_init, best_l, epochs, w_ml,
error_function)

plt.title("Gradient Descent Error Plot")
plt.xlabel("Epochs")
plt.ylabel("Error b/w analytical and predicted weights")
plt.plot(error_function)
plt.show()
```

First I have initialised weights as zero. Then , I have set epochs to 100 (I found out this is enough no. of iterations for convergence).

I have also created a error function variable to store the squared error between the weights of analytical and gradient descent solution. This will be used in plotting the graphs

To find out the best learning rate for the gradient descent, I have taken a set of values in different order as shown below

```
learning_rates = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10]
```

Next, For each of these learning rate, I have done the gradient descent algorithm. I took the best learning rate as the one which provides the most similar solution to the analytical solution. The solution which has the least error w.r.t to analytical is the best one.

Next, I have plotted the graph as required.

```
def gradient_descent (X, y, w, learning_rate, epochs, w_ml,
error_function):

    for i in range(epochs):
        gradient = (2 * (X.T @ ((X @ w) - y)))
        w = w - learning_rate * gradient
```

```
        error_function[i] = np.sum((w - w_ml) ** 2)

    return w
```

This is the gradient descent function I have used. The gradient and update function is calculated as per lectures taught in the class. I have added the error function update for plotting the graph

OBSERVATIONS

For a learning rate of = 0.0001, the gradient descent solution converges to the analytical solutions. Lower learning rates will also converge but will take more iterations so this is ideal

PART 3

EXPLANATION

```
w_init = np.zeros(X_train.shape[1])
epochs = 20
error_function= np.zeros(epochs)

w_stochastic = stochastic_gradient_descent(train_data, w_init, 0.0001,
epochs, w_ml, error_function)

plt.title("Stochiastic Gradient Descent Error Plot")
plt.xlabel("Epochs")
plt.ylabel("Error b/w analytical and predicted weights")
plt.plot(error_function)
plt.show()
```

This is similar code as the previous gradient descent one. We'll move on to the explanation of the function which is commmented

```
def stochastic_gradient_descent(data, w, learning_rate, epochs, w_ml,
error_function):

    for epoch in range(epochs):

        # Shuffle the training data for each epoch
        data.sample(frac=1).reset_index(inplace=True, drop=True)

        no_batches = 10 # Each batch has 100 data points

        # Iterate over each batch
        for i in range(0, no_batches):

            st_idx = i*int(len(data)/no_batches)
            end_idx = (i+1)*int(len(data)/no_batches)

            # Separate the features (X_rand) and target (y_rand)
```

```
            X_rand = data.iloc[st_idx:end_idx].drop("y", axis=1)
            y_rand = data.iloc[st_idx:end_idx]["y"]

            # Calculate the gradient
            gradient = 2 * (X_rand.T @ ((X_rand @ w) - y_rand))

            # Update the weights
            w = w - learning_rate * gradient

            # Calculate and store the error for the current epoch

        error_function[epoch] = np.sum((w - w_ml) ** 2)

    return w
```

OBSERVATIONS

The observarion is that the stochastic method converges faster than the normal gradient descent. In my case, The number of iterations it took to converge is approximaately the same but it might be same for a very large dataset

PART 4

EXPLANATION

The code for the ridge regression function is similar to gradient descent with a small addition of regularizer. This is given below

```
def ridge_regresssion(X, y, w, learning_rate, epochs, lamda):
    for i in range(epochs):
        gradient = (2 * (X.T @ ((X @ w) - y)) + 2 * lamda* w) / len(y)
        w = w - learning_rate * gradient

    return w
```

For cross valiation, I have splitted the train data into train and validation(80-20 split), then found the best lambda value

```
lamda = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100]
best_lamda = 0

lambda_error = np.inf
y_val_pred_ml = X_val_split @ w_ml
for lam in lamda :
    w_ridge = ridge_regresssion(X_train_split, y_train_split, w_init,
0.0001, epochs, lam)
    y_val_pred_ridge = X_val_split @ w_ridge
    if mse(y_val_pred_ml, y_val_pred_ridge) < lambda_error:
```

```
        lambda_error = mse(y_val_pred_ml, y_val_pred_ridge)
        best_lamda = lam
```

Here, I have found the error between predicted y values from analytical and ridge solutions. The best lambda is the one which produces the least error here.

Then I have printed the Ridge and Analytical error. I found out the ridge error is more.

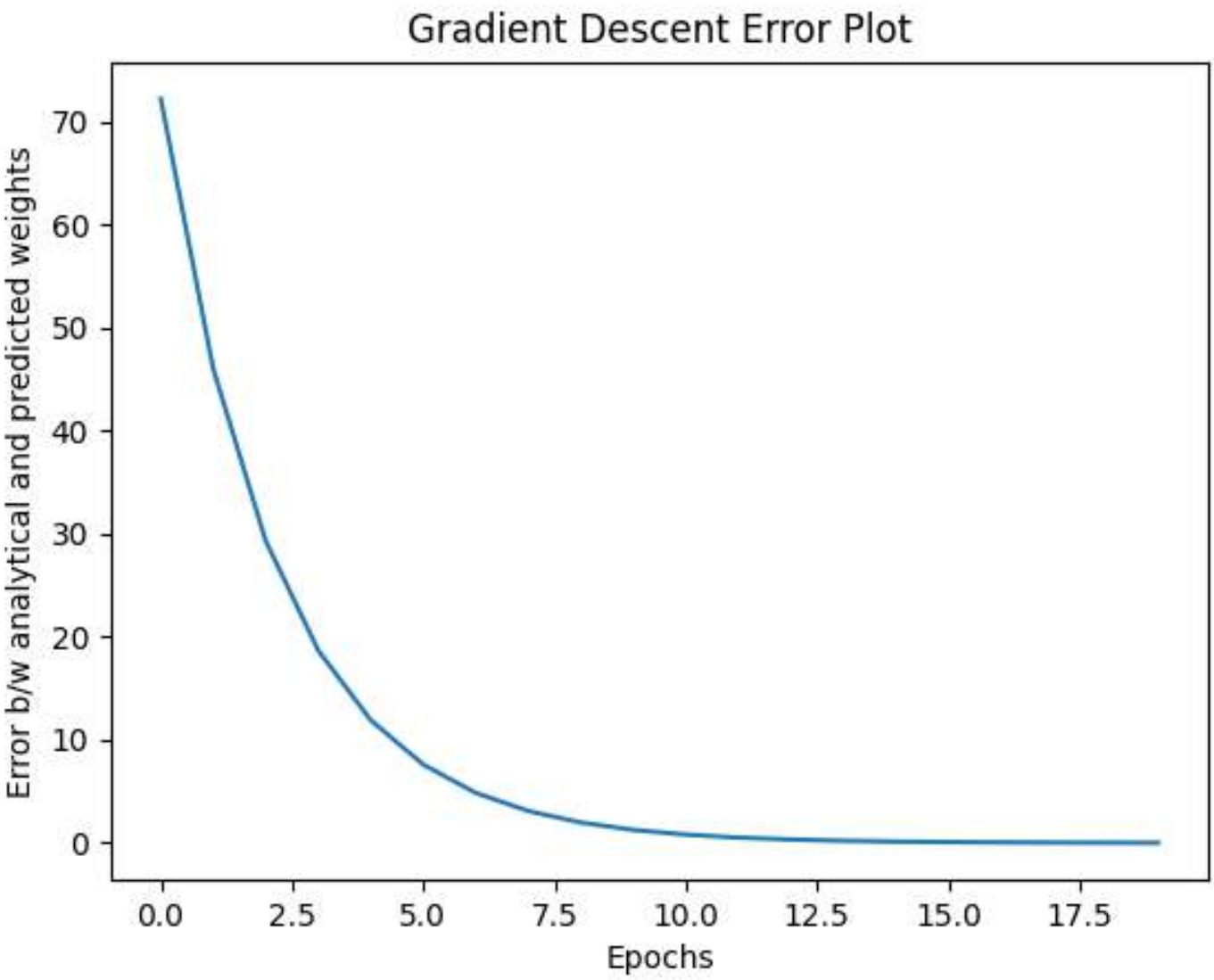
PART 5

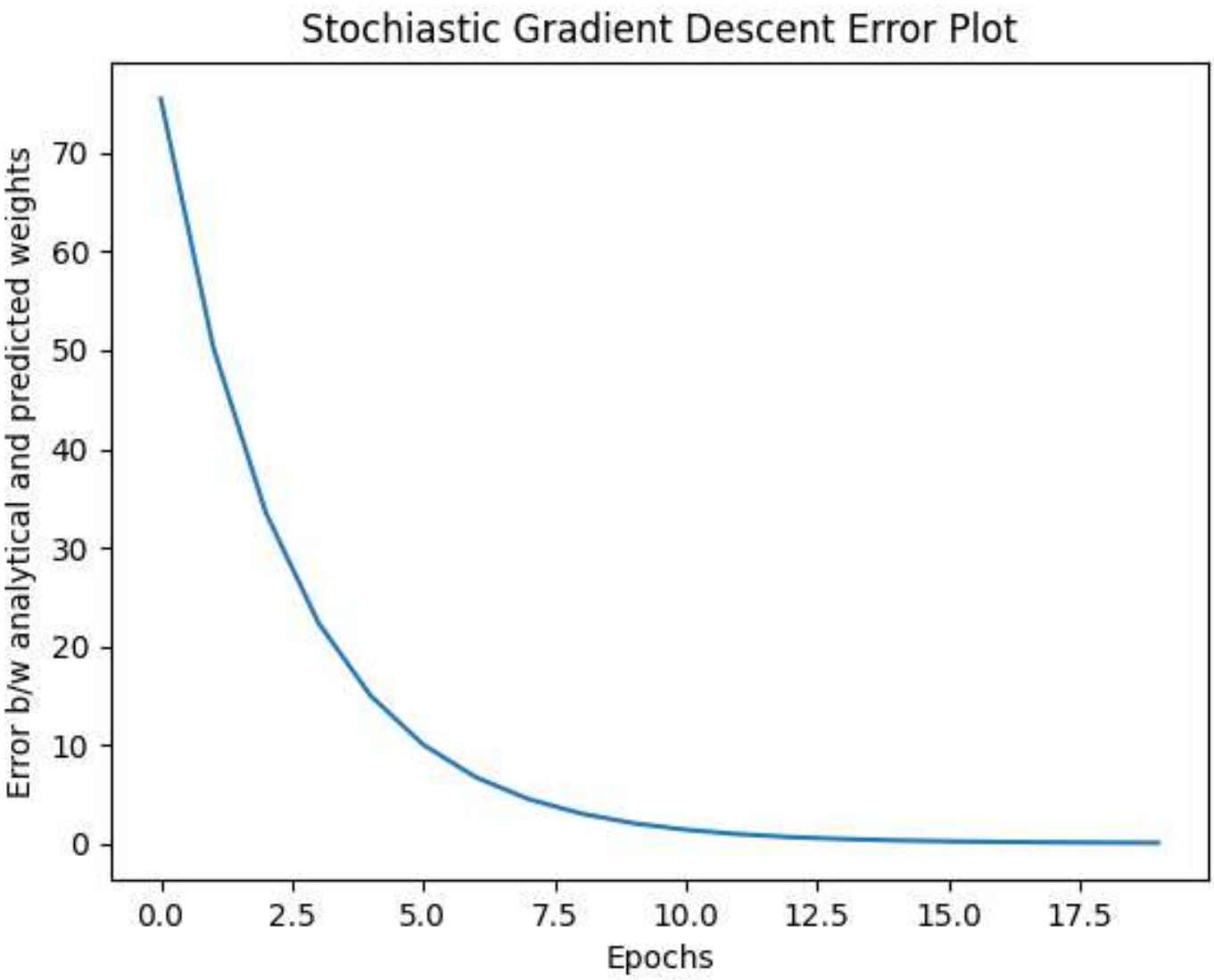
I have did gaussian kernel regression as it works even in non linear data set . And when I tried this here, I got a very low test error of 0.98. So, this kernel regression is better than other types of regression.

```
def guassian_kernel(x, xi, mu):
    z = np.linalg.norm(xi- x)/mu
    return (np.exp((-z**2)/2))/(np.sqrt(2*np.pi))

def kernel_regression(X_train, X_test, mu):
    weights = []
    for i, xi in X_test.iterrows():
        kernels = [guassian_kernel(x, xi, mu) for idx, x in
X_train.iterrows()]
        sim_score = [X_train.shape[0]*kernel/np.sum(kernels) for
kernel in kernels]
        weights.append(sim_score)
    weights = np.array(weights)
    return weights
```

First I have defined the gaussian kernel function (basic) and then defined the kernel regression function


Gradient Descent Error Plot
Error b/w analytical and predicted weights
70
60
50
40
30
20
10
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


Stochiastic Gradient Descent Error Plot
Error b/w analytical and predicted weights
70
60
50
40
30
20
10
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs