Assignment 1 – Linear Regression via Gradient Descent

Student number: 20324412

Module: CS7CS4/CSU44061

Part a

The data was read in, normalised and gradient descent was used to train a linear regression model of the data.

Part i. Data

The data was read in as below. The independent variable X and the target variable y were both extracted.

```
#Data
df = pd.read_csv('week1.txt')
df.head()
df.reset_index(level=0, inplace=True)
df.head()

#Feature
X = np.array(df.iloc[:,0])
X = X.reshape(-1,1)

#Target variable
y = np.array(df.iloc[:,1]);
y = y.reshape(-1,1)
```

Part ii. Data Normalisation

The data was normalised as follows;

```
#Normalise
y = (y - min(y))/(max(y)-min(y))
X = (X - min(X))/(max(X)-min(X))
```

A plot of the normalised data is shown in Figure 1.

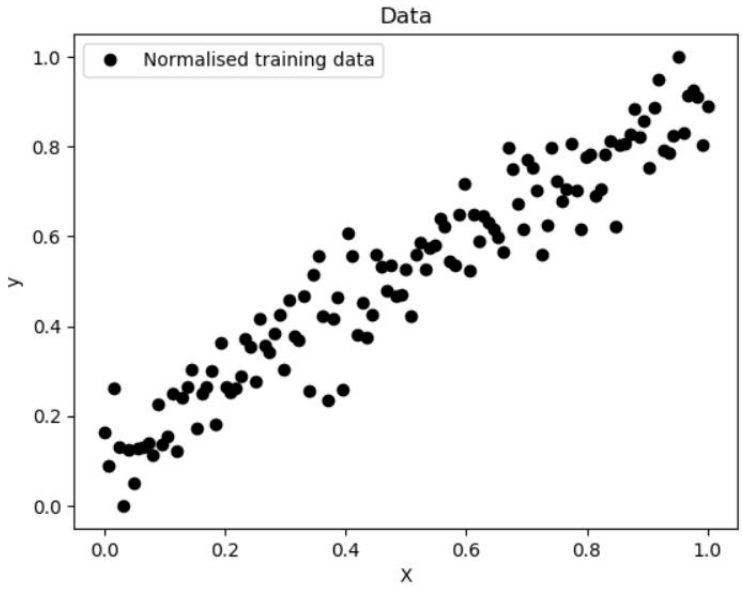

Figure 1. Normalised Data

Part iii. Gradient Descent to train Linear Regression model

The function `gradient_descent` was developed to train a linear regression model of the data via gradient descent. In linear regression, the model is of the following form;

$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x \qquad (1)$$

Whereby X is the independent variable, y is the target variable and θ_0, θ_1 are the unknown model parameters. Gradient descent is an optimization algorithm that can be used to determine the unknown model parameters by finding their values which minimise the cost function as below;

Cost function

$$J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \quad \text{where } h_\theta(x) = \theta_0 + \theta_1 x \qquad (2)$$

The function `gradient_descent` as below takes as parameters the number of iterations, the learning rate α, the feature X and the target variable y. The model parameters theta0, theta1 are randomly initialised and via gradient descent their final values are determined. The function returns the cost and the model parameters θ_0 and θ_1

```
def gradient_descent(iterations, alpha, X, y):

    #Variable Setup
    m = len(y)
    theta0 = np.random.random(1)
    theta1 = np.random.random(1)
    cost = np.zeros(iterations)

    #Gradient Descent
    for i in range(iterations):
        h = theta0 + theta1*X
        d0 = -(2*alpha/m)*np.sum(h-y)
        d1 = -(2*alpha/m)*np.sum(np.dot(X.T, (h-y)))
        theta0 = theta0 + d0
        theta1 = theta1 + d1
        cost[i] = (1/m)*np.sum(np.square(h-y))

    return cost, theta0, theta1
```

Part b

A linear regression model was trained on the data.

Part i. Gradient descent – for a range of learning rates

A range of learning rates α for the gradient descent algorithm were trialled to investigate the effect on model training. The parameter values tested were; [0.005, 0.01, 0.1, 0.2, 0.5, 0.8]. The resultant cost function was plotted across iterations for each of the values as shown in Figure 2. The learning rate has a significant effect on training. For small learning rates the algorithm is slow to converge. The extreme case of this is observed for a learning rate of 0.005 as the cost function has not converged within 1000 iterations, as shown in Figure 2. On the other hand, too large a learning rate can result in the cost function failing to converge. In this case the algorithm can often jump over minima, failing to converge to the global minimum. This is the case for a learning rate of 0.8 as displayed.

A learning rate of approximately 0.1 appears optimal in this instance as the cost function has converged within 200 iterations. This is clearly evident in Figure 3, when the model is trained for 200 iterations.

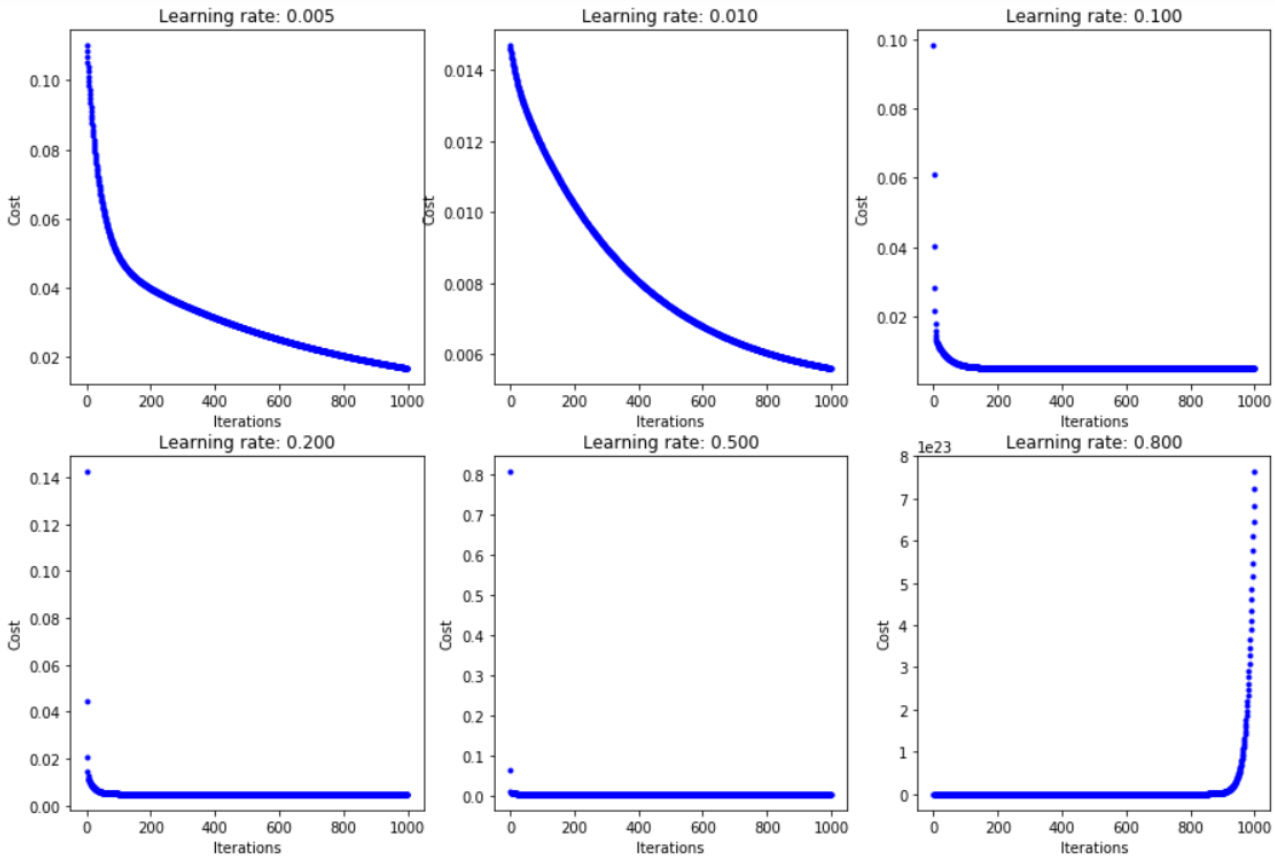

Figure 2. The Cost function of the gradient descent algorithm as it changes across iterations for a range of learning rates; [0.005, 0.01, 0.1, 0.2, 0.5, 0.8]

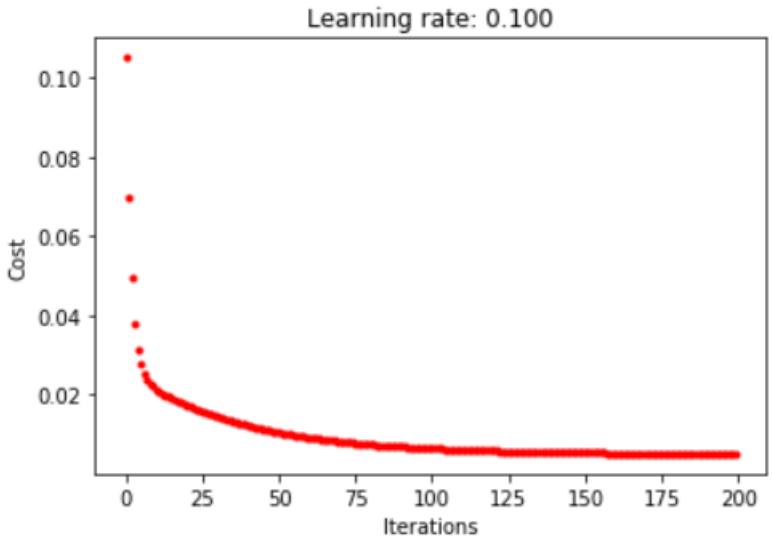

Figure 3. Convergence of cost function within 200 iterations

The function `apply_gradient_descent` as below was used to implement the gradient descent algorithm across the range of learning rates. The function produces the plots shown in figure 2. It also returns a dataframe of the model parameters and the cost function across the range of learning rates.

```
def apply_gradient_descent(iterations, l_rates, X, y):
    ''' Apply gradient descent across a range of learning rates. Plots the resultant cost vs iterations.
    Returns - dataframe of model parameters and cost for all learning rates l_rates '''

    #Set up
    fig = plt.figure(figsize=(15, 10))
    count = 0
    df_results = []

    for alpha in l_rates:

        #Apply gradient descent for given learning rate
        cost, theta0, theta1 = gradient_descent(iterations, alpha, X, y)
        count +=1

        #Plot Cost
        plt.subplot(2, 3, count)
        plt.plot(range(iterations), cost,'b.')
        plt.title('Learning rate: %.3f' %alpha)
        plt.xlabel('Iterations')
        plt.ylabel('Cost')

        #Store model parameters and cost in a dictionary
        d = {
        'l_rate': alpha,
        'theta0' : '%.5f'%(theta0[0]),
        'theta1' : '%.5f' %(theta1[0]),
        'final cost': '%.5f' %(cost[iterations-1])
        }

        df_results.append(d) #, ignore_index=True)

    plt.show()

    df_results = pd.DataFrame(df_results) #Results; Model parameters and cost across range of learning rates

    return df_results
```

The resultant dataframe is shown below which includes the model parameters and cost across a range of learning rates;

	l_rate	theta0	theta1	final cost
0	0.005	0.31857	0.42168	0.01686
1	0.010	0.16488	0.70852	0.00560
2	0.100	0.11956	0.79312	0.00498
3	0.200	0.11956	0.79312	0.00498
4	0.500	0.11956	0.79312	0.00498
5	0.800	703877900139.70923	377125672459.87671	764119695186151467909120.00000

Table 1. Model parameters and final cost function across the range of learning rates trialled.

Part ii. Linear Regression model

For a learning rate of 0.1 the model trained via gradient descent is as follows;

$$y = \theta_0 + \theta_1 x = 0.11956 + 0.79312x \qquad (3)$$

The regression line is plotted against the data as shown in Figure 4. The model appears to be a good fit and captures the upward increasing trend of the data.

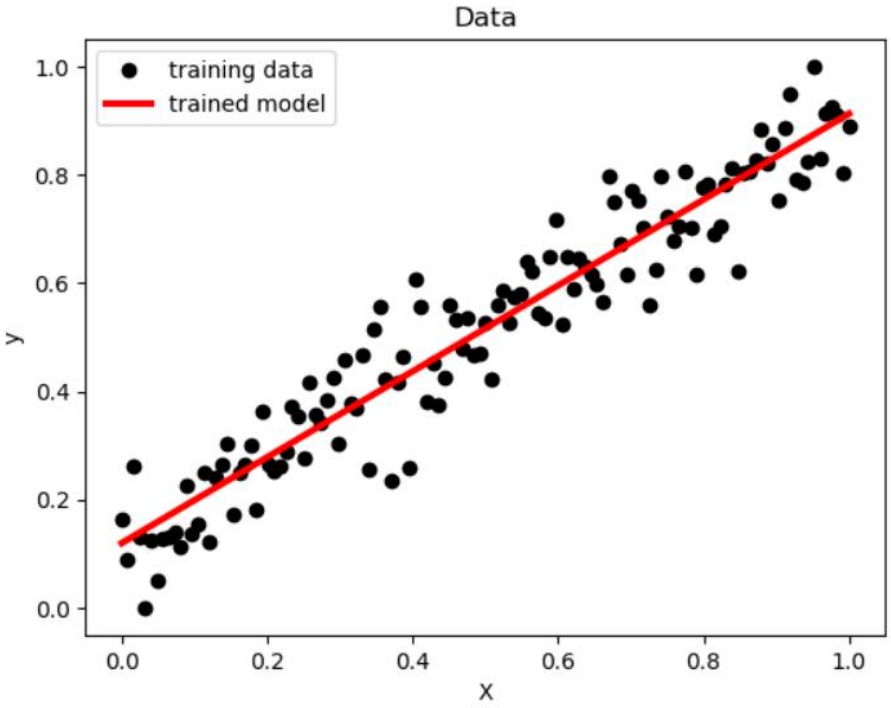

Figure 4. Training data and trained linear regression model.

The model parameters of the regression model for all learning rates are shown in Table 1. For learning rates in the range 0.01 to 0.5, the model parameters are all consistent; the gradient descent algorithm appears to have reached the global minimum in all cases. Evidently for large learning rates such as 0.8 the algorithm fails to converge.

Part iii: Comparison to Baseline Model

The final model and a baseline model are compared below. By visually inspecting the data as in figure 4, a constant value of 0.5 was deemed appropriate for the baseline model

Trained Model

$y = 0.11956 + 0.79312x$ (3)

Baseline model;

$y = 0.5$ (4)

The models are plotted against the data below in figure 5

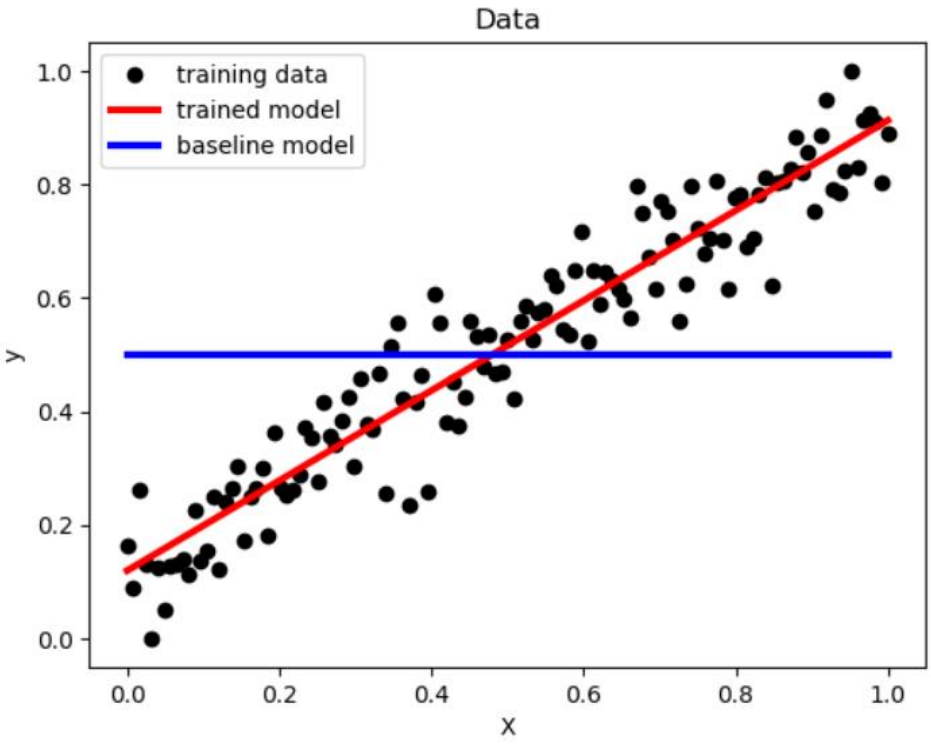

Figure 5. Training data, trained model and baseline model.

A comparison of the cost of the trained model and the baseline model is as below. There is a significant difference between the final cost of both models. This is as expected and evident from Figure 5; the trained model provides a much better fit of the data.

	Trained model	Baseline model
Cost function	0.0056	0.2084

The cost for the trained model was determined via the gradient_descent function. The cost of the baseline was determined as follows;

```
#Cost of baseline model
def baseline_model_cost(theta0, y):
    'Cost of baseline model'

    #Variable Setup
    m = len(y)
    cost = (1/m)*np.sum(np.square(theta0-y))

    return cost

theta0_baseline = 0.5
cost_baseline = baseline_model_cost(theta0_baseline, y)
```

Part iv. Regression model – sklearn

The linear regression function from the sklearn library was then used to train a linear regression model on the data. The code used is as follows;

```
#Part 4 - Linear Regression - sklearn
reg = LinearRegression().fit(X, y)

reg.coef_
array([[0.79312161]])

reg.intercept_
array([0.11955553])
```

Both methods return identical model parameters as shown below.

Linear Regression Model	Gradient Descent	sklearn
Theta0	0.11956	0.11956
Theta1	0.79312	0.79312

The fact that both methods resulted in the same model parameters is a good indication that the gradient descent algorithm reached the global minimum.

Appendix – Code

```
#Week 1 Assignment - Linear Regression via Gradient Descent

#Imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

#********************************** Part a ***********************************
#Data
df = pd.read_csv('week1.txt')
df.head()
df.reset_index(level=0, inplace=True)
df.head()

#Feature
X = np.array(df.iloc[:,0])
X = X.reshape(-1,1)

#Target variable
y = np.array(df.iloc[:,1]);
y = y.reshape(-1,1)

#Normalise
y = (y - min(y))/(max(y)-min(y))
X = (X - min(X))/(max(X)-min(X))

#Visualise
plt.plot(X, y, 'o', color='black');
plt.xlabel('X')
plt.ylabel('y')
plt.title('Data')
plt.show()
```

```
#**************************************

#Linear Regression - Gradient descent
def gradient_descent(iterations, alpha, X, y):

    #Variable Setup
    m = len(y)
    theta0 = np.random.random(1)
    theta1 = np.random.random(1)
    cost = np.zeros(iterations)

    #Gradient Descent
    for i in range(iterations):
        h = theta0 + theta1*X
        d0 = -(2*alpha/m)*np.sum(h-y)
        d1 = -(2*alpha/m)*np.sum(np.dot(X.T, (h-y)))
        theta0 = theta0 + d0
        theta1 = theta1 + d1
        cost[i] = (1/m)*np.sum(np.square(h-y))

    return cost, theta0, theta1
```

```
#********************************** Part b ************************************
def apply_gradient_descent(iterations, l_rates, X, y):
    ''' Apply gradient descent across a range of learning rates. Plots the resultant cost vs iterations.
    Returns - dataframe of model parameters and cost for all learning rates l_rates '''

    #Set up
    fig = plt.figure(figsize=(15, 10))
    count = 0
    df_results = []

    for alpha in l_rates:

        #Apply gradient descent for given learning rate
        cost, theta0, theta1 = gradient_descent(iterations, alpha, X, y)
        count +=1

        #Plot Cost
        plt.subplot(2, 3, count)
        plt.plot(range(iterations), cost,'b.')
        plt.title('Learning rate: %.3f' %alpha)
        plt.xlabel('Iterations')
        plt.ylabel('Cost')

        #Store model parameters and cost in a dictionary
        d = {
        'l_rate': alpha,
        'theta0' : '%.5f'%(theta0[0]),
        'theta1' : '%.5f' %(theta1[0]),
        'final cost': '%.5f' %(cost[iterations-1])
        }

        df_results.append(d) #, ignore_index=True)

    plt.show()

    df_results = pd.DataFrame(df_results) #Results; Model parameters and cost across range of learning rates

    return df_results
```

```
#Part 1 - Apply gradient descent across a range of learning rates

iterations = 1000
l_rates = [0.005, 0.01, 0.1, 0.2, 0.5, 0.8]
df_results = apply_gradient_descent(iterations, l_rates, X, y)

#*********************************
#Part 2 - Trained model (optimal learning rate)

alpha = 0.1
cost, theta0, theta1 = gradient_descent(iterations, alpha, X, y)

#Plot cost across 200 iterations (zoom in)
iterations = 200
cost, theta0, theta1 = gradient_descent(iterations, alpha, X, y)

#Plot Cost
plt.plot(range(iterations), cost,'r.')
plt.title('Learning rate: %.3f' %alpha)
plt.xlabel('Iterations')
plt.ylabel('Cost')

#Plot trained model vs training data
plt.plot(X, y, 'o', color='black', label = 'training data');
plt.plot(X, y_grad, color = 'red', linewidth= 3.0, label = 'trained model')
plt.legend(loc="upper left")
plt.xlabel('X')
plt.ylabel('y')
plt.title('Data')
plt.show()
```

```
#*********************************
#Part 3 - Baseline model

#Cost of baseline model
def baseline_model_cost(theta0, y):
    'Cost of baseline model'

    #Variable Setup
    m = len(y)
    cost = (1/m)*np.sum(np.square(theta0-y))

    return cost

theta0_baseline = 0.5
cost_baseline = baseline_model_cost(theta0_baseline, y)

#Plot trained model vs baseline vs data
y_baseline = theta0_baseline*np.ones(len(X))

plt.plot(X, y, 'o', color='black', label = 'training data');
plt.plot(X, y_grad, color = 'red', linewidth= 3.0, label = 'trained model')
plt.plot(X, y_baseline, color = 'blue', linewidth= 3.0, label = 'baseline model')
plt.legend(loc="upper left")
plt.xlabel('X')
plt.ylabel('y')
plt.title('Data')
plt.show()


#*********************************
#Part 4 - Linear Regression - sklearn
reg = LinearRegression().fit(X, y)
reg.coef_
reg.intercept_
```