Machine Learning Assignment 1 - Week 1

Thomas Kelly - 16323455
tkelly2@tcd.ie

First line of data set: # id:14-5383--14

(a) (i) Write a short python program that reads the data you downloaded

The pandas library is used to read the data from the csv file.

```
df = pd.read_csv("week1.csv", comment='#')
```

(a) (ii) … and normalises the data

As described in lectures, the data is normalised by calculating the mean and standard deviation. This allows us to normalise the data by shifting it onto a smaller scale towards having a mean of 0, and standard deviation of 1.

- $x_n = \frac{x_i - \mu}{}$

```
X = np.array(df.iloc[:, 0])
y = np.array(df.iloc[:, 1])

X = X.reshape(-1, 1)
y = y.reshape(-1, 1)

x_mean = np.mean(X)
y_mean = np.mean(y)

x_std = np.std(X)
y_std = np.std(y)

x_normalised = (X - x_mean) / x_std
y_normalised = (y - y_mean) / y_std
```

(a) (iii) Use gradient descent to train a linear regression model

The cost function implements the expression calculating the cost or error between the predicted value and actual value for output ‘y’.

- $J() = \frac{1}{m} \sum_{i=1}^{m} ((h(x) - y))^2$

The gradientDescent function repeatedly calls this cost function, training or optimising the values for theta by calculating a difference, or ‘step_size’ which aims to minimise the cost function with each iteration.

The values for iterations and learning rate (or alpha) can be modified to change the speed and accuracy of the algorithm's convergence.

```
def costF(input_x, y, theta):
   calculate_inner = np.power(((input_x * theta) - y), 2)
   return np.sum(calculate_inner) / len(X)

def gradientDescent(input_x, y, theta, alpha):
   for i in range(0, iterations):
       step_size = (-2*alpha/len(input_x)) * \
           np.sum(((input_x * theta) - y) * input_x, axis=0)
       theta = theta + step_size
       cost = costF(input_x, y, theta)
       log_cost_array[i] = cost  # Log the cost at this iteration
       iteration_array[i] = i  #  Log the value of this iteration
   return (theta, cost)

trained_theta_array, cost = gradientDescent(x_normalised,
y_normalised, theta, learning_rate)
```

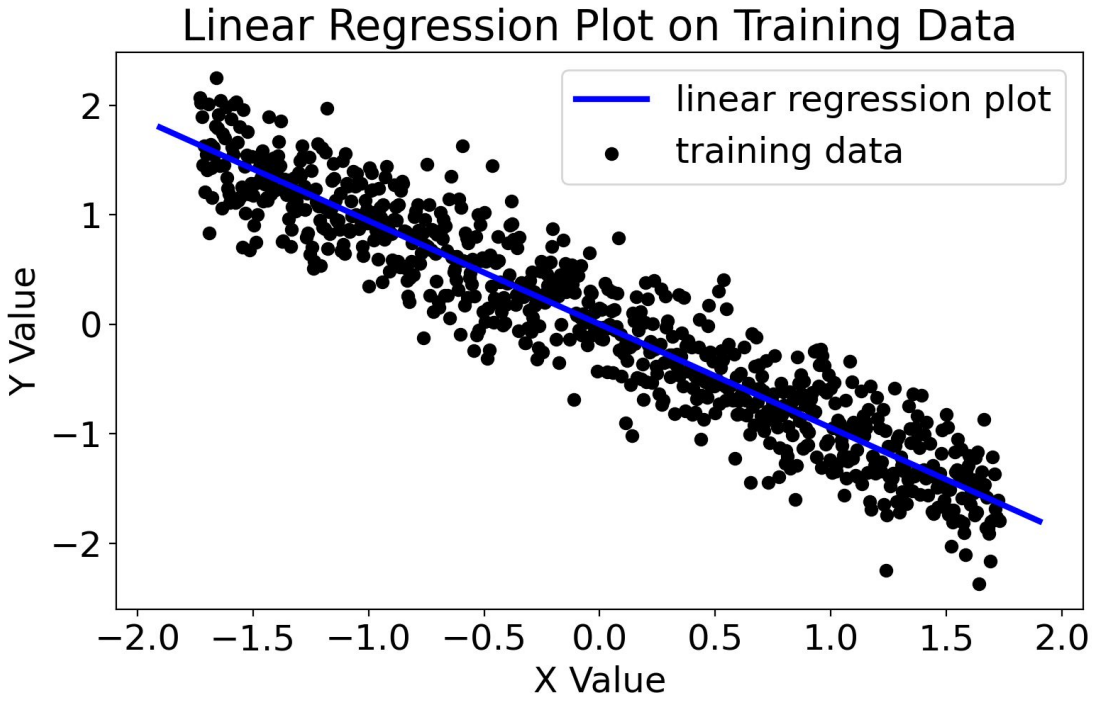

Figure 1; Linear Regression plot

After the set number of *iterations,* the gradient descent function halts and returns the minimised cost, along with the optimised values for theta that result in this cost.
In my case, the *trained_theta_array* returns;

- [$_0$, $_1$] = [0.0001329, -0.9445]
- Cost = 1.107

The above plot shows the training data in black, with a superimposed linear regression model plot generated using the expression;

- h(x) = $_0$ + $_1$x
- y_normalised = trained_theta_array[0] + trained_theta_array[0] * x_normalised

(b) (i) Try a range of learning rates...

Learning rates of 0.001, 0.01 and 0.1 were used to compare how different learning rates can affect the performance of the training process.
The value of the cost function was calculated on each iteration and plot here across each iteration of the training process.

- **α = 0.001**

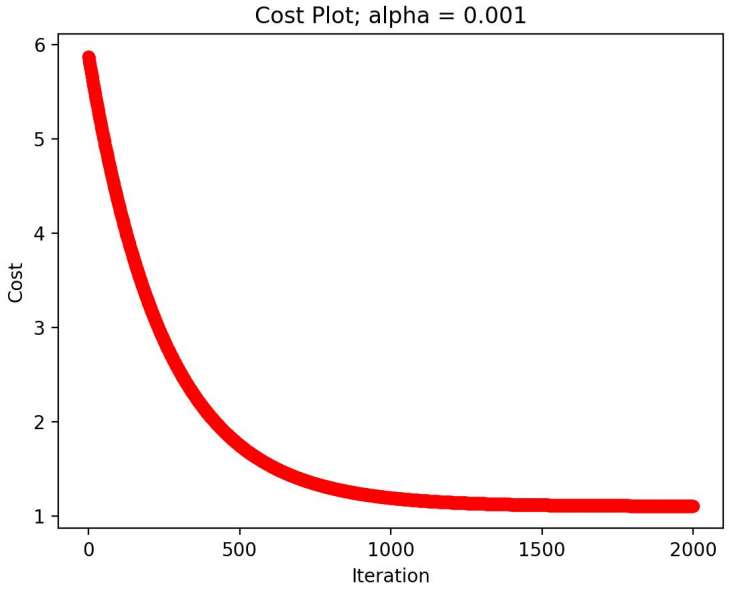

Figure 2; Cost plot for learning rate = 0.001

As is evident here, using a small learning rate results in the model taking a higher amount of iterations to converge at its minimum cost. The cost function flattens out at its optimum value of 1.109 around approximately 1700 iterations.

- **α = 0.01**

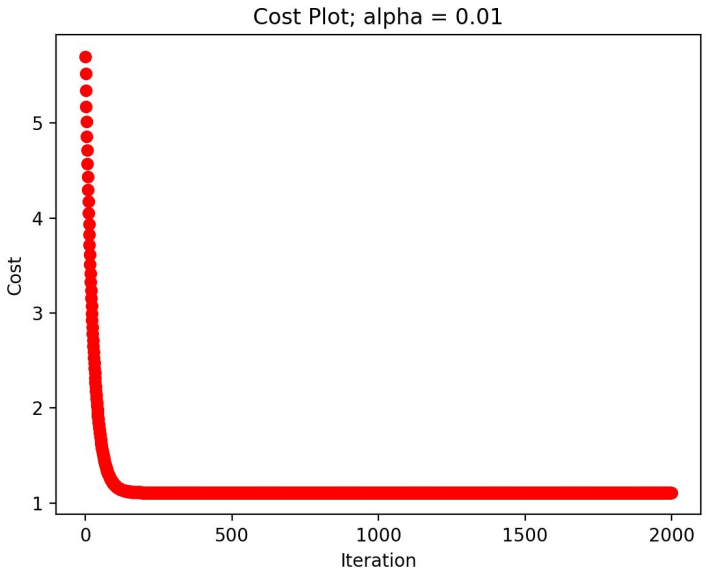

Figure 3; Cost plot for learning rate = 0.01

Using a learning rate of an order of magnitude higher, the model converges at its optimum cost value much more quickly, flat-lining at roughly 200-250 iterations.

- **α = 0.1**

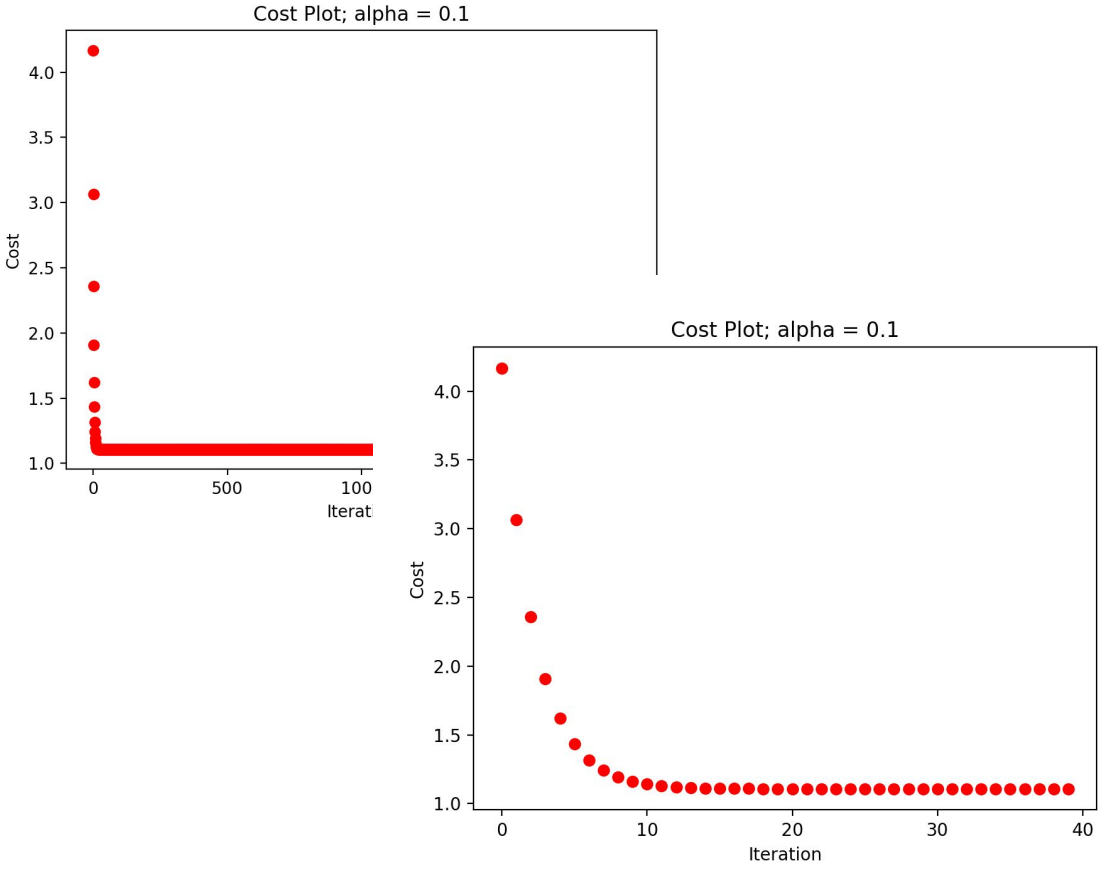

Figure 4; Cost plot for learning rate = 0.1, (zoomed in)

Zooming in on the graph produced here, we can observe that the cost function converges very quickly, in about 20 iterations. While this is the best case scenario examined here, a learning rate slightly higher than this will result in overfitting, resulting in the model being useless.

(b) (ii) Report the parameter values of linear regression model after it has been trained

The algorithm reported the following values;

- Θ_0 = 0.0001329
- Θ_1 = -0.9445

The linear model expression should therefore look as follows:

- h(x) = 0.0001329 - 0.9445x

(b) (iii) Compare the cost of the trained model with the cost associated with a baseline model

The trained model reflected a final cost value of **1.107.**

Examining the data in figure 5 below, a constant value of 0 was selected to reflect a baseline model's prediction.

- θ_0 = 0 …was selected based on the spread of the training data around 0
- θ_1 = 0 ...was selected so the baseline model gave a constant prediction

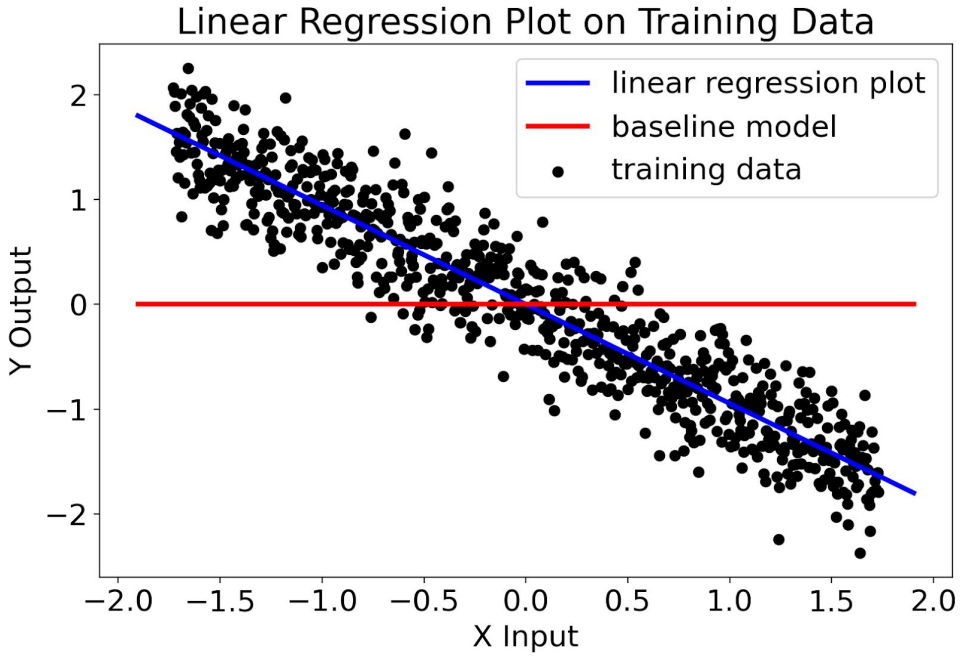

Figure 5; Plot baseline model with linear regression model

Passing this into the cost function formula;

- $J() = \frac{1}{m}\sum_{i=1}^{m}((h(x) - y))^2$ = 2

Evidently the error (cost) here is approximately twice as bad than that yielded by the linear regression model.

(b)(iv) Now use sklearn, and compare it to the previous model

Using the same data normalised in the same fashion as previously described, the model is generated using the following code;

```
X_train, X_test, y_train, y_test = train_test_split(
   x_normalised, y_normalised, test_size=0.2, random_state=0)


regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

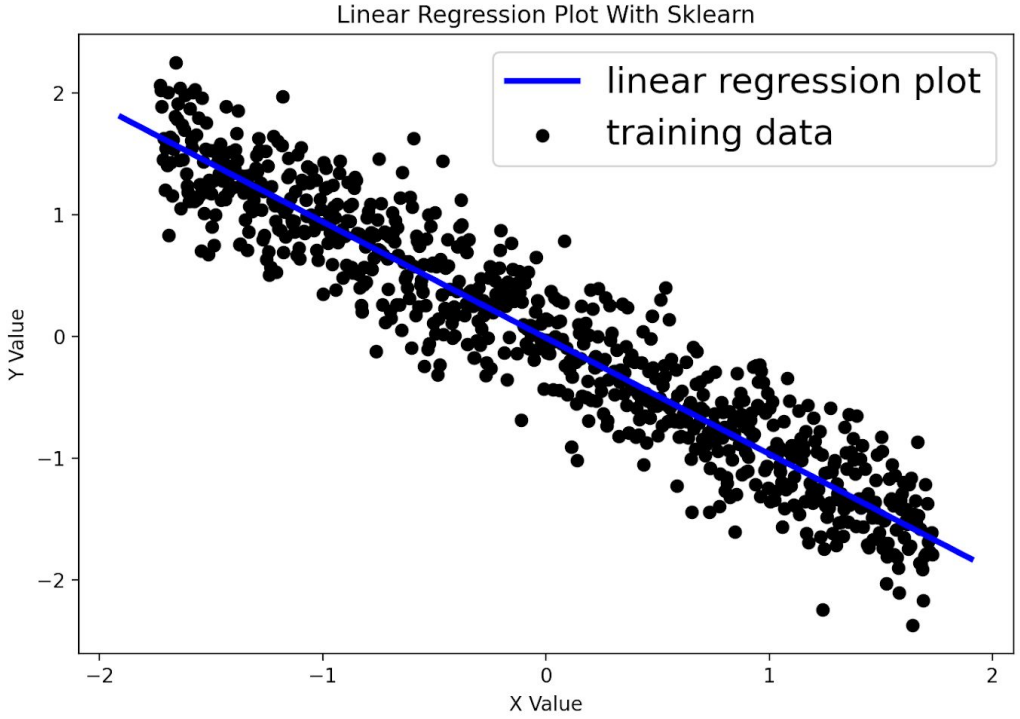

Figure 6; Linear Regression using Sklearn library

The diagram shows a linear regression plot over the training data implemented with Sklearn. This greatly simplifies the code and reduces the understanding needed to implement such a model.

Examining both model plots, the difference is negligible or impossible to spot. One would assume that the gradient descent algorithm should have the same accuracy as that of the Sklearn implementation after sufficient iterations; in order for the cost function to converge at its absolute minimum value.

That being said, using Sklearn also avoids the possibility of the gradient descent algorithm getting stuck at a local minimum, rather than the absolute minimum, which would complicate its implementation further.

Code

week1.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# (a)(i) read the data in
df = pd.read_csv("week1.csv", comment='#')

X = np.array(df.iloc[:, 0])
y = np.array(df.iloc[:, 1])

X = X.reshape(-1, 1)
y = y.reshape(-1, 1)

x_mean = np.mean(X)
y_mean = np.mean(y)

x_std = np.std(X)
y_std = np.std(y)

# (a)(ii) Normalise the data
x_normalised = (X - x_mean) / x_std
y_normalised = (y - y_mean) / y_std

ones = np.ones([x_normalised.shape[0], 1])
x_normalised = np.concatenate([ones, x_normalised], 1)
```

```
theta = np.array([[1.0, 1.0]])

iterations = 1000
learning_rate = 0.1

log_cost_array = [0] * iterations
iteration_array = [0] * iterations



def costF(input_x, y, theta):
   calculate_inner = np.power(((input_x * theta) - y), 2)
   return np.sum(calculate_inner) / len(X)

# (a)(iii) Gradient Descent Algorithm Implementation
def gradientDescent(input_x, y, theta, alpha):
   for i in range(0, iterations):
       step_size = (-2*alpha/len(input_x)) * \
           np.sum(((input_x * theta) - y) * input_x, axis=0)
       theta = theta + step_size
       cost = costF(input_x, y, theta)
       log_cost_array[i] = cost  # Log the cost at this iteration
       iteration_array[i] = i  #  Log the value of this iteration
   return (theta, cost)



trained_theta_array, cost = gradientDescent(
   x_normalised, y_normalised, theta, learning_rate)



re_normalised = (np.array(df.iloc[:, 0]).reshape(-1, 1) - x_mean)
/ x_std
axes = plt.gca() # for plotting
xs = np.array(axes.get_xlim())
ys = trained_theta_array[0][0] + trained_theta_array[0][1] * xs
```

week1_sklearn_version.py

```
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

df = pd.read_csv("week1.csv", comment='#')

X = np.array(df.iloc[:, 0])
y = np.array(df.iloc[:, 1])

X = X.reshape(-1, 1)
y = y.reshape(-1, 1)

x_mean = np.mean(X)
y_mean = np.mean(y)

x_std = np.std(X)
y_std = np.std(y)

x_normalised = (X - x_mean) / x_std
y_normalised = (y - y_mean) / y_std

X_train, X_test, y_train, y_test = train_test_split(
   x_normalised, y_normalised, test_size=0.2, random_state=0)

regressor = LinearRegression()
regressor.fit(X_train, y_train)
```