A. Introduction

A.1. Objective

The objective of this application is to solve a linear regression problem with the help of the Gradient Descent algorithm. This application utilizes an already built dataset containing student marks where final marks are dependent on midterm marks.This application uses this dataset to create a linear regression model which helps detect the final mark based on the midterm mark.

A.2. Theory

This application utilizes methods such as standardization, linear regression, error (cost function) and gradient descent. This part of the report will identify each of these and briefly theorize the concept behind them.

A.2.1. Standardization

Standardization is a tool used for normalization in machine learning algorithms. In this application, we deal with standardized linear regressions, regression models where the variables are standardized. The mathematical definition of standardization is given below [1]:

$$x' = \frac{x - \bar{x}}{\sigma}$$

Here, the independent x values are subtracted from their mean and then divided by the standard deviation. This *x'* variable will have a zero mean and unit standard deviation [1].

A.2.1. Linear Regression

Linear regression is a statistical approach to modelling a relationship between a dependent variable and an independent variable [4]. Consider x to be an independent variable and y to be a dependent variable, we can define the linear regression relationship as [1]:

$$Y = m(x) + b$$

This is an equation for a line, where m is the slope and b is the y-intercept. This equation is used to train a model within a dataset and predict the value of y for any given x value. We try to search for the best values of slope and y-intercept in order to build the best-fit line, the better fit the line, the minimum the errors [5].

Error (Cost Function)

The cost function finds the error in our predicted values for slope and y-intercept. The purpose is to reduce this error in order to obtain the best fitted slope and y-intercept, ultimately this will result in an accurate linear regression model. The cost function can be determined using the following equation [1]:

$$E = \frac{1}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)^2$$

Here, we first determine the difference between the actual y value and the predicted y value from y = m(x) + c. We square this difference and then find the sum of all the squares for every value of x [5]. This relation can be further described as the following, where m(x) + c is substituted for y prediction [1].

$$E = \frac{1}{n}\sum_{i=0}^{n}(y_i - (mx_i + c))^2$$

Gradient Partial Derivatives

The Gradient Descent algorithm helps minimize the cost function, this yields in a further accurate model [4]. Through this algorithm, we find various different values for slope (m) and y-intercept (b), until the slope reaches a convergence point [2]. In order to produce this algorithm, we use partial derivative equations. The partial derivatives are given as follows [1]:

$$\frac{\partial}{\partial \mathrm{m}} = \frac{2}{N}\sum_{i=1}^{N} -x_i\left(y_i - (mx_i + b)\right)$$

$$\frac{\partial}{\partial b} = \frac{2}{N}\sum_{i=1}^{N} -\left(y_i - (mx_i + b)\right)$$

When running the algorithm, we have the freedom of starting at any value of slope and y-intercept. The application manual for this application initializes slope at -0.5 and y-intercept at 0, hence this report will use those values. Once our values are initialized, the algorithm will march downhill for each iteration on the cost function towards the best line. Each iteration will update the slope and y-intercept until a line that yields very minimal error occurs [5]. The partial derivatives produce the direction each iteration moves, we are able to update them using a learning rate [1]. The learning rate controls the size of steps we take when moving downhill for each iteration. If the learning rate is too large, the minimum error may be skipped over. A smaller learning rate will require more amount of iterations to gain the right minimum error [1].

Updating the slope and y-intercept using the learning rate can be done using the following formula [1]:

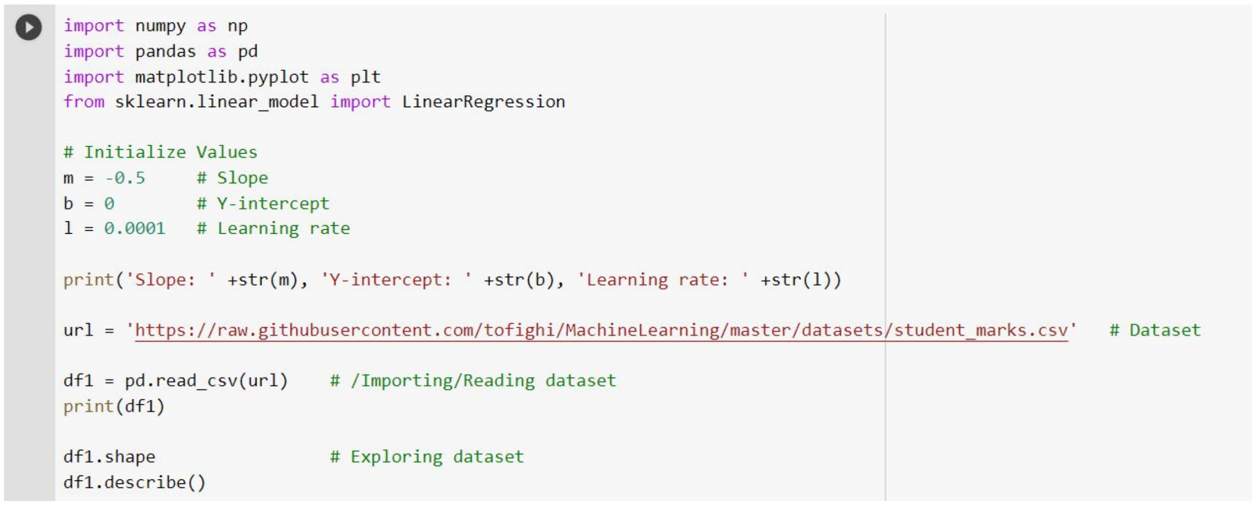
$$m_{new} = m_{old} - \alpha \frac{\partial E}{\partial \mathrm{m}}$$

$$b_{new} = b_{old} - \alpha \frac{\partial E}{\partial \mathrm{b}}$$

IMPLEMENTATION

A. Initializing Data

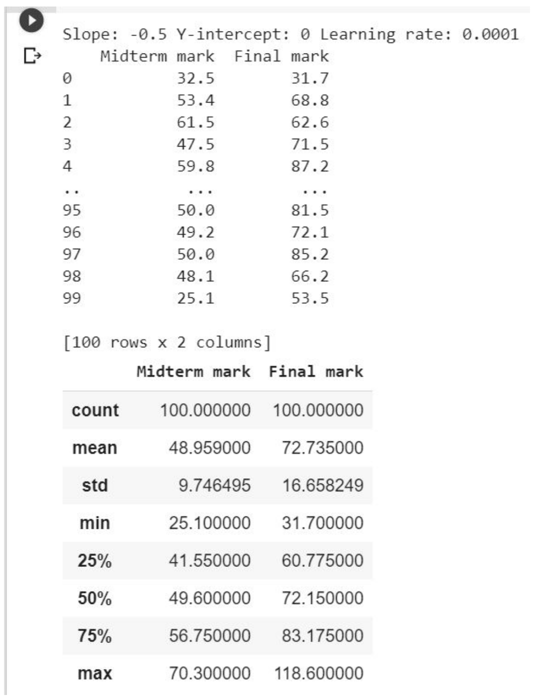
A.1. Initialize m, b and α as -0.5, 0 and 0.0001 respectively



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

# Initialize Values
m = -0.5     # Slope
b = 0        # Y-intercept
l = 0.0001   # Learning rate

print('Slope: ' +str(m), 'Y-intercept: ' +str(b), 'Learning rate: ' +str(l))

url = 'https://raw.githubusercontent.com/tofighi/MachineLearning/master/datasets/student_marks.csv'   # Dataset

df1 = pd.read_csv(url)    # /Importing/Reading dataset
print(df1)

df1.shape                 # Exploring dataset
df1.describe()
```

Figure A.1. *Code for Initializing Model Values.*

Here, we use python libraries numpy, pandas, matplotlib, and sklearn to help build, model and plot our data. Values for m, b and the learning rate have been established. The student marks dataset which is used through this application is imported and defined as *df1*.



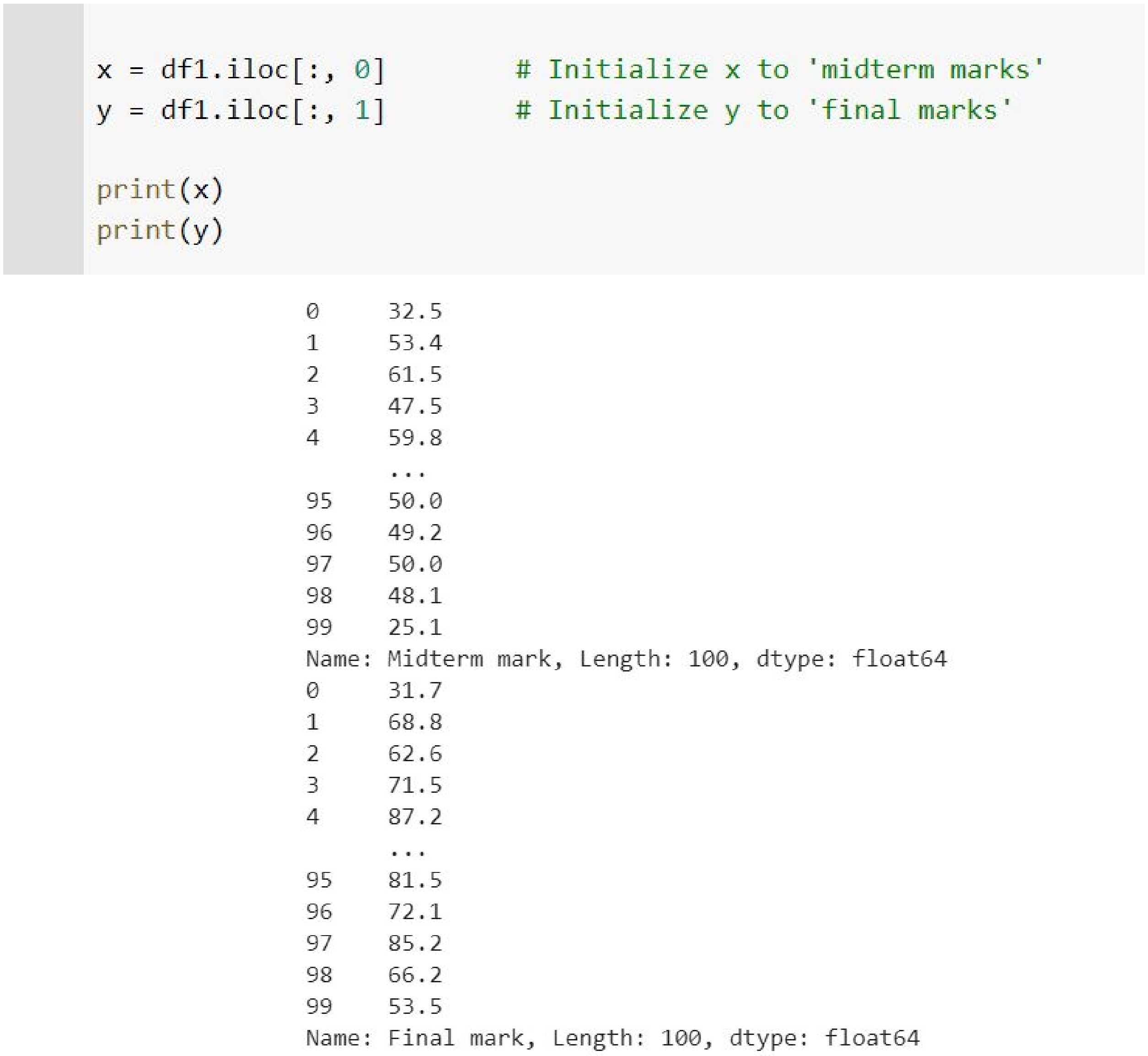
```
Slope: -0.5 Y-intercept: 0 Learning rate: 0.0001
    Midterm mark  Final mark
0           32.5        31.7
1           53.4        68.8
2           61.5        62.6
3           47.5        71.5
4           59.8        87.2
..           ...         ...
95          50.0        81.5
96          49.2        72.1
97          50.0        85.2
98          48.1        66.2
99          25.1        53.5

[100 rows x 2 columns]
```

	Midterm mark	Final mark
count	100.000000	100.000000
mean	48.959000	72.735000
std	9.746495	16.658249
min	25.100000	31.700000
25%	41.550000	60.775000
50%	49.600000	72.150000
75%	56.750000	83.175000
max	70.300000	118.600000

Figure A.2. *Output of A.1 Code.*

The describe() function helps determine statistics of the given dataset. For this application, we are only concerned with standard deviation and the mean of midterm and final marks, this function calculates these for us immediately. Standard deviation for midterm marks is about 9.7465 whereas the mean is 48.959. This will later come in handy when we standardize our model.

Now that we have our initial values for the slope, y-intercept and learning rate established. We can indicate what our x and y values are, this means making the midterm marks column within the dataset equal our independent x variable and final marks equal our dependent y variable. We do this with the help of the *iloc* function, as shown in ***Figure A.3***. The output for this function yields the figure below that.



```python
x = df1.iloc[:, 0]        # Initialize x to 'midterm marks'
y = df1.iloc[:, 1]        # Initialize y to 'final marks'

print(x)
print(y)
```

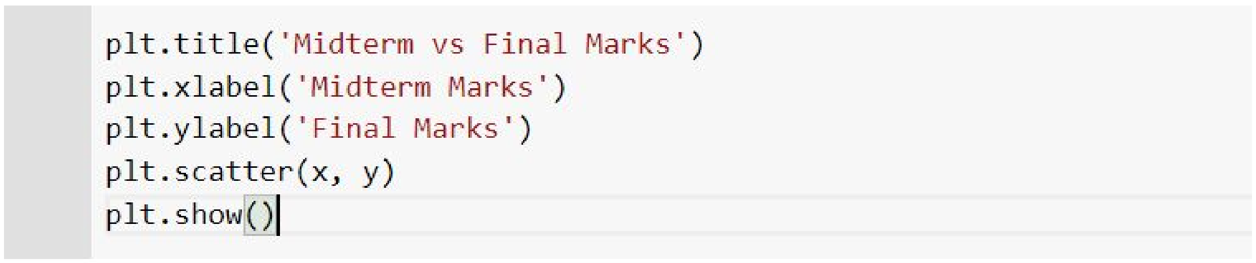
```
0     32.5
1     53.4
2     61.5
3     47.5
4     59.8
      ...
95    50.0
96    49.2
97    50.0
98    48.1
99    25.1
Name: Midterm mark, Length: 100, dtype: float64
0     31.7
1     68.8
2     62.6
3     71.5
4     87.2
      ...
95    81.5
96    72.1
97    85.2
98    66.2
99    53.5
Name: Final mark, Length: 100, dtype: float64
```

Figure A.3. *Code for defining x and y with respective outputs.*

B. Plotting Data-Points and Regression Line

B.1. Plotting Data-Points from Data-Set

Plotting the data points is an easy task, we use the initialized *plt* variable from matplotlib.pyplot. Pyplot is a state-based sub-module, it helps with building and generating interactive graphs. Looking at ***Figure B.1***, we can see the scatter plot which is presented where x is the midterm marks and y is the final marks.



```
plt.title('Midterm vs Final Marks')
plt.xlabel('Midterm Marks')
plt.ylabel('Final Marks')
plt.scatter(x, y)
plt.show()
```

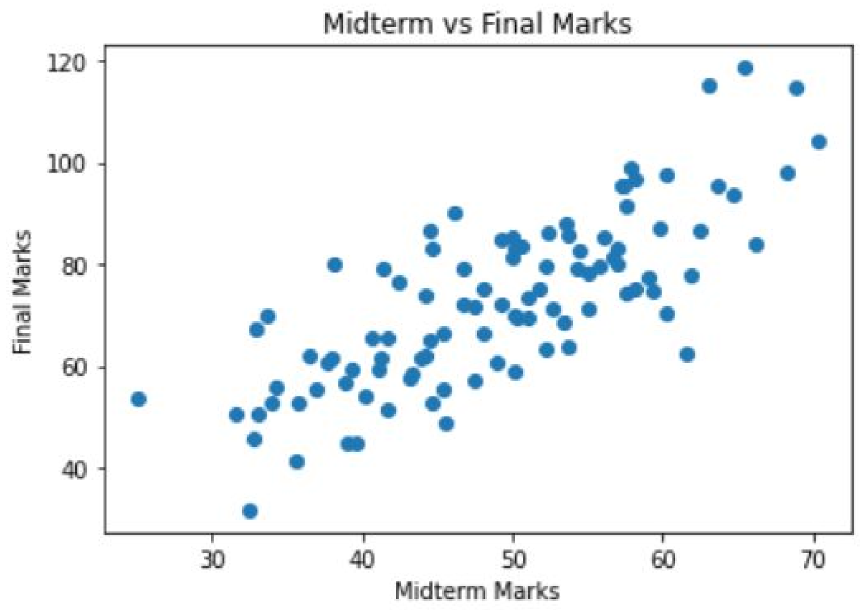
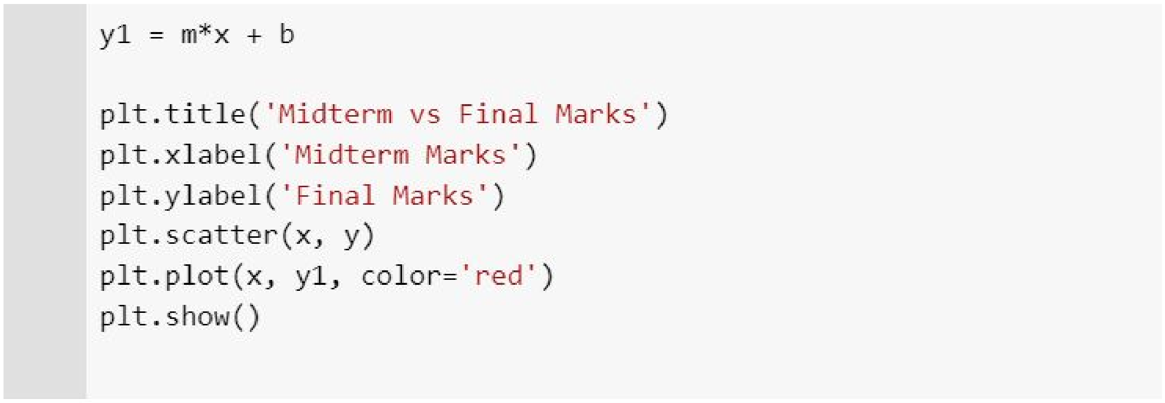


Figure B.1. *Code and Graph for Data-Set*

Our y values, midterm marks, are independent because we aren't able to mathematically compute the midterm marks. However, we are able to predict what the dependent values, final marks or x values, are going to be from our y values. This application takes on this approach to predict y values and build a regression model.

B.2. Initial Regression Line

When plotting the linear regression, we are concerned with building the best-fit line. We are already given our slope and y-intercept (m = -0.5, b = 0), so we can simply initialize our y1 = m*x + b formula and plot the best-fit line by calling the matplotlib.pyplot.plot(x, eq) with x as the array of midterm marks and the eq as y1 = m*x + b.



```
y1 = m*x + b

plt.title('Midterm vs Final Marks')
plt.xlabel('Midterm Marks')
plt.ylabel('Final Marks')
plt.scatter(x, y)
plt.plot(x, y1, color='red')
plt.show()
```

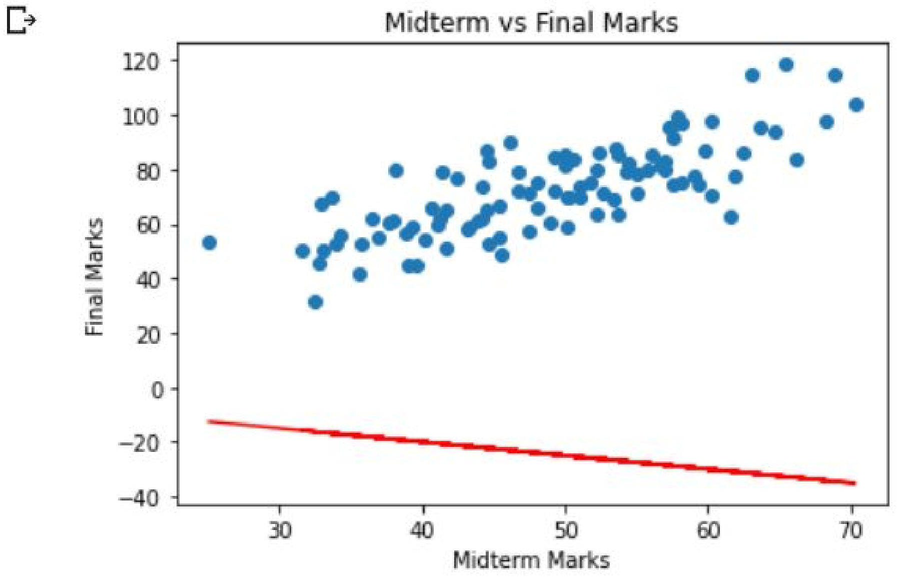


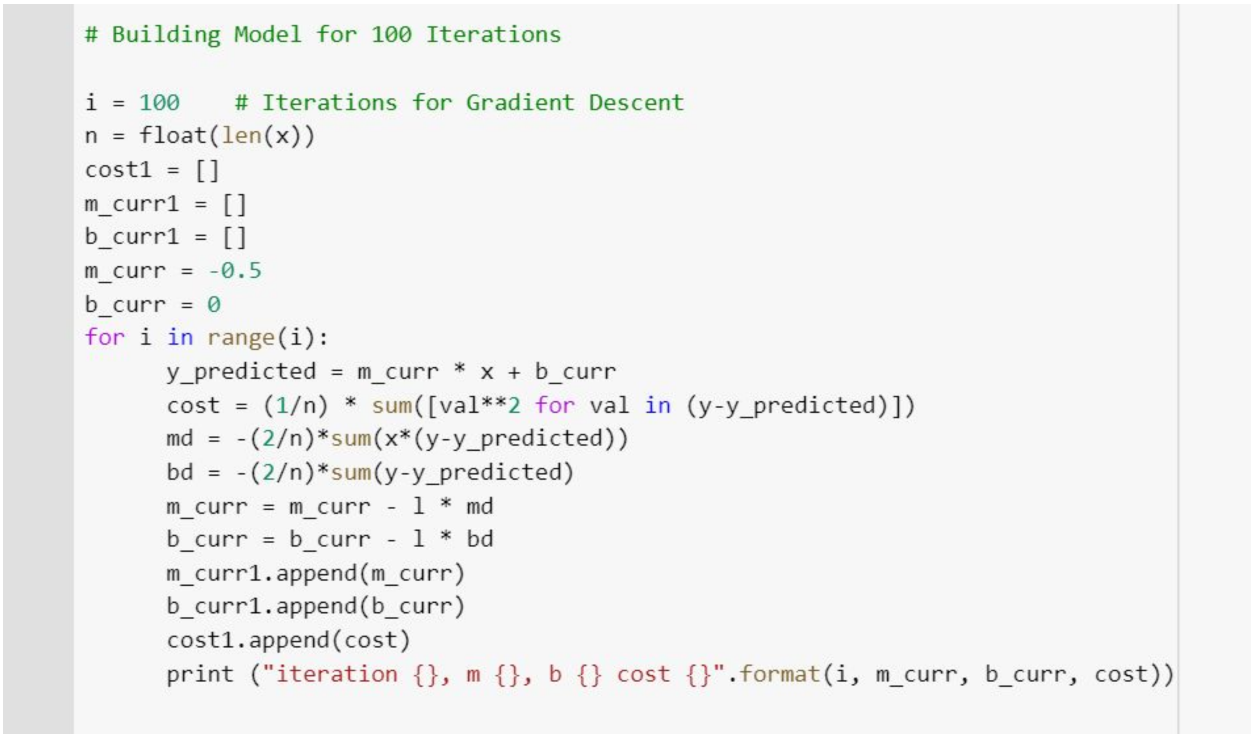
Figure B.2. *Code and Plot for Initial Regression Line.*

The regression line displayed in ***Figure B.2*** is a product of an untrained model, here we have used the slope of -0.5 and y-intercept of 0. It is clear that these values do not give us an accurate representation of the best-fit line. Since our slope is -0.5, this results in a decreasing best-fit line. Our next approach is to accurately determine the rightful slope and y-intercept that will yield the minimum error and give us a precise regression line.

C. Linear Regression with 100 Iterations

C.1. Building Model

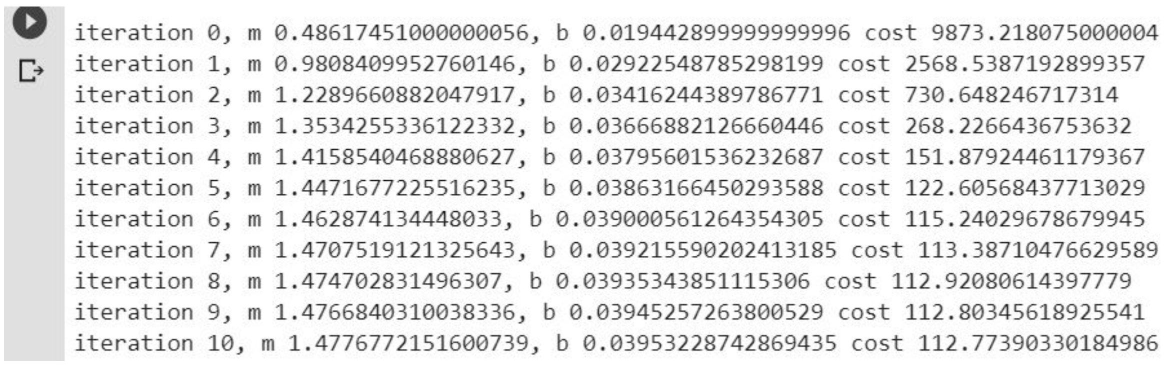
Now, we try to determine the proper slope and y-intercept for our model. We build a model that updates the slope and y-intercept for 100 iterations and plot the linear regression. This code implementation can be seen in ***Figure C.1***, here we also have the cost function and gradient derivatives (equations for these are shown in the ***theory*** section) established to help us build our model. Arrays for the updated slope, y-intercept and cost are used to append the values for each at every iteration and these values are used to plot the best fitted regression line for our data.



```
# Building Model for 100 Iterations

i = 100    # Iterations for Gradient Descent
n = float(len(x))
cost1 = []
m_curr1 = []
b_curr1 = []
m_curr = -0.5
b_curr = 0
for i in range(i):
      y_predicted = m_curr * x + b_curr
      cost = (1/n) * sum([val**2 for val in (y-y_predicted)])
      md = -(2/n)*sum(x*(y-y_predicted))
      bd = -(2/n)*sum(y-y_predicted)
      m_curr = m_curr - l * md
      b_curr = b_curr - l * bd
      m_curr1.append(m_curr)
      b_curr1.append(b_curr)
      cost1.append(cost)
      print ("iteration {}, m {}, b {} cost {}".format(i, m_curr, b_curr, cost))
```

Figure C.1. *Code for Building Model for 100 Iterations.*

The output of this code will print out a tuple of values for iteration, updated slope, updated y-intercept and error for each iteration. This output is very lengthy, seeing as it produces calculations for 100 iterations, this report will only display the first ten. The Python Jupyter Notebook file, which contains all the code, can be run to see the rest of the calculated values per variable.



```
iteration 0, m 0.48617451000000056, b 0.019442899999999996 cost 9873.218075000004
iteration 1, m 0.9808409952760146, b 0.02922548785298199 cost 2568.5387192899357
iteration 2, m 1.2289660882047917, b 0.03416244389786771 cost 730.648246717314
iteration 3, m 1.3534255336122332, b 0.03666882126660446 cost 268.2266436753632
iteration 4, m 1.4158540468880627, b 0.03795601536232687 cost 151.87924461179367
iteration 5, m 1.4471677225516235, b 0.03863166450293588 cost 122.60568437713029
iteration 6, m 1.462874134448033, b 0.039000561264354305 cost 115.24029678679945
iteration 7, m 1.4707519121325643, b 0.039215590202413185 cost 113.38710476629589
iteration 8, m 1.474702831496307, b 0.03935343851115306 cost 112.92080614397779
iteration 9, m 1.4766840310038336, b 0.03945257263800529 cost 112.80345618925541
iteration 10, m 1.477677215160739, b 0.03953228742869435 cost 112.77390330184986
```

Here, we can see that for the first iteration, we start when m is approximately 0.5 and b is 0.019. These values give us an error value of 9873.2, this is a very big error. If we had used these values to conduct a regression model, our model would be severely inaccurate. Thankfully, our model continues to go down the slope and y-intercept values, producing smaller values of error. Once our loop ends, our model gives us the following values for slope and y-intercept. These values give us the minimum amount of error: `m = 1.4789586122510086 and b = 0.025273343524573104`

C.2. Plotting Data-Points and Updated Regression Line

After defining the updated functions, we simply plot the data-points and the new regression line, similar to what we did previously with an updated *y2* using our newly updated slope and y-intercept. This regression line takes into account the best slope and y-intercept that can be used to produce a best-fitted regression. The model defined in the section above helps us reach this conclusion. ***Figure C.2*** displays this implementation.

```
y2 = m_curr*x + b_curr
plt.title('Midterm vs Final Marks')
plt.xlabel('Midterm Marks')
plt.ylabel('Final Marks')
plt.scatter(x, y)
plt.plot(x, y2, color='red')  # regression line
plt.show()
```

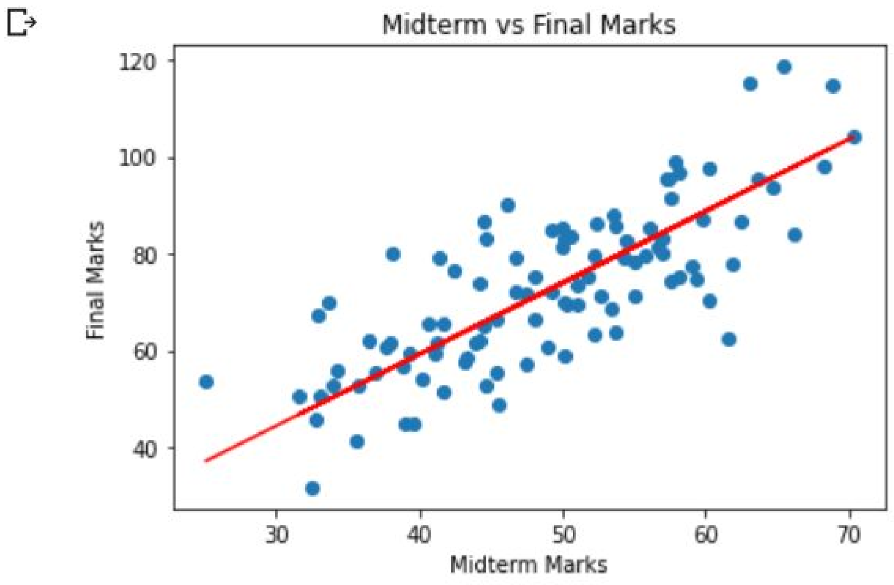
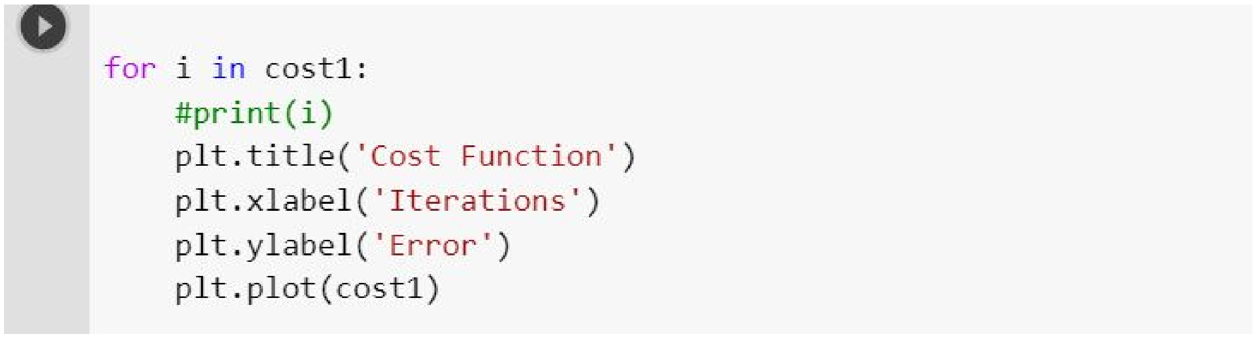


Figure C.2. *Code for Plotting Data-Set and Regression Line for 100 Iterations.*

Compared to section **B.2**, where our best-fit line was decreasing and very inaccurate, we can conclude that with the help of the gradient descent derivatives, we were able to find the most fitted graph for our model.

C.3. Error at Each Iteration

We've already defined our error function previously, in section **C.1**. We have also used this definition to append our error values per iteration into our *Cost1* array, we can use this array to now plot our error graph. The following displays the code and output for the cost function.



```
for i in cost1:
    #print(i)
    plt.title('Cost Function')
    plt.xlabel('Iterations')
    plt.ylabel('Error')
    plt.plot(cost1)
```

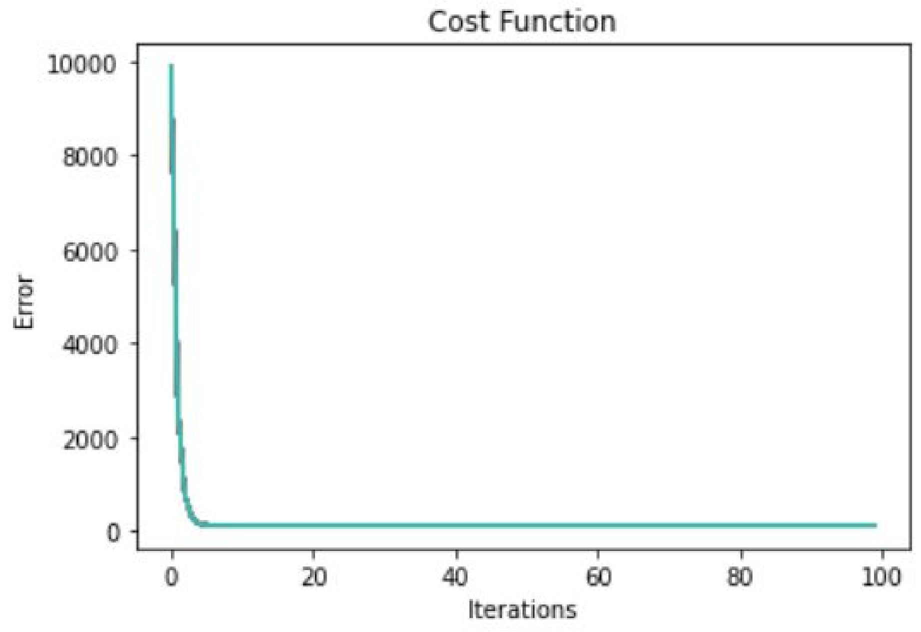


Figure C.3. *Code and Plot of Cost Function.*

As we can see in our error graph above, also in accordance with the output in section **C.1**, when we have our first iteration, our error is above 9000. This is true seeing as our output data shows that for the first iteration, our error is equal to 9873.2. This graph concludes that as our model marches downhill through the values of our slope, our error drastically decreases. With only a hundred iterations, we steadily approach our most minimum error. In the next section we will determine what happens with our error graph when we run it through 2000 iterations.

D. Linear Regression with 2000 Iterations

We now want to conduct what we have already conducted in section **C** but with 2000 iterations and observe what changes.We use the same data and the same code, but this time we only change the iterations from 100 to 2000. The code is the same minus the change of iterations, hence this report will not display the code entirely. Though, the code can be found within the Google notebook.

D.1. Regression Line for 2000 Iterations

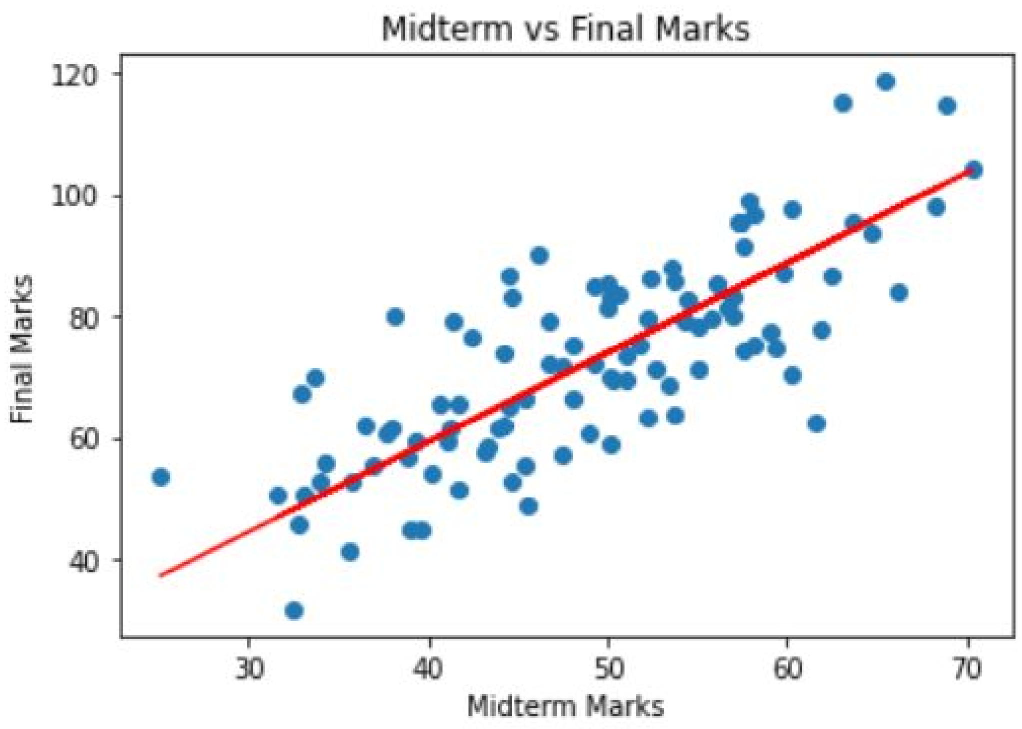


Figure D.1. *Plot of Regression Line for 2000 iterations.*

D.2. Cost Function for 2000 Iterations

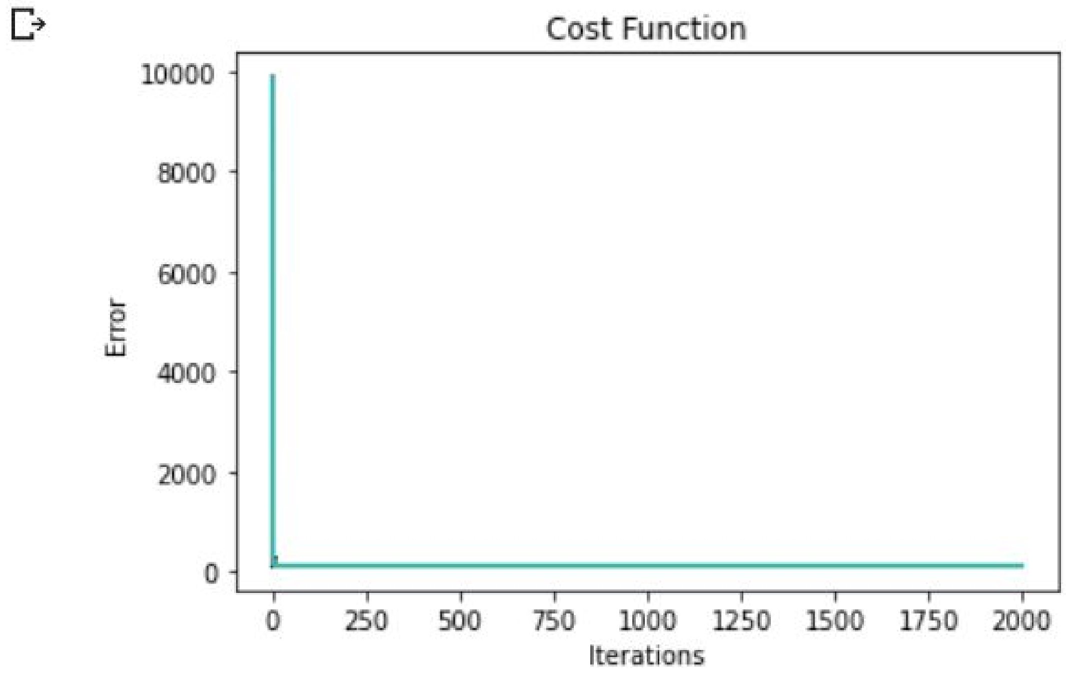
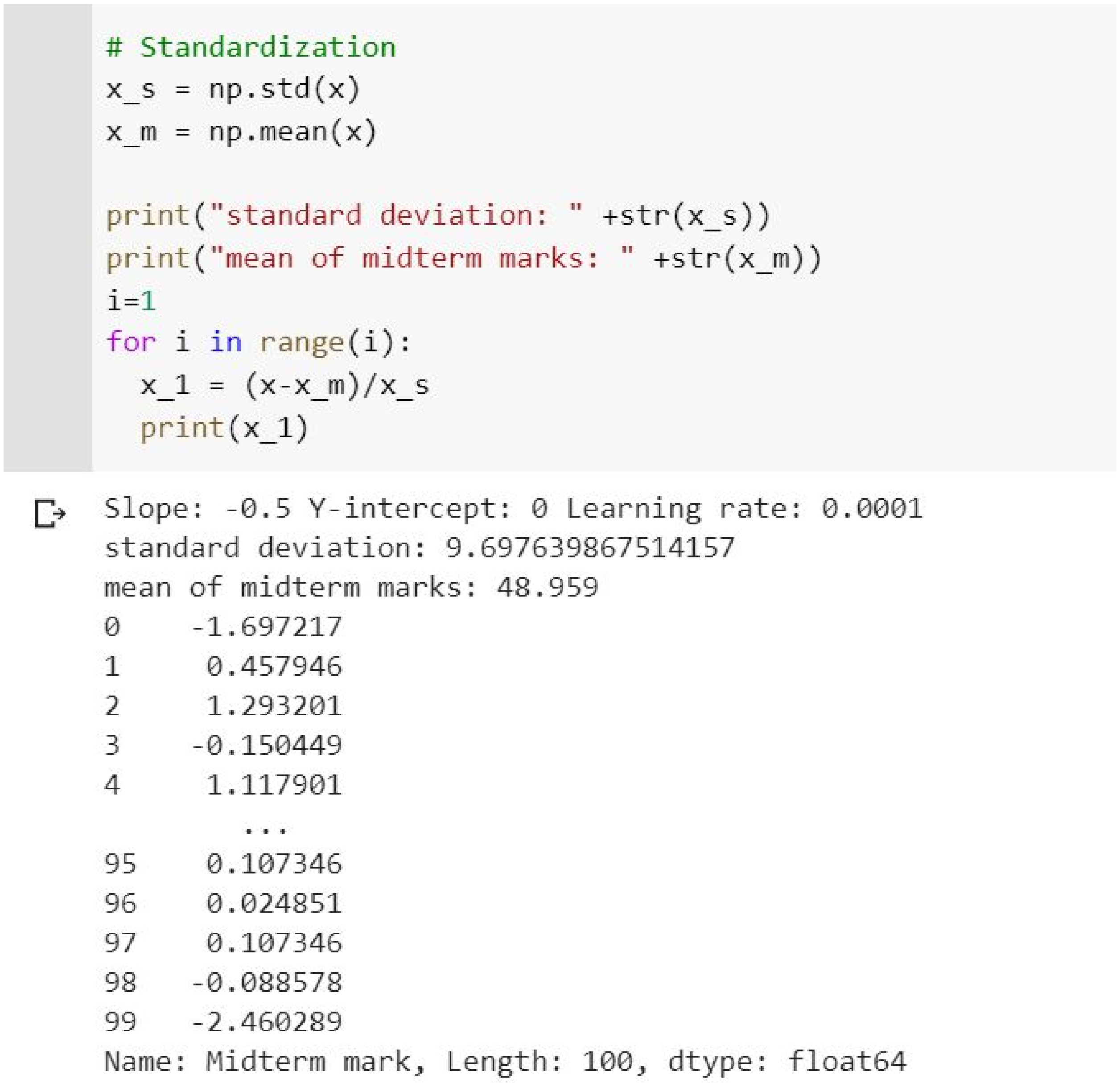


Figure D.2. *Cost Function Graph for 2000 Iterations.*

The cost function for 2000 iterations is displayed above. Our graph looks trickier to read, seeing as we have iterated a large number of times. Despite the large number of iterations, we are still able to conclude that the minimum error still equals what we initially had found for 100 iterations. The larger the iterations, the harder it is to find the rightful minimum error. With smaller iterations, we are easily able to determine the closest approximation of minimum error.

E. Standardization

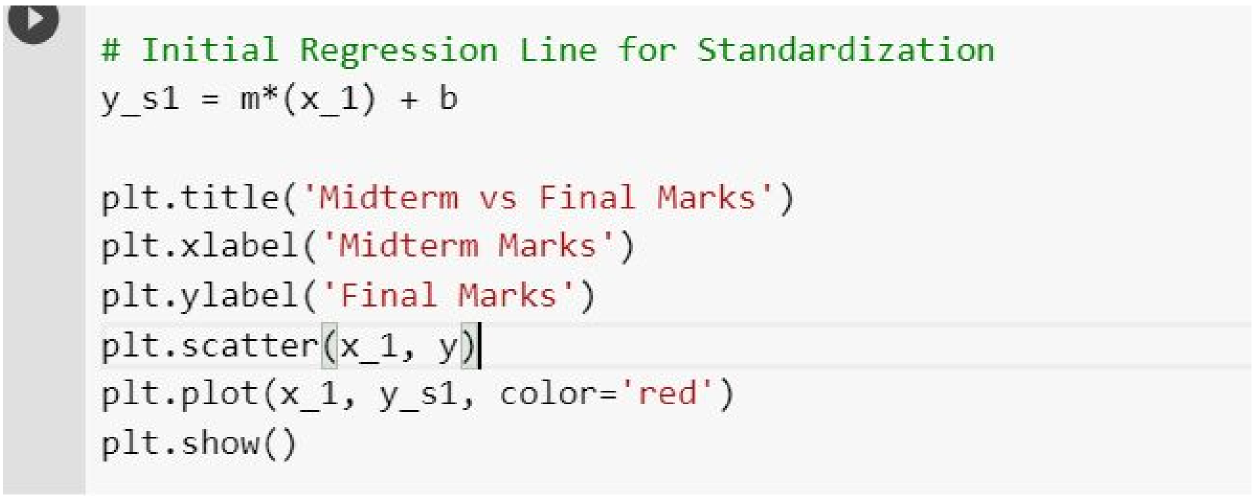
So far, through the report, we have worked with unstandardized values. We now use standardized values of x to run our model and obtain the regression line and cost function to see what happens. As mentioned in ***theory***, the standardized variable will have a zero mean and unit standard deviation. The code below uses the numpy module to determine the standard deviation and the mean of our independent values of x. These values are printed below the code for reference. In order to find our standardized x' value, we run the standardized function through a for loop and print the results. The standardized x (x') values are displayed below the code.



```
# Standardization
x_s = np.std(x)
x_m = np.mean(x)

print("standard deviation: " +str(x_s))
print("mean of midterm marks: " +str(x_m))
i=1
for i in range(i):
  x_1 = (x-x_m)/x_s
  print(x_1)
```

```
Slope: -0.5 Y-intercept: 0 Learning rate: 0.0001
standard deviation: 9.697639867514157
mean of midterm marks: 48.959
0    -1.697217
1     0.457946
2     1.293201
3    -0.150449
4     1.117901
        ...
95    0.107346
96    0.024851
97    0.107346
98   -0.088578
99   -2.460289
Name: Midterm mark, Length: 100, dtype: float64
```

Figure E.1. *Code and Output for Initializing Standardized Value.*



```
# Initial Regression Line for Standardization
y_s1 = m*(x_1) + b

plt.title('Midterm vs Final Marks')
plt.xlabel('Midterm Marks')
plt.ylabel('Final Marks')
plt.scatter(x_1, y)
plt.plot(x_1, y_s1, color='red')
plt.show()
```

```
Slope: -0.5 Y-intercept: 0 Learning rate: 0.0001
standard deviation: 9.697639867514157
mean of midterm marks: 48.959
```

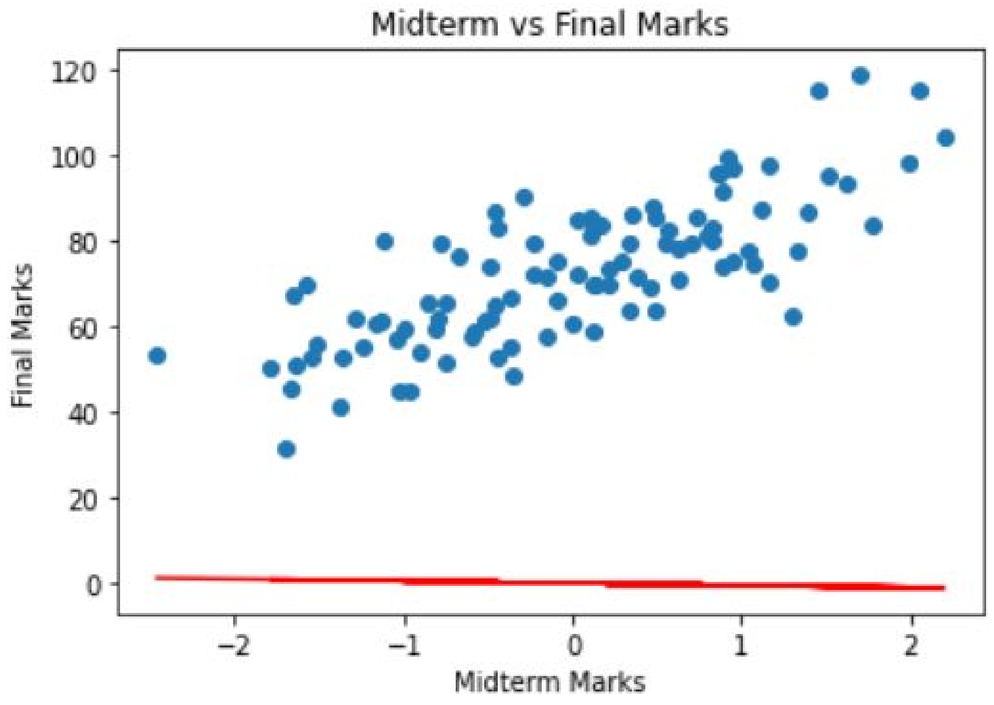
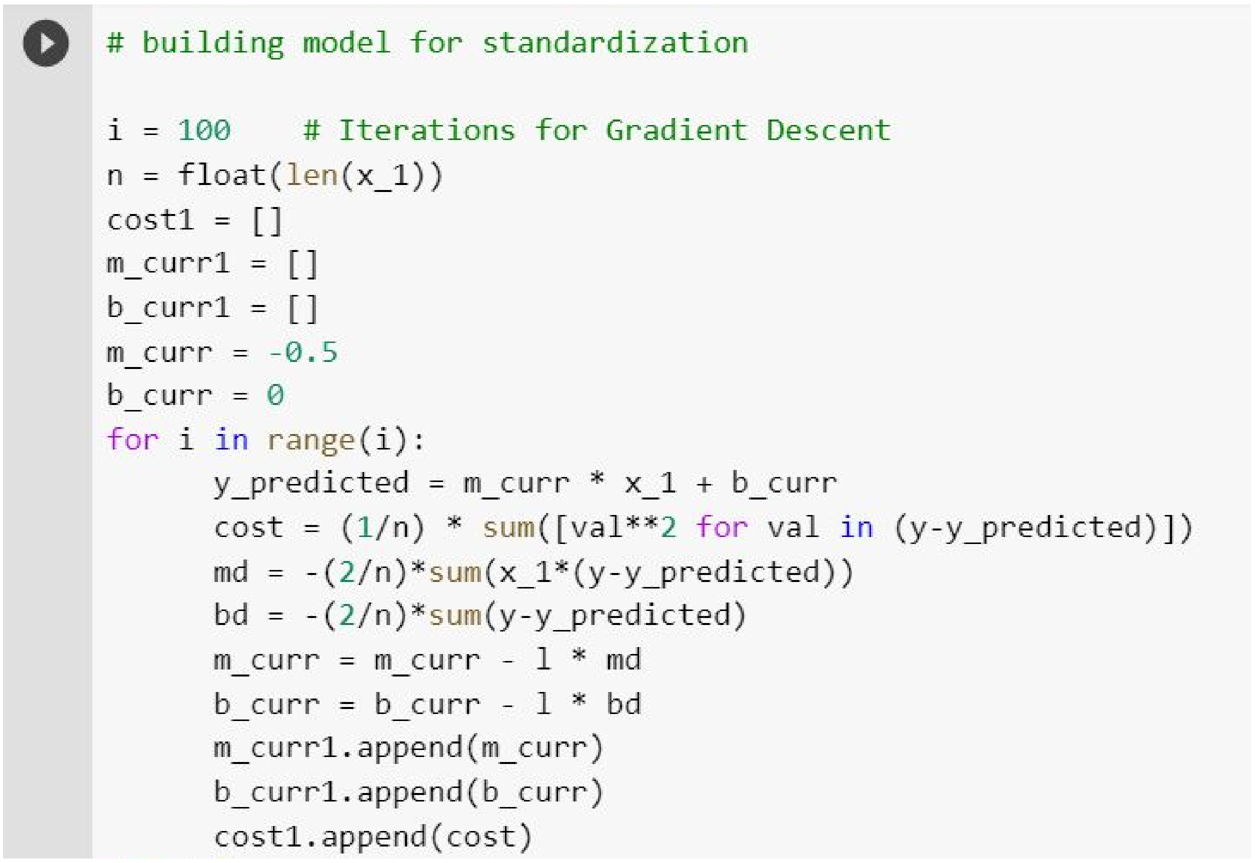


Figure E.2. *Code and Plot for Initial Regression Line.*

Similarly with what we've done before, we determine the regression line using the standardized x values. We haven't yet trained our model and we don't know what the best slope or y-intercept is, we follow the application manual (m= -0.5 and b = 0) and find the initial regression line.

Looking at ***Figure E.2***, our best-fit line is extremely inaccurate and filled with error. This problem results from an untrained model and us using a slope which is definitely not precise.

We will now try to train our model and find a more accurate slope. The code below is similar to the previous section, however we use our standardized x value rather than the initial. This code runs for the duration of 100 iterations.



```
# building model for standardization

i = 100    # Iterations for Gradient Descent
n = float(len(x_1))
cost1 = []
m_curr1 = []
b_curr1 = []
m_curr = -0.5
b_curr = 0
for i in range(i):
      y_predicted = m_curr * x_1 + b_curr
      cost = (1/n) * sum([val**2 for val in (y-y_predicted)])
      md = -(2/n)*sum(x_1*(y-y_predicted))
      bd = -(2/n)*sum(y-y_predicted)
      m_curr = m_curr - l * md
      b_curr = b_curr - l * bd
      m_curr1.append(m_curr)
      b_curr1.append(b_curr)
      cost1.append(cost)
```

Figure E.3.a. *Code for Building Model for 100 Iterations.*

Similarly with previous sections, we have obtained the graphs (***Figure E.3.b*** and ***Figure E.3.c***) for the regression line and the cost function for 100 iterations. These graphs are displayed below. Looking at the regression line, we can see that the output hasn't changed much even with a trained model of 100 iterations. Our output is the same as it was when we ran our initial model for standardization. Looking at the cost function, we can confidently confirm that the minimum error rate is hard to determine, compared to our unstandardized model before. With only 100 iterations, we may be able to say that minimum error is below 500. Let's try to run it for 2000 iterations and see how our graphs change.

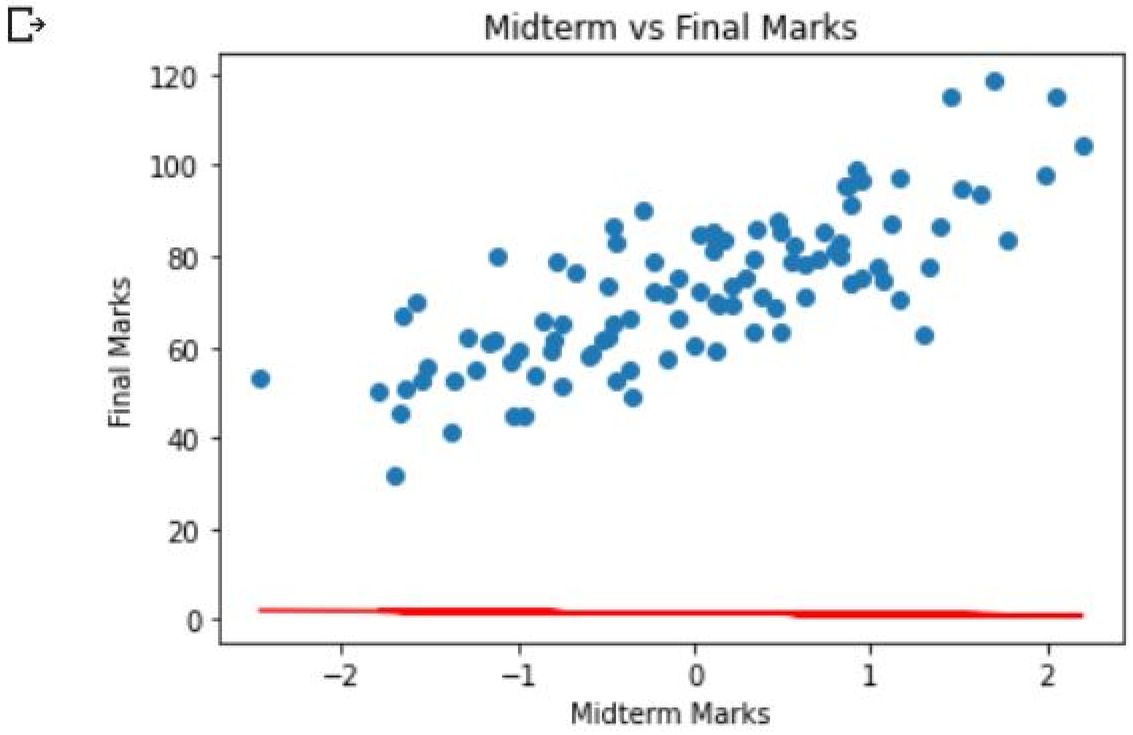


Figure E.3.b. *Standardized Linear Regression for 100 Iterations.*

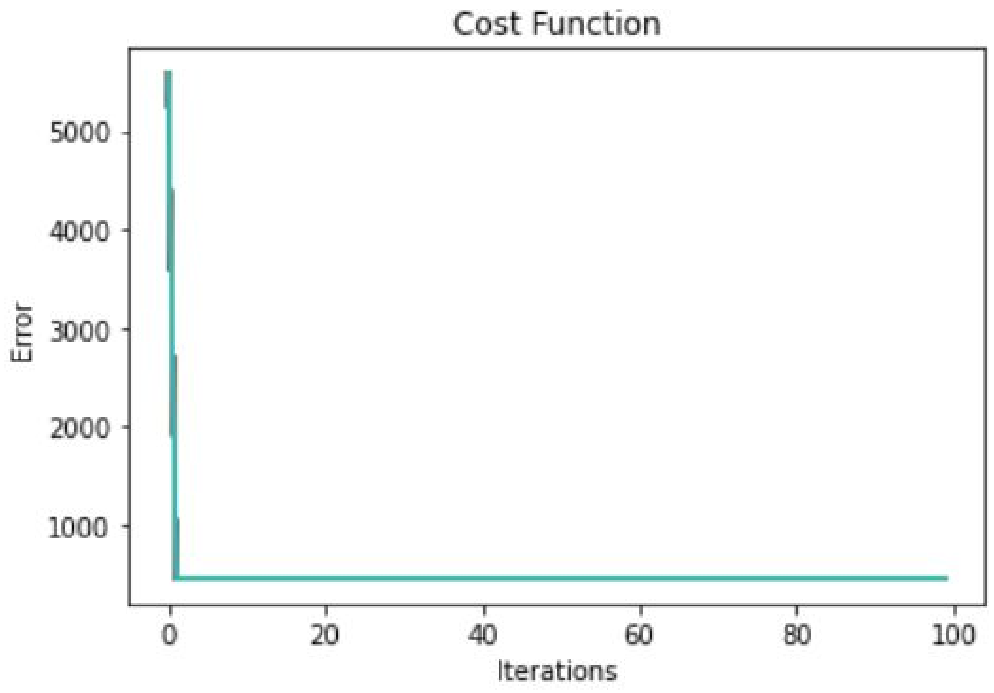


Figure E.3.c. *Standardized Cost Function for 100 Iterations.*

In ***Figure E.4.a*** and ***Figure E.4.b***, we have the produced graphs for 2000 iterations. The code we've run is similar to the previous section, minus the different iterations. Looking at the regression line below, we can say that with the higher number of iterations, our best-fitted line is becoming closer to precise than it was when we ran 100 iterations. Looking at the cost function graph, we can also see how the graph becomes even harder to read and our minimum error rate looks larger than for 100 iterations. We are getting closer to the minimum error value when we ran our model for 100 iterations. Hence, we can conclude that for standardized linear regression, the higher the iterations, the further away we are to determining minimum error. The lower the iterations, the closer we may be from producing an accurate model.

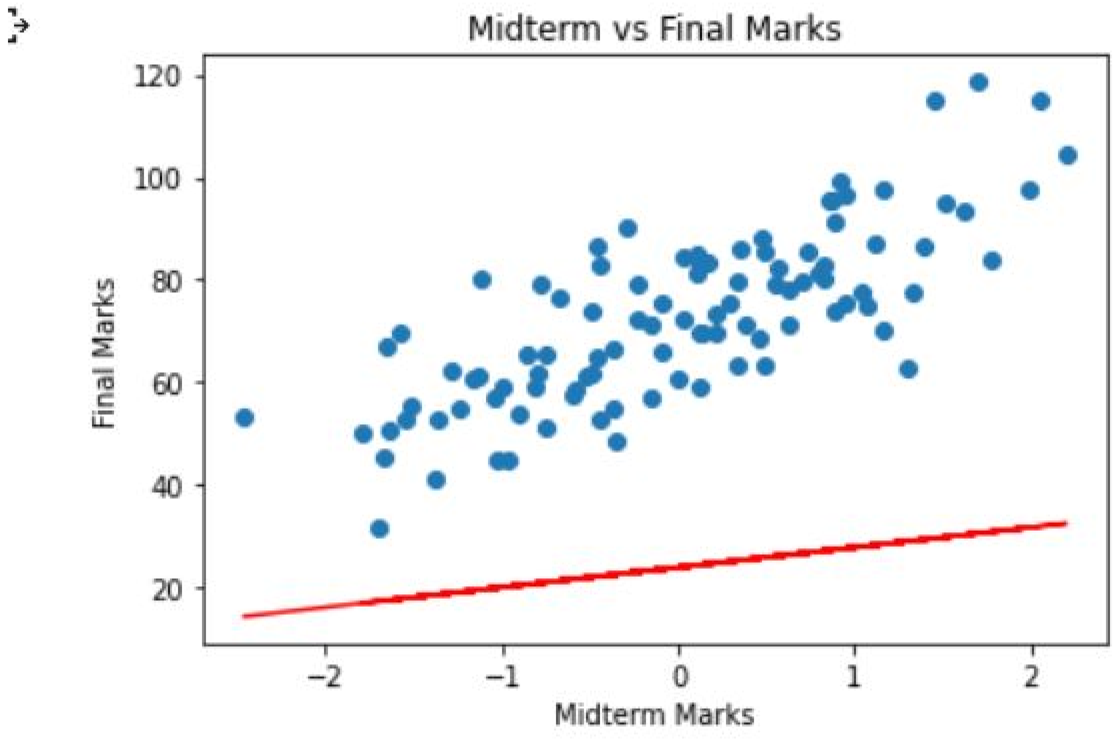


Figure E.4.a. *Standardized Linear Regression for 2000 Iterations.*

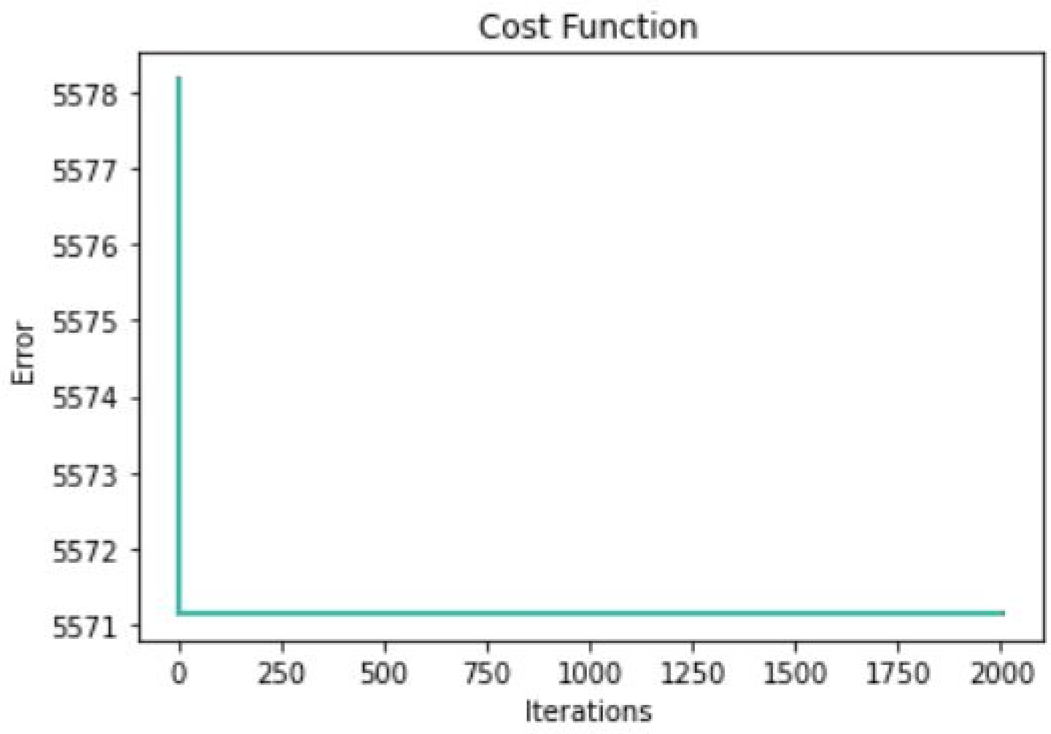


Figure E.4.b. *Standardized Cost Function for 2000 Iterations.*

F. Changing Learning Rate to 0.1

Theoretically, when we increase the learning rate, we may step over the minimum error [1]. Changing the learning rate to 0.1 for 50 iterations yields the following cost function graph in ***Figure F.1***. 100 iterations was producing an error: `OverflowError: (34, 'Numerical result out of range').` Hence, 50 iterations were used. A cost function for 50 iterations when learning rate is 0.0001 is also produced and displayed in ***Figure F.2*** for comparison. Looking at ***Figure F.1***, we can see that the error increases with each iteration. We can properly see the minimum error for when we use 0.0001 as our learning rate as opposed to 0.1. Theoretically, learning rate is meant to give us the rate of speed where the gradient moves during the descent algorithm. When we set our learning too high (0.1, in our case) we receive an unstable result, our model is learning too fast from the gradient and we can't comprehend where the minimum error rate is. If we set it too low, it may result in a slower convergence rate. However, looking at ***Figure F.2***, we can see that our graph reaches convergence steadily. It may not be accurate, it is still better than using 0.1 as our learning rate. Setting our learning rate to zero would result in a straight line, meaning our model is not learning anything from the gradient.

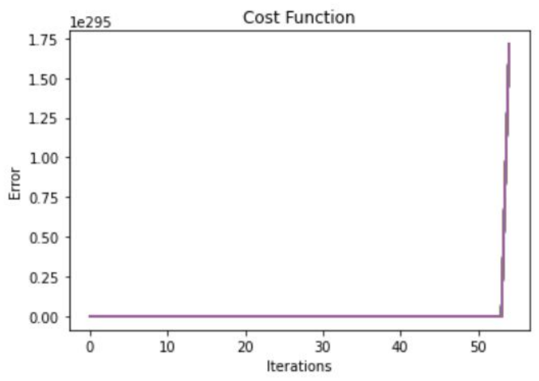


Figure F.1. *Cost Function using 0.1 Learning Rate.*

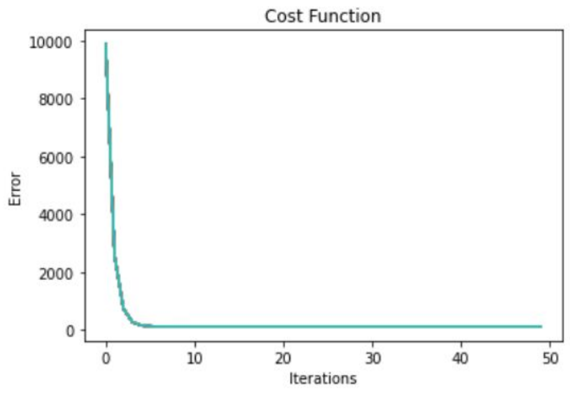


Figure F.2. *Cost Function using 0.0001 Learning Rate.*

References

[1] D. Bhalla, “Standardization”, ListenData, December 2016. [Online].

[2] S. Mainkar, “Gradient Descent in Python”, Medium, August 2018. [Online].

[3] Prabhu, “Linear Regression Simplified”, Medium, May 2018. [Online].