Fahad Fiaz (303141) Exercise 0

Q1.1

A) Word count program:

1) Open file in read mode

```
file = open("word_count.txt", "r")
```

2) Read data from file and split it into words. Here we have not passed any argument to split function. In this case split function will split based on default separator which is any whitespace. It will remove any leading or trailing whitespaces with words.

```
totalWords = file.read().split()
```

3) List of words to exclude.

```
stopwords = ['the', 'a', 'an', 'be']
```

4) List comprehension to remove stop words.

```
reduced = [words for words in totalWords
           if words.lower() not in stopwords]
```

5) Dictionary subclass Counter used for counting occurrence of words. most_common function which return list of the n most common elements and their counts.

```
Top10wordcount = dict(
    Counter(reduced).most_common(
        10))
```

6) Using pre-defined style provided by Matplotlib.

```
plt.style.use('fivethirtyeight')
```

7) Set title for the axes.

```
plt.title("Occurances of Unique Words")
```

8) Set the label for the x-axis.

```
plt.xlabel("Words")
```

9) Set the label for the y-axis.

```
plt.ylabel("Count")
```

10) Used to give padding

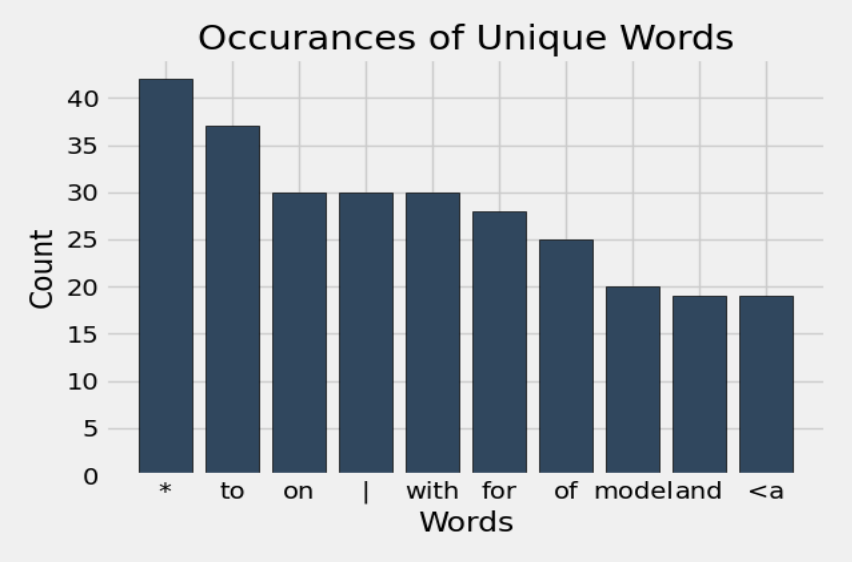
```
plt.tight_layout(0)
```

11) In this case visualization of result looks better with bar graph so I have drawn a bar graph.

```
plt.bar(Top10wordcount.keys(), Top10wordcount.values(),
        color='#30475e', edgecolor="black")
```

12) Show chart

```
plt.show()
```



This graph shows most occurring word among top 10 occurring words is “*” and least occurring word is “<a”.

My implementation of counter function:

```
def my_word_count(words):
    counter = {}
    for word in words:
        if not word in counter:
            counter[word] = 1
        else:
            counter[word] += 1

    print(counter)
```

B) Matrix Multiplication:

1) Initialize A matrix of dimension 100 × 20 with random values.

```
A = np.random.random((100, 20))
```

2) mean and standard deviation.

```
mu, sigma = 2, 0.01
```

3) This function generates a vector of dimension 20 x 1 with mean = 2 and standard deviation = 0.01 and sample of numbers drawn from the normal distribution .

```
v = np.random.normal(mu, sigma, (20, 1))
```

4) Iterative multiply (element-wise) each row of matrix A with vector v and sum the result of each iteration in another vector c.

```
c = np.dot(A, v)
```

5) Calculate standard deviation of the new vector c.

```
std = np.std(c)
```

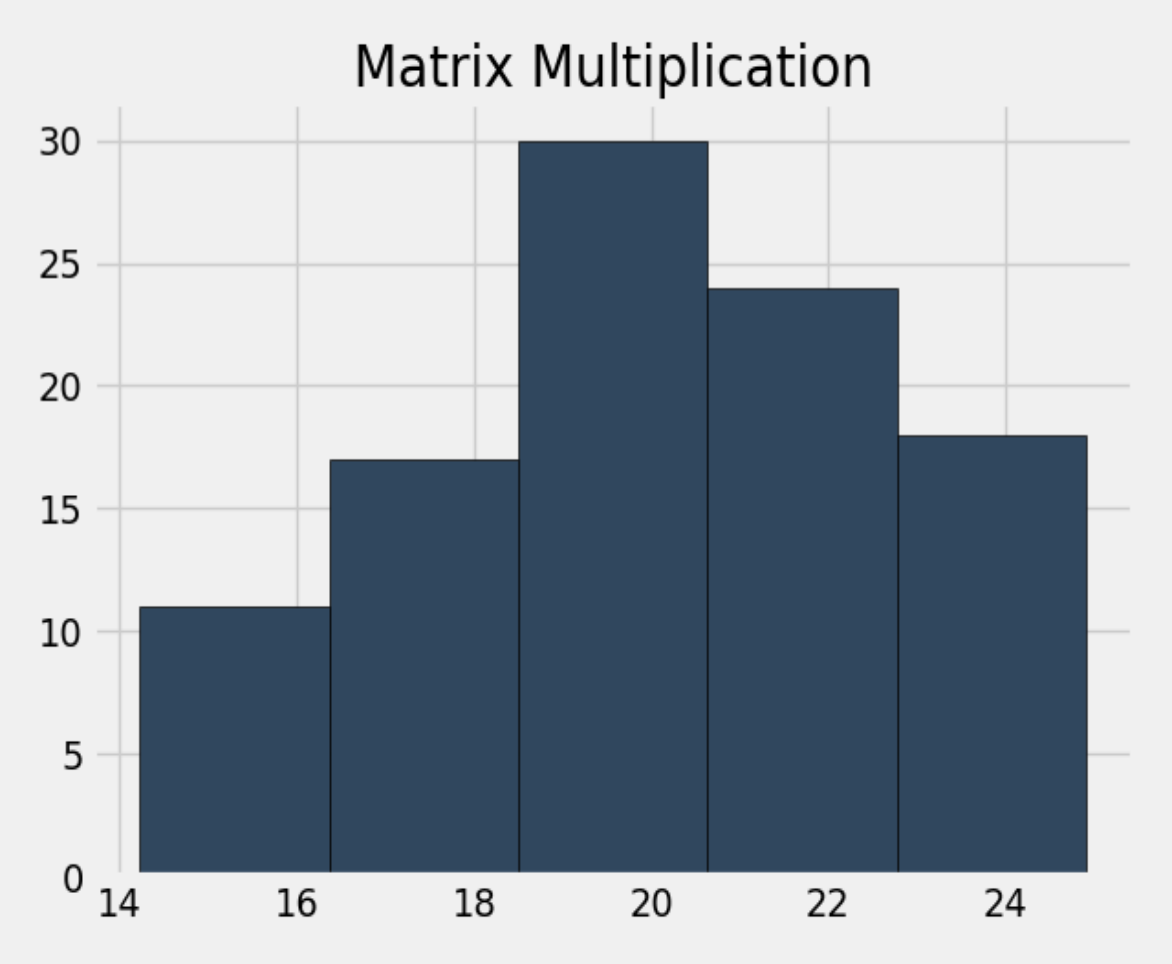
6) Calculate mean of the new vector c.

```
mean = np.mean(c)
```

7) Draw histogram for vector with 5 bins.

```
plt.style.use('fivethirtyeight')  # using pre-defined style provided by Matplotlib.
plt.title("Matrix Multiplication")  # Set title for the axes.
plt.hist(c, bins=5, color='#30475e', edgecolor="black")

plt.show()
```



Q1.2

Linear Regression through exact form:

1) Following function generate 3 sets of simple data. All three matrix **A, B, C** has mean 2 and standard deviation in the order [0.01,0.1,1]. The function np.random.normal generates a matrix of specific shape with specific mean and standard deviation and sample of numbers drawn from the normal distribution.

```
2) def generate_dataset(mu, shape, *sigmas):
       A, B, C = np.random.normal(mu, sigmas[0], shape), \
                 np.random.normal(mu, sigmas[1], shape), \
                 np.random.normal(mu, sigmas[2], shape)
       return A, B, C
```

```
mu = 2  # mean
sigma1, sigma2, sigma3 = [0.01, 0.1, 1]  # standard deviation
shape = (100, 2) #dimensions of matrix
A, B, C = generate_dataset(mu, shape, sigma1, sigma2, sigma3)
```

3) Function that take matrix as input, calculate and return values of $\beta 0$ and $\beta 1$.

```
def learn_simple_line_regression(matrix):
    x_mean = np.mean(matrix, axis=0)[0] taking mean of first column in matrix
    y_mean = np.mean(matrix, axis=0)[1] taking mean of second column in matrix
    b1 = np.sum(np.multiply(matrix[:, 0] - x_mean, matrix[:, 1] - y_mean)) /
np.sum((matrix[:, 0] - x_mean) ** 2) #calculating beta1
    b0 = y_mean - np.multiply(b1, x_mean) #calculating beta0
    return b0, b1
```

4) Function to calculate the predictions for each training example in matrix. It uses training examples and β0, β1 values for prediction.

```
def predict_simple_line_regression(matrix, beta0, beta1):
    y_hat = beta0 + np.multiply(beta1, matrix[:, 0]) #calculating y_hat
    return y_hat
```

5) Function to plot line graph. This function takes input feature vector (x), actual prediction (y) and prediction done by using $\beta 0$ and $\beta 1$. Plt.plot function is used to Plot y versus x as lines (3rd argument in the function is the line style that we want on plot).

```
def draw_graph(x, y, prediction):
    plt.plot(x, y, '.', color="red")
    plt.plot(x, prediction, '-', color="green")
    plt.show()
```

6) This function calculate residual sum of squares (RSS) error. This tells us how our regression line fits the data. This function take actual values and model prediction values and return error between actual and predicted values.

```
def calculate_rss(y, prediction):
    error = np.sum((y - prediction) ** 2)
    print(error)
```

Effect of σ on the line that is predicted:

I have generated three dataset with following specifications:

Matrix	Mean(μ)	Standard Deviation
A	2	0.01
B	2	0.1
C	2	1

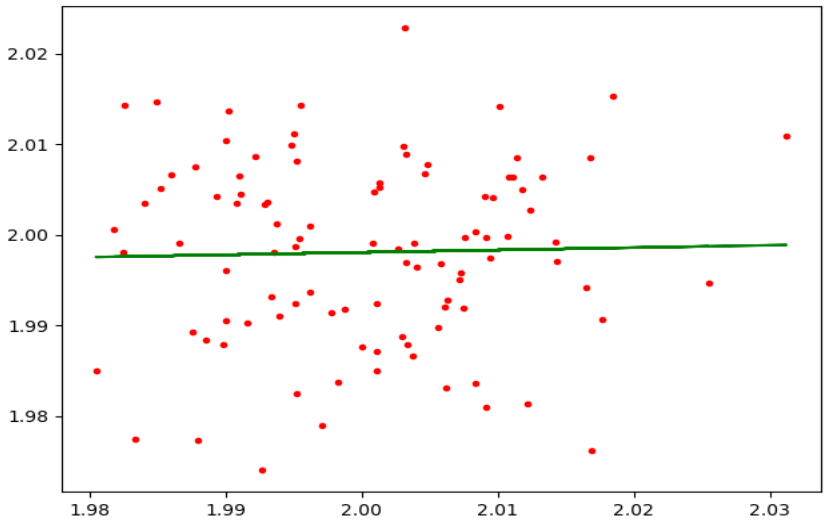
Smaller standard deviation make the distribution appear as a thinner curve so matrix A will have values that will be closely centered around the mean. Similarly if we increase the standard deviation then the distribution will have flat and wide curve, also centered around the mean. This means the points in the matrix C are more dispersed than in matrix A or B.

Matrix A:

Now when we do train and fit the model on matric A, the regression line better fit the data and the error is least.

```
b0, b1 = learn_simple_line_regression(A)
prediction = predict_simple_line_regression(A, b0, b1)
calculate_rss(A[:, 1], prediction)
draw_graph(A[:, 0], A[:, 1], prediction)
```

Error: 0.01

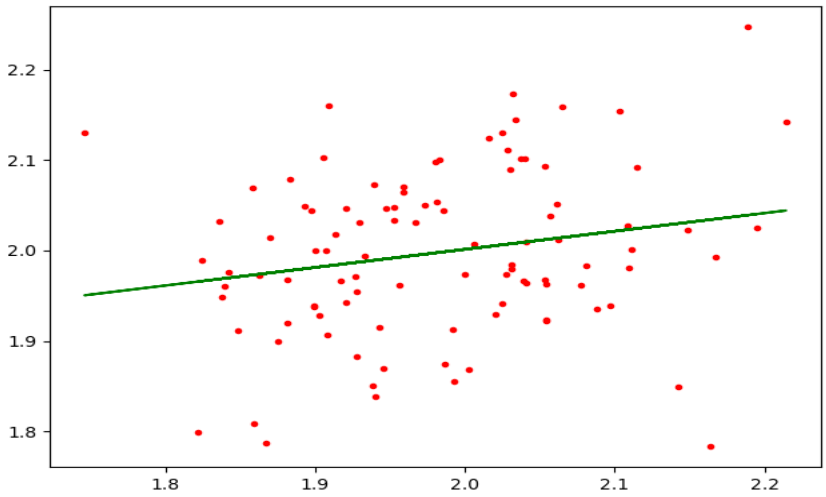


Matrix B:

Now when we do train and fit the model on matrix B, the regression line fit the data but the error increase on this data.

```
b0, b1 = learn_simple_line_regression(B)
prediction = predict_simple_line_regression(B, b0, b1)
calculate_rss(B[:, 1], prediction)
draw_graph(B[:, 0], B[:, 1], prediction)
```

Error: 0.80

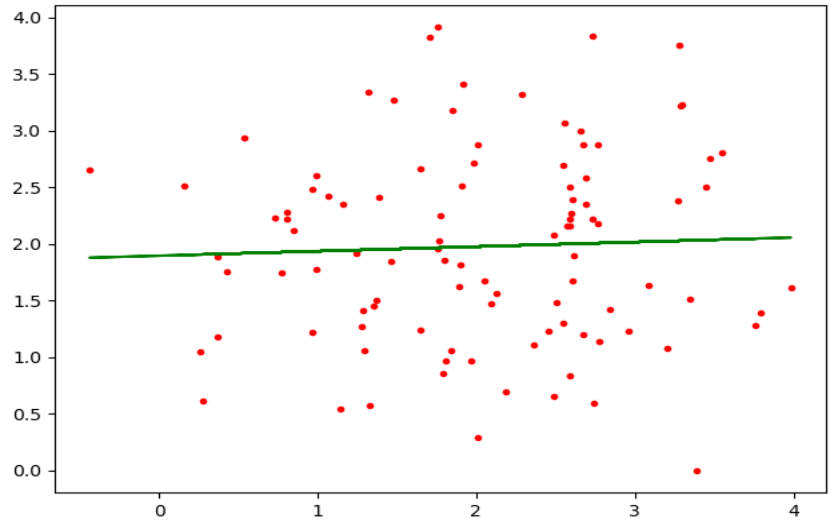


Matrix C:

Now when we do train and fit the model on matrix C, the regression line fit the data but our dataset points are more dispersed which lead to increase in error.

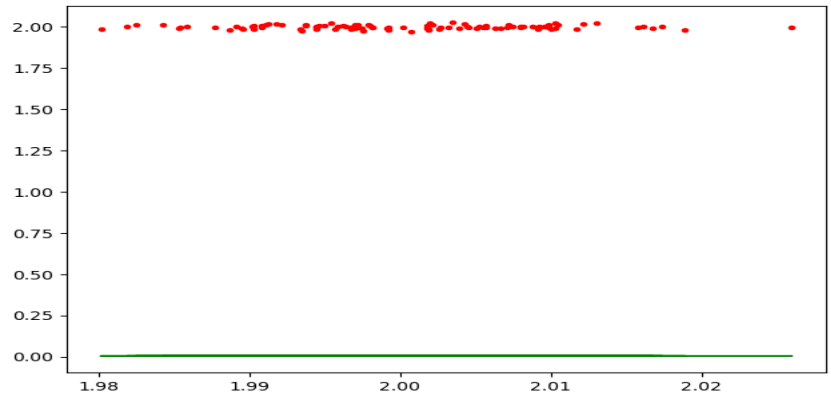
```
b0, b1 = learn_simple_line_regression(C)
prediction = predict_simple_line_regression(C, b0, b1)
calculate_rss(C[:, 1], prediction)
draw_graph(C[:, 0], C[:, 1], prediction)
```

Error: 73.7

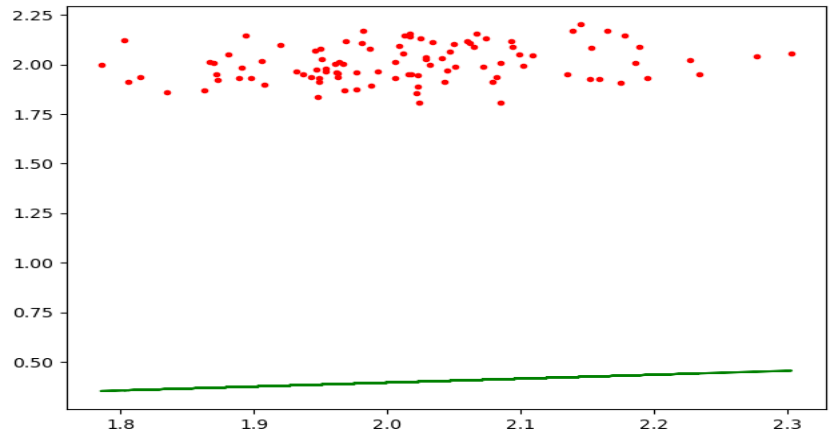


<u>If $\beta 0$ is set to zero then</u> regression line will goes through origin(0, 0) irrespective of σ.

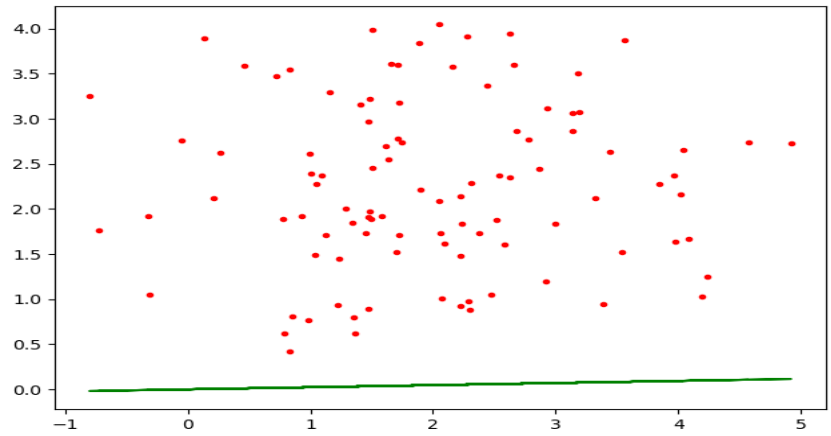
<u>σ =0.01</u>



σ =0.1:

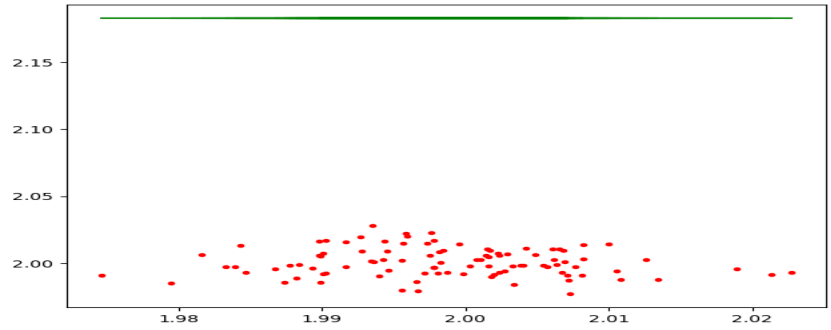


σ = 1:

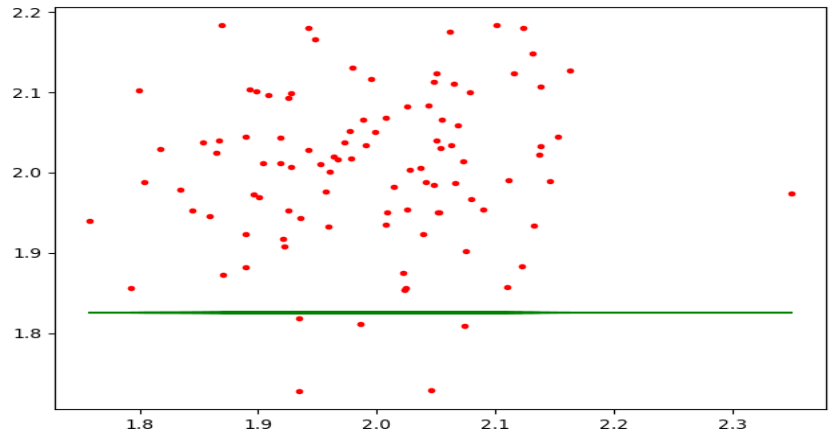


If *β*1 is set to zero then regression line slope becomes zero irrespective *σ*.

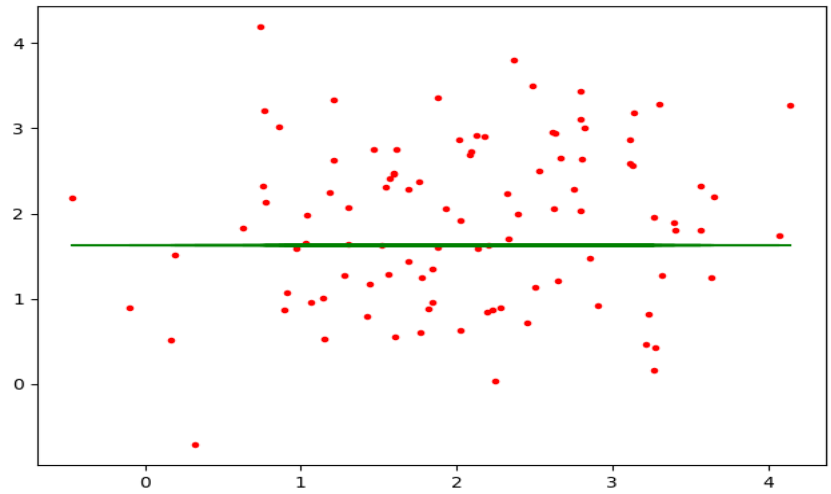
σ =0.01



σ =0.1



σ =1



numpy.linalg lstsq* for *learning parameter's:

Here vstack function combine two vectors vertically into single numpy array.

np.linalg.lstsq gives parameters of regression line.

```
def calculate_and_predict_simple_line_regression(matrix):
    Matrix = np.vstack([matrix[:, 0], np.ones(len(matrix))]).T
    m, c = np.linalg.lstsq(Matrix, matrix[:, 1], rcond=None)[0]
    print("Values generated by build in function lstsq m = {} , c = {}".format(m, c))
```

```
    newPrediction = m * matrix[:, 0] + c
    draw_graph(matrix[:, 0], matrix[:, 1], newPrediction) #my function to plot graph
```

My implementation of np.linalg.lstsq function. It return same values as np.linalg.lstsq function.

```
def my_own_implementation_of_lstsq(matrix):
    Matrix = np.vstack([matrix[:, 0], np.ones(len(matrix))]).T
    m, c = np.dot(np.dot(np.linalg.inv(np.dot((Matrix.T), Matrix)), Matrix.T),
matrix[:, 1])
    print("Values generated by my implementation m = {} , c = {}".format(m, c))
```