DS203 ASSIGNMENT
E2

SOURAV SREENATH
24B1841
BTECH-ENGINEERING PHYSICS
INDIAN INSTITUTE OF TECHNOLOGY
BOMBAY

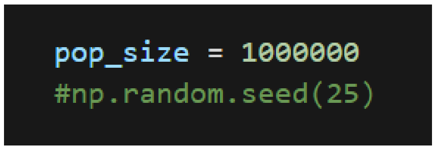
1. **SECTION 1:**
   Q.1:
   The population standard deviation and the sampling distribution's standard deviation is connected by the well known standard error formula.

$$SE = \frac{\sigma}{\sqrt{n}}$$

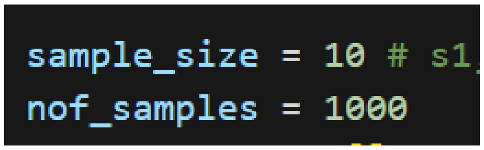
Here, SE is the Standard Error or the SD of the sampling distribution. The sigma represents the SD of the population.

For the given .ipynb file, we have a random population generator which for this question i have set to the normal distribution, but the beauty of the sampling distribution is that, whatever the population's distribution, the sampling distribution is always a normal distribution. n represents the sample size.

I ran the program and analyzed the data for different seeds of the populations and also for different sample sizes. The pictures are shown below, and my inference will follow.



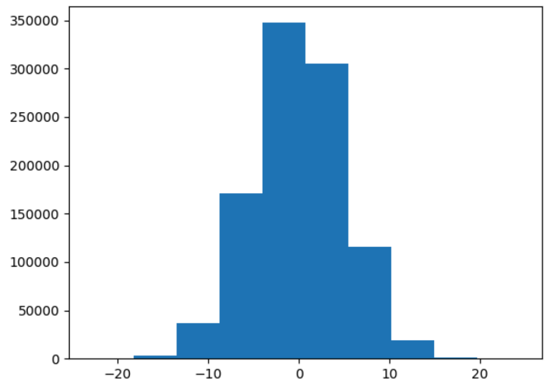
```
pop_size = 1000000
#np.random.seed(25)
```



```
sample_size = 10 # s1
nof_samples = 1000
```

For this particular question, i have only chosen the non-seed population variation in multiple runs. As the sample size is kept constant for this question, the only thing to check is whether both the SDs are proportional. The sample size variation is specifically asked in the 3rd question, hence i have left that out for that question. For this part, only the variation of the population is taken.

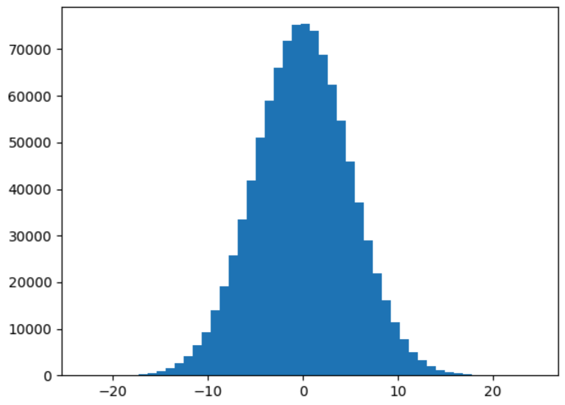
The below are the figures and data that i got.

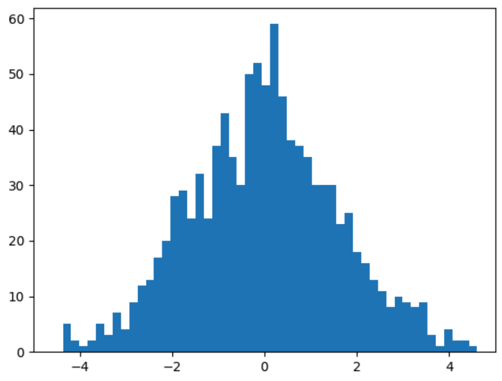


This is the prepared population data in Run1. The bin count here is 10, hence it is not that clear that it is a gaussian.

(X-axis : entries/data points
Y-axis : frequency of entries in an interval.)

This is also a graph of the prepared population data in Run1 but, to make the curve clearer, the bin size has been increased, ie: the interval lengths have been reduced, This is just for clarity. (Same x and y axes as the prev graph)

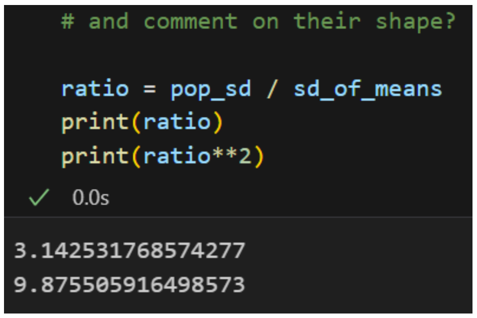




This is the sampling distribution with respect to the means. As per the given code, its around 1000 samples with each sample having randomly chosen values.

(X-axis : means
Y-axis : Frequency of means appearing interval wise)

This is the ratio i got for the first run. The ratio is approx sqrt(10) which is expected as the relation must be the constant and equal to the sqrt(sample size).

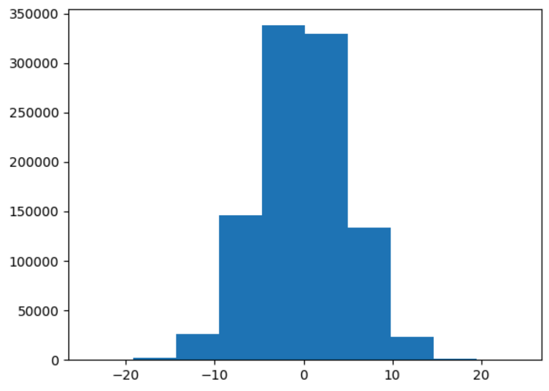
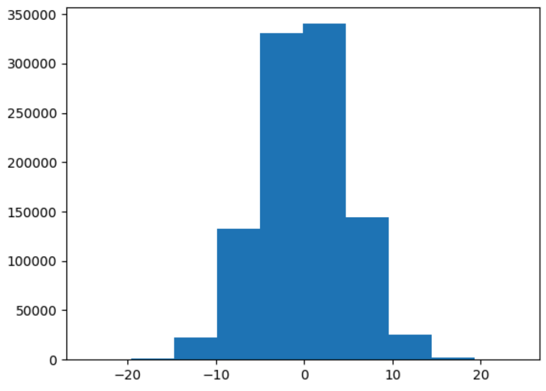
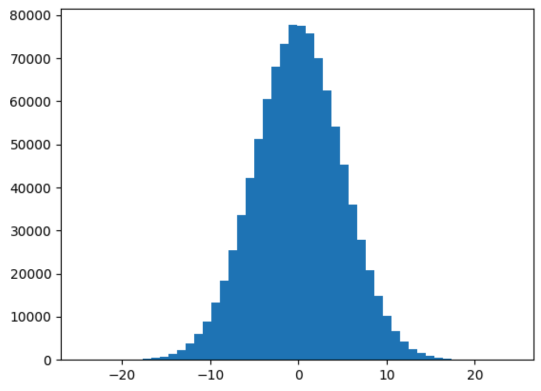
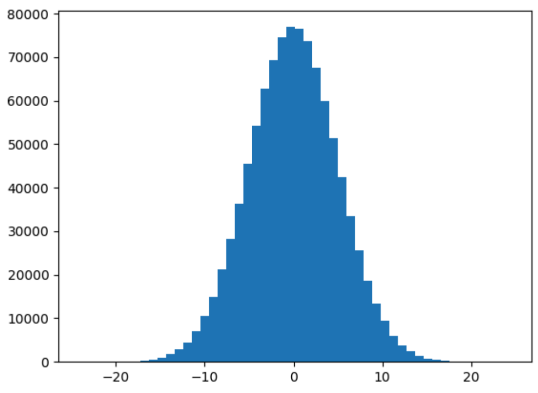
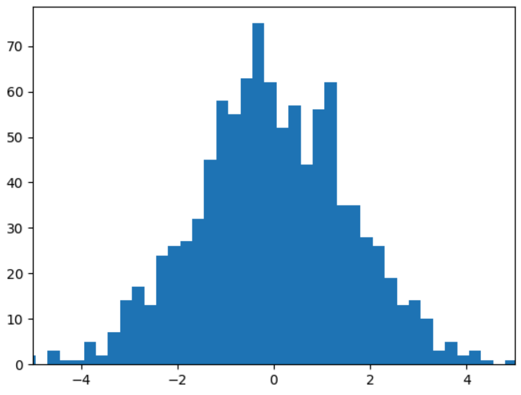
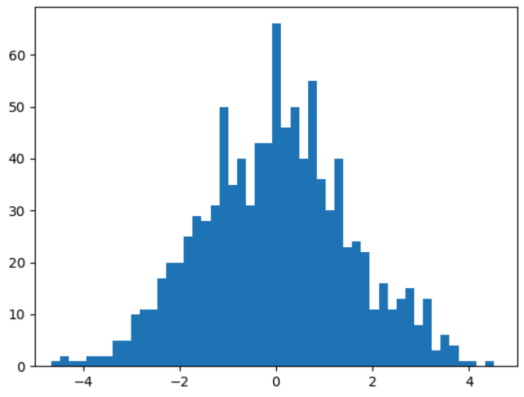
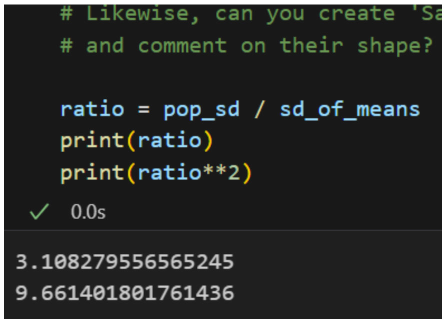


```
# and comment on their shape?

ratio = pop_sd / sd_of_means
print(ratio)
print(ratio**2)
```

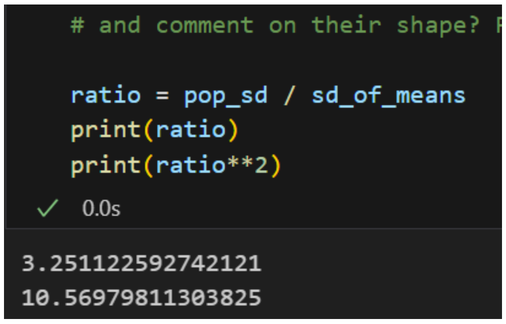
✓ 0.0s

```
3.142531768574277
9.875505916498573
```

Similarly, for the other 2 runs,

```
# Likewise, can you create 'Sa
# and comment on their shape?

ratio = pop_sd / sd_of_means
print(ratio)
print(ratio**2)
```
✓ 0.0s
```
3.108279556565245
9.661401801761436
```



```
# and comment on their shape? F

ratio = pop_sd / sd_of_means
print(ratio)
print(ratio**2)
```
✓ 0.0s
```
3.251122592742121
10.56979811303825
```

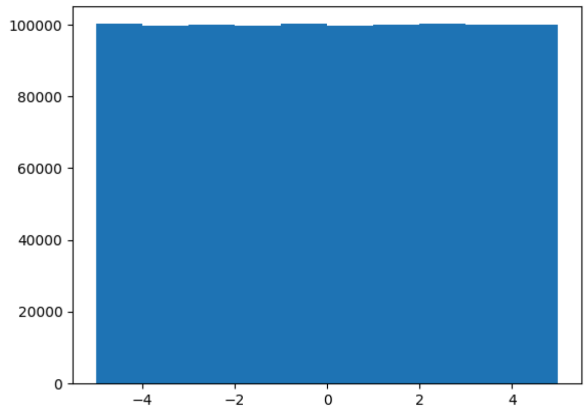
On the left side, you can see the results from Run2 and on the right, Run3. As expected, other than some noise related variations in the ratio, the ratio seems to be quite close to the expected ratio of sqrt(10). Here, I have only varied the runs and not changed or rather touched the sample size and that will be talked about in the 3rd question. The inference from this exercise is simply that the ratio seems constant and close to sqrt(10) for any random population set. So the inference from this question is that, the SDs are linearly related to each other by proportionalities under constant sample size.

Q.2:

The program has the functionality to take any random distribution for the population. The given distributions are gaussian, uniform, poisson, binomial, triangular, etc. I commented and uncommented each piece one by one and got the below distributions.

Uniform:

Quantities on the axes are the same as the previous exercises with no difference.

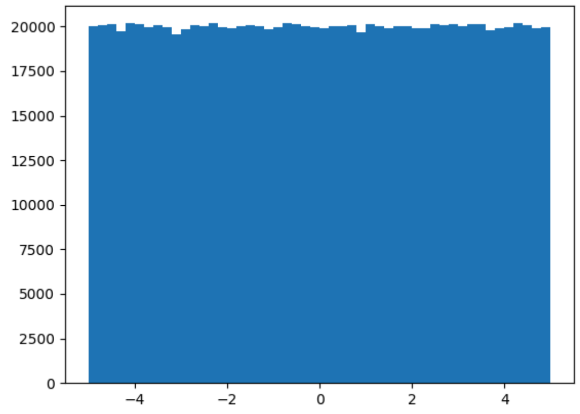


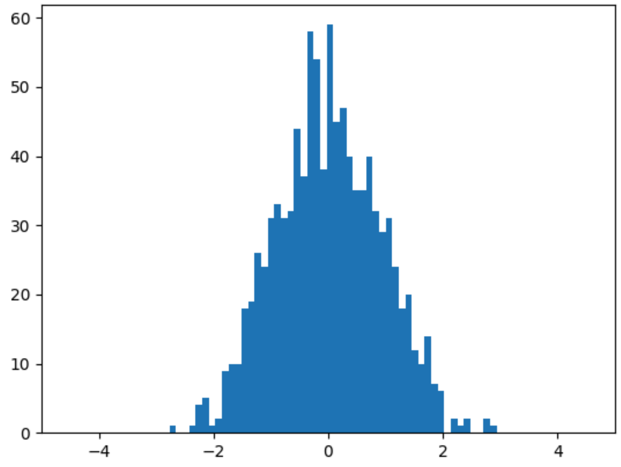
This is the first plot. It contains the population distribution for the uniform distribution with the default bin size.

(X-axis : entries/data points
Y-axis : frequency of entries in an interval.)

This is the second plot. This is also the population distribution but with a bigger bin number. But as the distribution is uniform, the data looks to close together in any situation.

(X-axis : entries/data points
Y-axis : frequency of entries in an interval.)

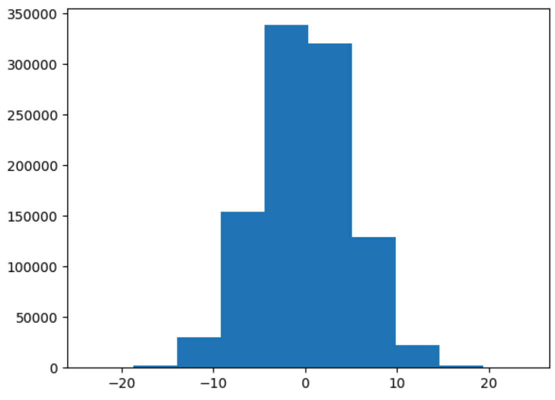


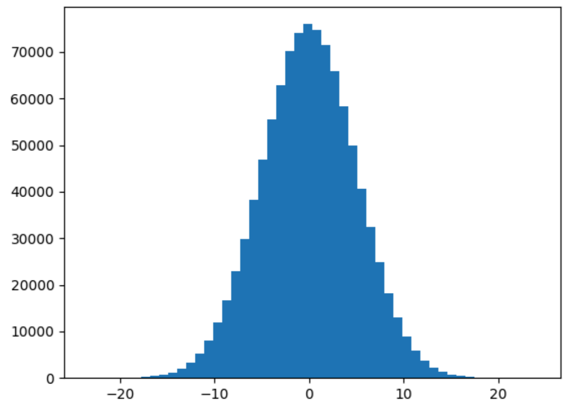


This is the final plot and this describes the sampling distribution for the uniform distribution case.

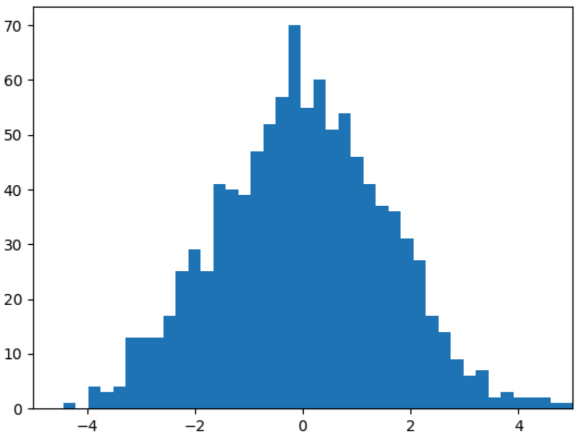
(X-axis : means
Y-axis : Frequency of means appearing interval wise)

Normal:



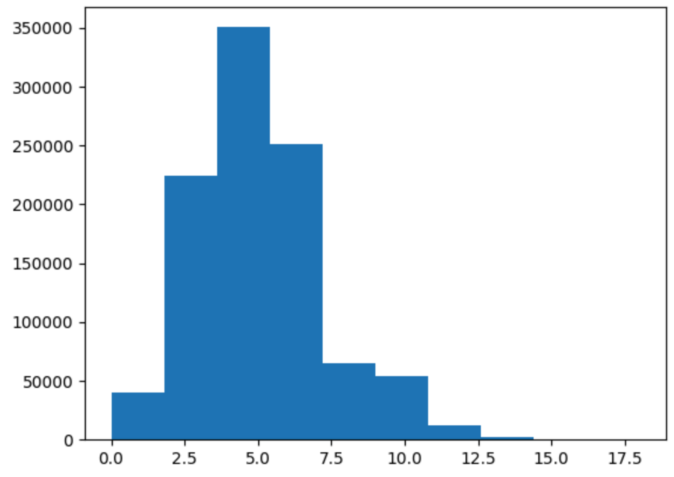


70000
60000
50000
40000
30000
20000
10000
0
−20
−10
0
10
20

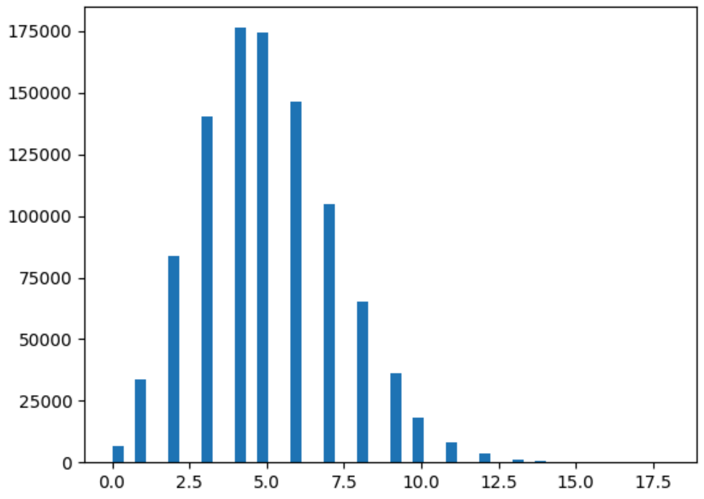


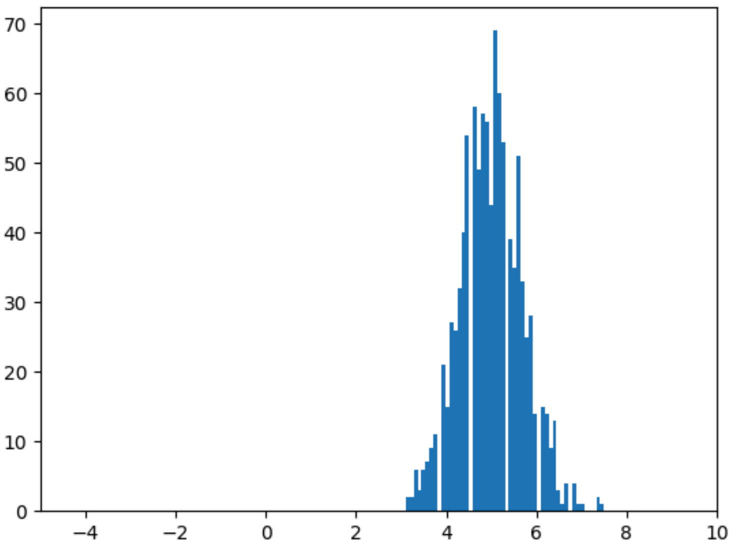
70
60
50
40
30
20
10
0
−4
−2
0
2
4

Poisson:



350000
300000
250000
200000
150000
100000
50000
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

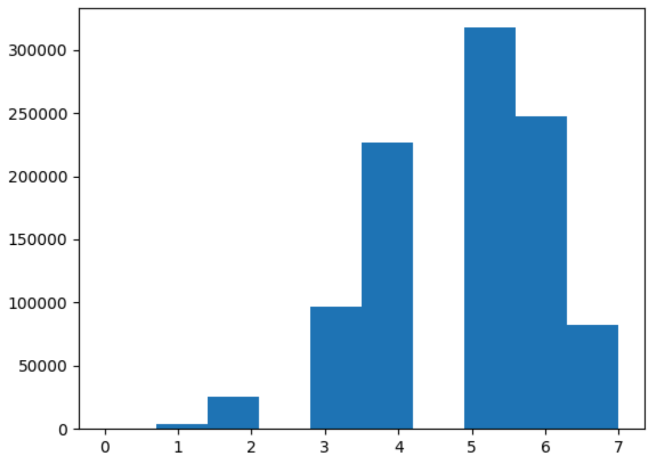


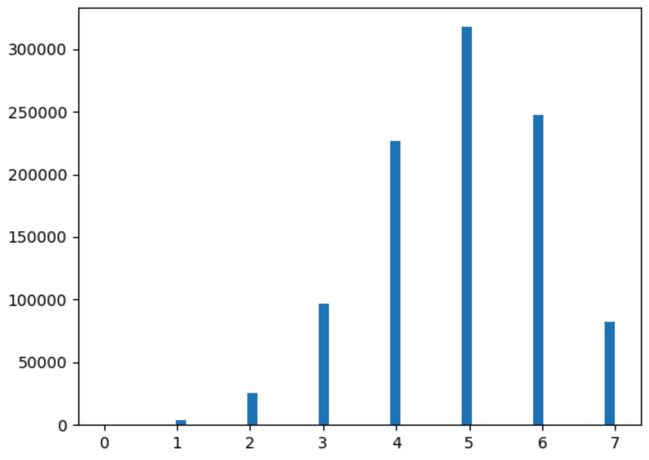


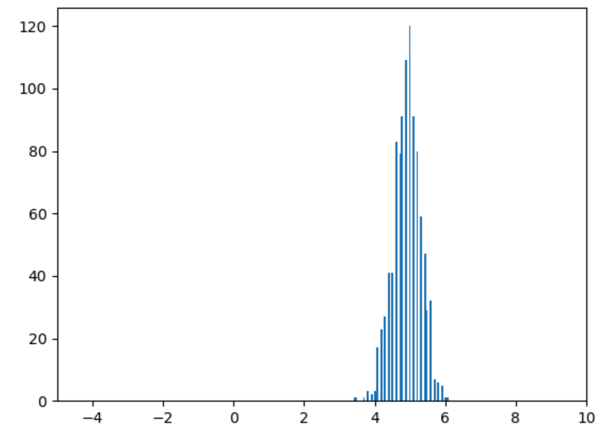
I had to change the xlim so that the entire graph could get captured in the 3rd plot.

The reason for the gaps will be discussed later in the analysis part of the Q.

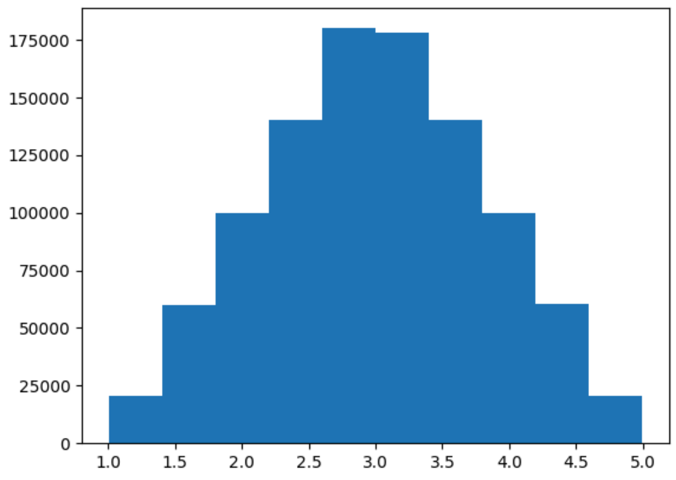
Binomial:

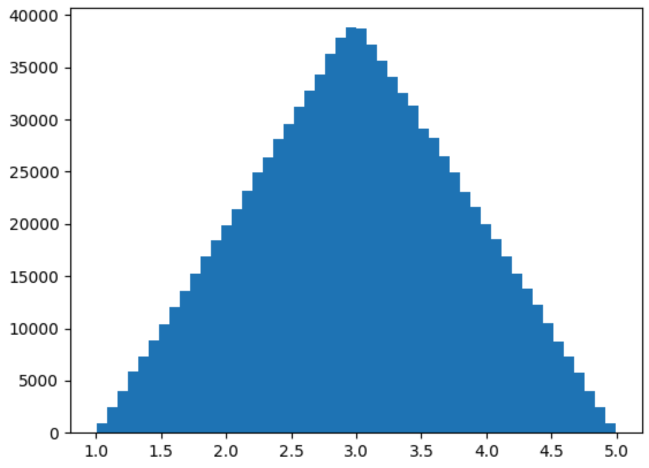


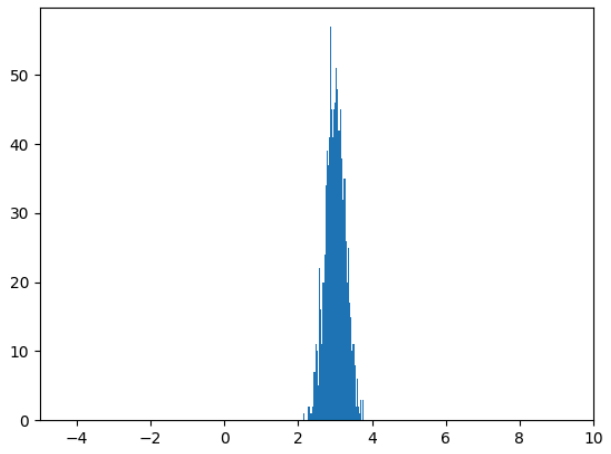




Triangular:


175000
150000
125000
100000
75000
50000
25000
0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


40000
35000
30000
25000
20000
15000
10000
5000
0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


50
40
30
20
10
0
−4
−2
0
2
4
6
8
10

Analysis:
The distributions provided are some of the most integral distributions in the entirety of statistics. The distributions still follow the SD proportionality relationships even though i have not emphasized it through pictures. The uniform distribution produces numbers between set limits for a given number of times, hence the plot looks like a filled rectangle. But still if we observe it, the population distribution might be uniform, but the sampling distribution still resembles a normal or gaussian curve.

The normal population distribution is quite straightforward where the population is modelled as a gaussian curve and after sampling, producing the sampling graph gives a gaussian curve.

The binomial and poisson distributions are special. This is because, till now we have only dealt with uniform or normal distributions, which are said to be continuous distributions. That means the probability for occurrence of the data point is continuous in some sense. But the poisson and binomial distribution produce data that is discrete. That means some values are favoured more than others in a discrete fashion. Hence, we see some holes in the graph and these are the less preferred values.

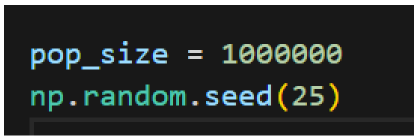
The triangular is also quite obvious on how it functions.

Now, onto the main point. Whatever the initial population distribution. We see that the final sampling distribution curve always resembles or better put, is a normal or gaussian curve. This is the central or main idea of the Central Limit Theorem. The distribution of means of samples always forms a gaussian curve. Let it be the discrete sequences or continuous sequences, or let it be a uniform sequence which has no pattern, still when taking the means of the samples selected, it always resembles a Gaussian curve. This is the main inference from this question. The validation of the Central Limit Theorem and the Sampling Distribution with respect to the mean.

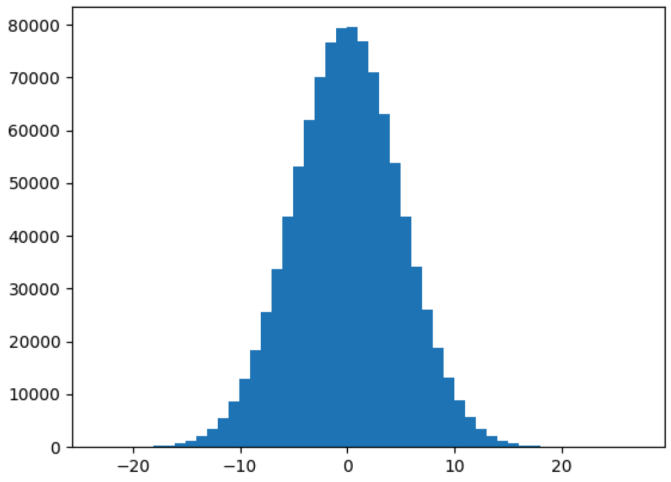
<u>Q.3:</u>
The next question deals with the question of what happens when we change our sample size. For this question, I have set the population distribution to the normal or gaussian curve with a set seed so that i can see what happens when the sample size changes with the same population.

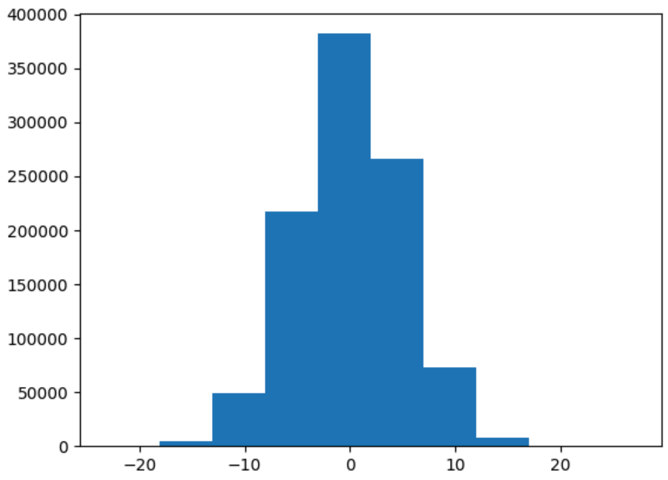
For this I have compiled some data and it is shown below. The axes of the graphs and the order is as followed before.

As the population remains the same, I wont include the population graphs in each. But the sampling distribution will be there because the narrowing and widening give a strong idea. (Things included: Sample sizes, Ratio and Sampling Dist.)

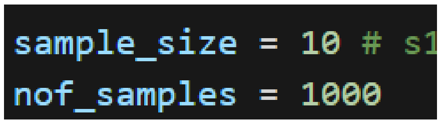


```
pop_size = 1000000
np.random.seed(25)
```

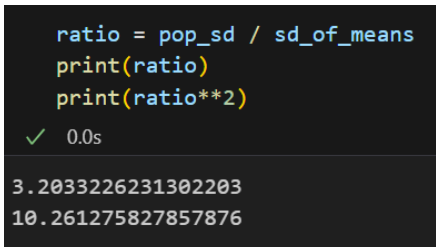




Sample sizes checked:



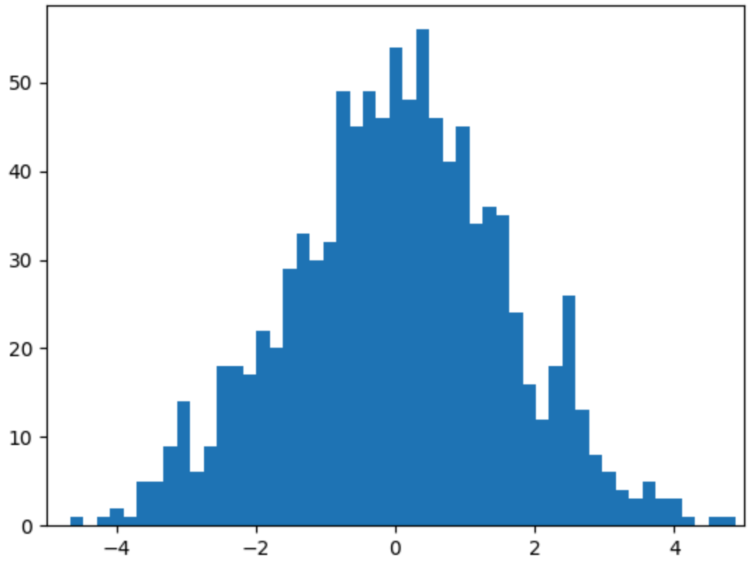
```
sample_size = 10 # s1
nof_samples = 1000
```

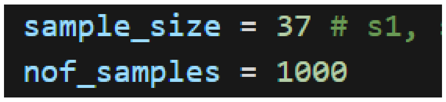


```
ratio = pop_sd / sd_of_means
print(ratio)
print(ratio**2)
```

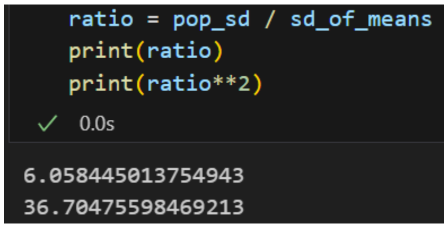
✓ 0.0s

```
3.2033226231302203
10.261275827857876
```





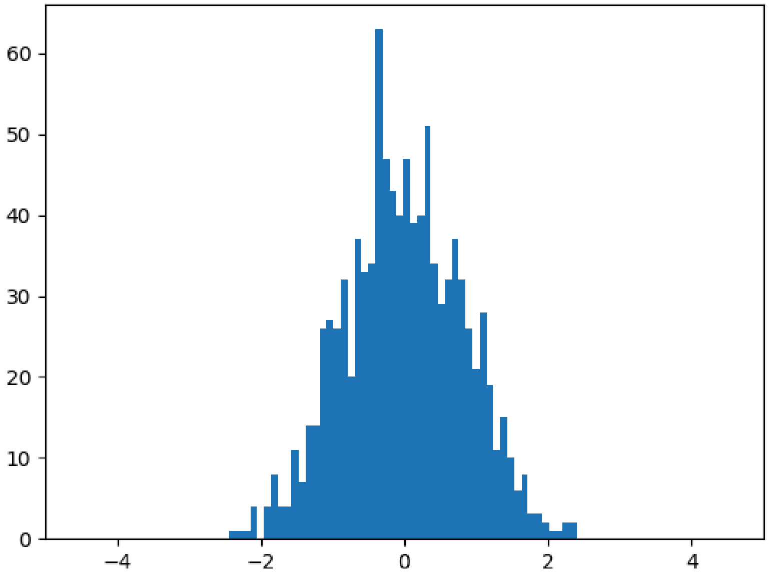
```
sample_size = 37 # s1,
nof_samples = 1000
```

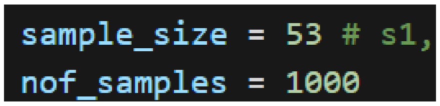


```
ratio = pop_sd / sd_of_means
print(ratio)
print(ratio**2)
```

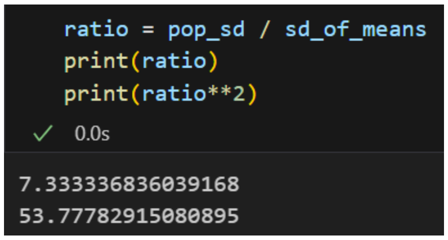
✓ 0.0s

```
6.058445013754943
36.70475598469213
```

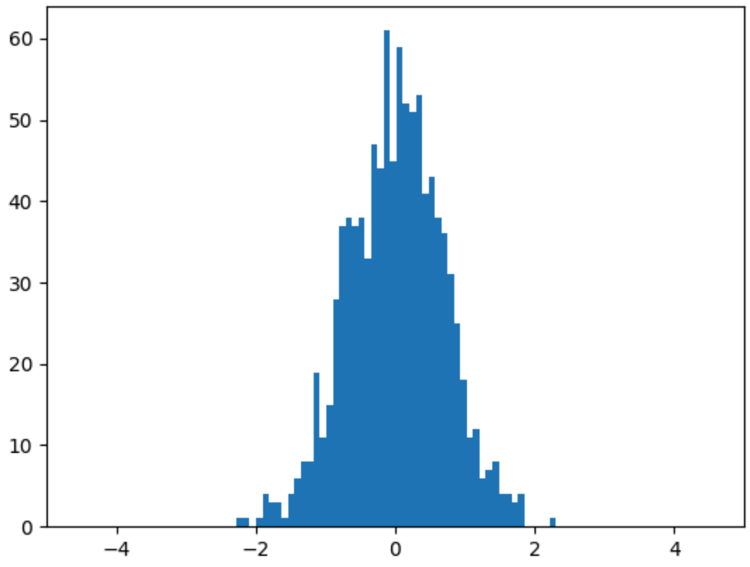


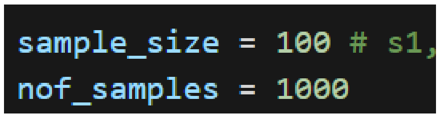


```
sample_size = 53 # s1,
nof_samples = 1000
```

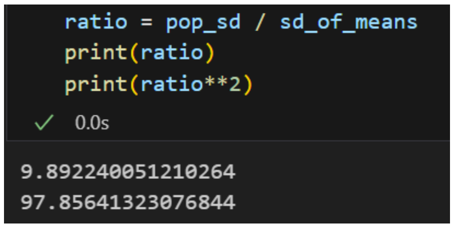


```
    ratio = pop_sd / sd_of_means
    print(ratio)
    print(ratio**2)
 ✓  0.0s

7.333336836039168
53.77782915080895
```

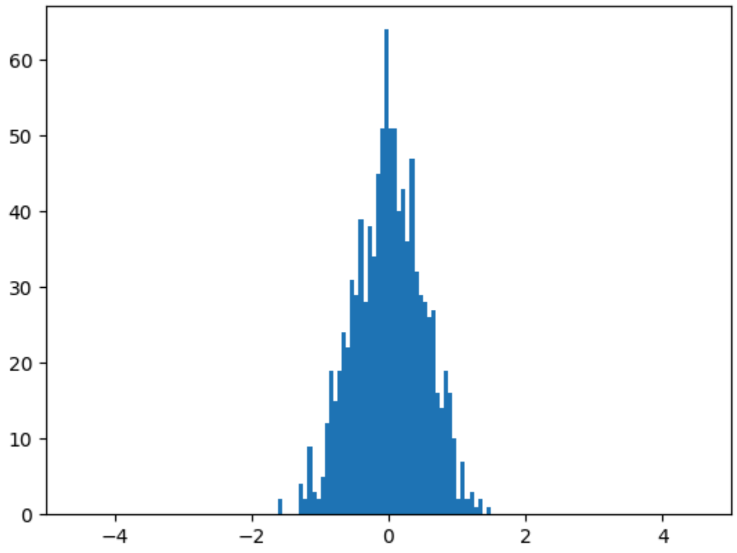


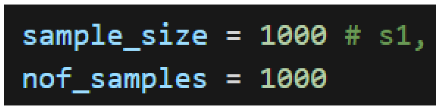


```
sample_size = 100 # s1,
nof_samples = 1000
```

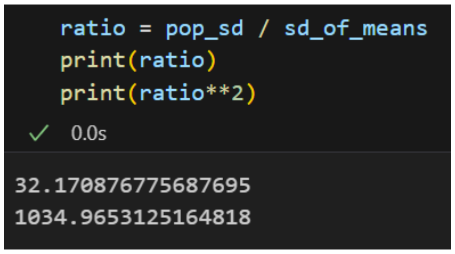


```
    ratio = pop_sd / sd_of_means
    print(ratio)
    print(ratio**2)
 ✓  0.0s

9.892240051210264
97.85641323076844
```





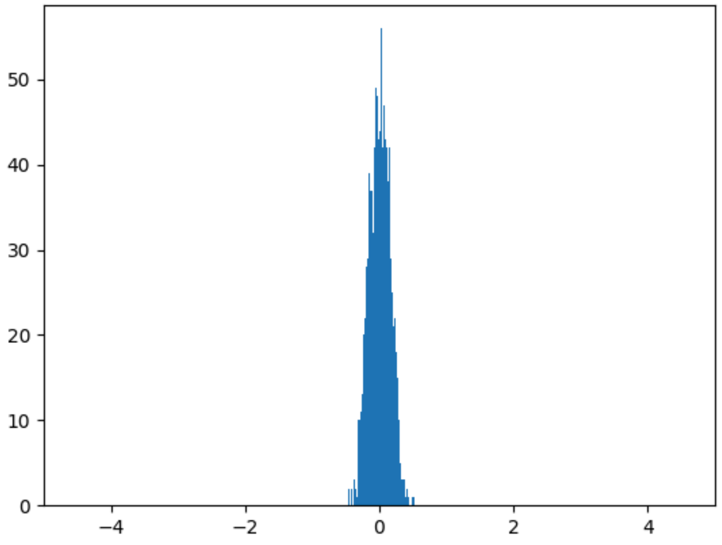
```
sample_size = 1000 # s1,
nof_samples = 1000
```



```
ratio = pop_sd / sd_of_means
print(ratio)
print(ratio**2)
```

✓ 0.0s

```
32.170876775687695
1034.9653125164818
```



Analysis:
From the above prints and plots, one thing that can be inferred is that the ratio value diverges for large values of sample sizes. This is quite easy to explain and it is due to the accumulation of random errors in the sample. But one very interesting thing to note is that the sampling distribution becomes thinner in each run. This also can be explained by the formula from Q.1. The Standard Error or the SD of the sampling distribution depends on the population SD which is a constant and the reciprocal of the sqrt(sample size). So, as the sample size increases, the Standard Error must decrease, hence we get a sharper plot.

This question can be an extension for Q.1. There we found out the proportionality relationship between quantities. But, through this, we also know that, by keeping the SD of pop as a constant, the Std Error is proportional to 1/sqrt(Sample size). So, the entire relationship between the two quantities has been justified.

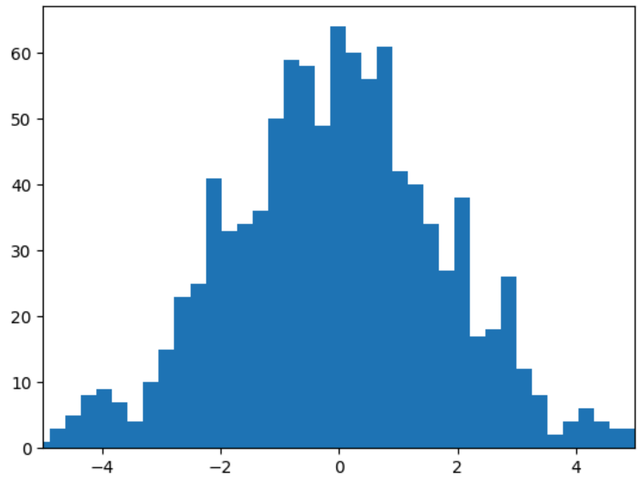
Q.4)

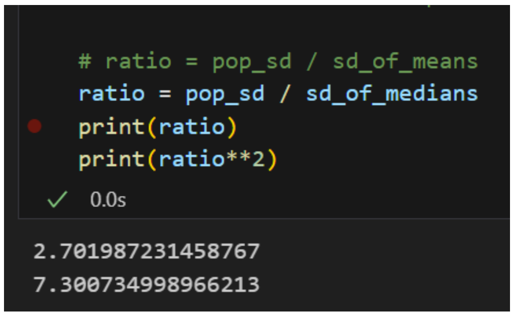
For this question instead of taking the mean of the samples, I have opted to take the median of the samples and make the sampling distribution for the Median.

I have chosen a constant population that is the Gaussian for this. I am not providing the separate pictures of the population curves as they havent changed as i am using the same parameters.

These are the results i got:

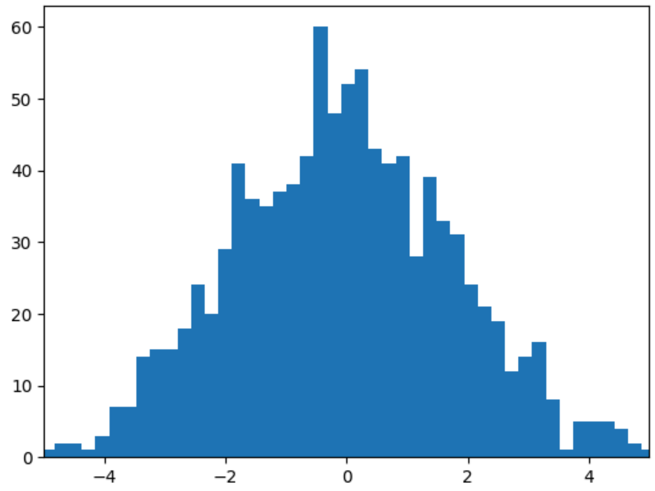
For different runs of the same distribution but with different seed(Gaussian chosen) and a constant sample size of 10:

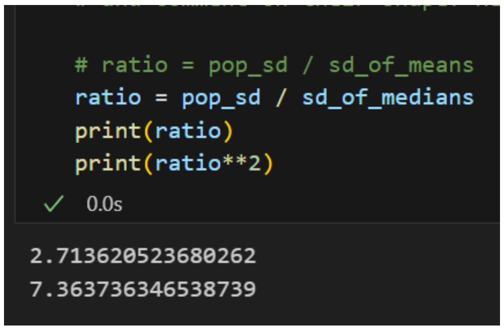




```
# ratio = pop_sd / sd_of_means
ratio = pop_sd / sd_of_medians
print(ratio)
print(ratio**2)
```
✓ 0.0s

```
2.701987231458767
7.300734998966213
```





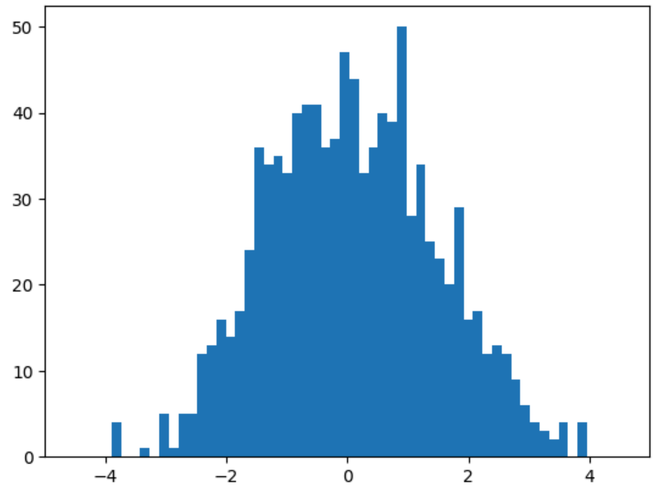
```
# ratio = pop_sd / sd_of_means
ratio = pop_sd / sd_of_medians
print(ratio)
print(ratio**2)
```
✓ 0.0s

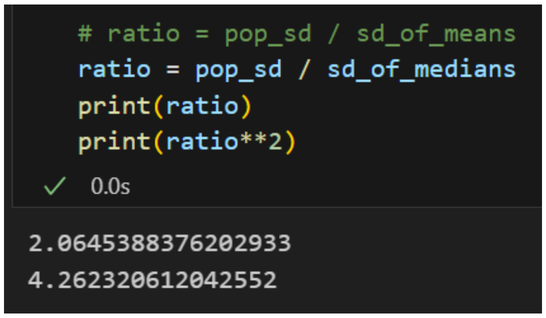
```
2.713620523680262
7.363736346538739
```

Now, pointing out the most obvious thing, the ratio has changed. Now the ratio is close to 7.3 rather than 10. It seems as though an extra factor has jumped into the equation which is kept constant throughout multiple runs. Hence, it seems the constant is not any variable but a constant for the conditions or situations we have taken.

Now choosing different distributions with the sample size constant at 10, we get the sampling distributions and the ratios as:

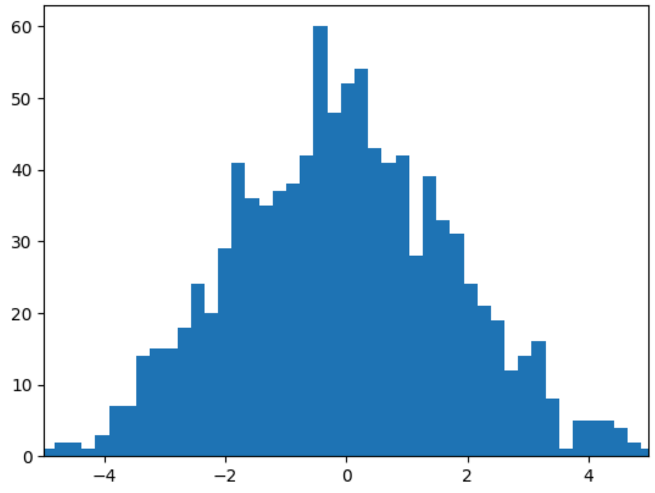
<u>Uniform:</u>

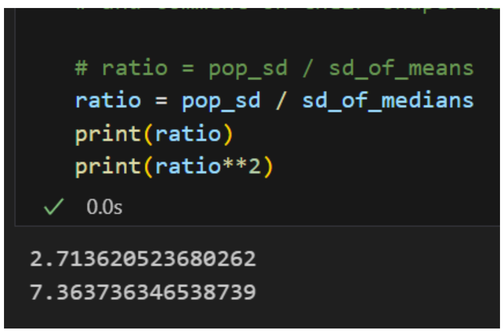




```
    # ratio = pop_sd / sd_of_means
    ratio = pop_sd / sd_of_medians
    print(ratio)
    print(ratio**2)
 ✓  0.0s

2.0645388376202933
4.262320612042552
```

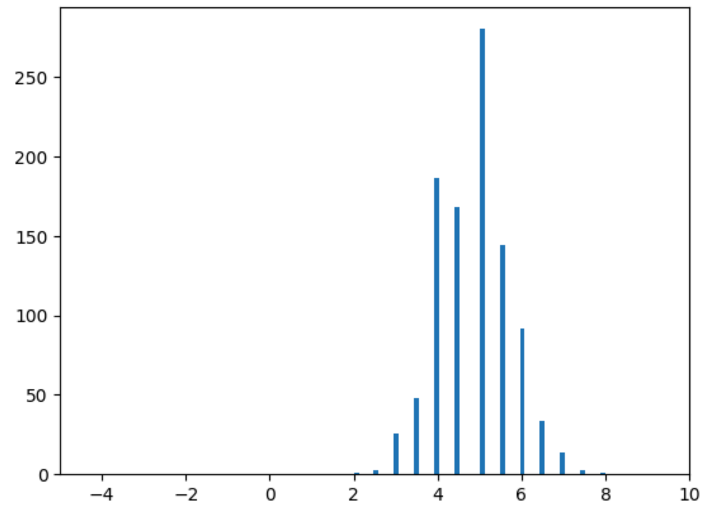
<u>Gaussian:</u>

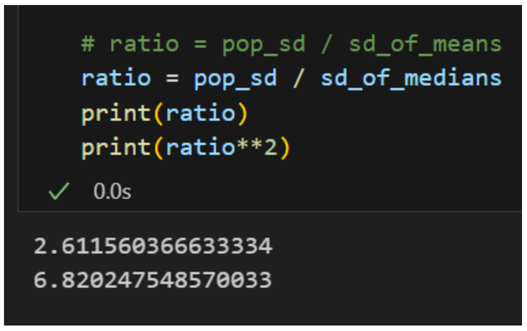




```
    # ratio = pop_sd / sd_of_means
    ratio = pop_sd / sd_of_medians
    print(ratio)
    print(ratio**2)
 ✓  0.0s

2.713620523680262
7.363736346538739
```

Poisson:



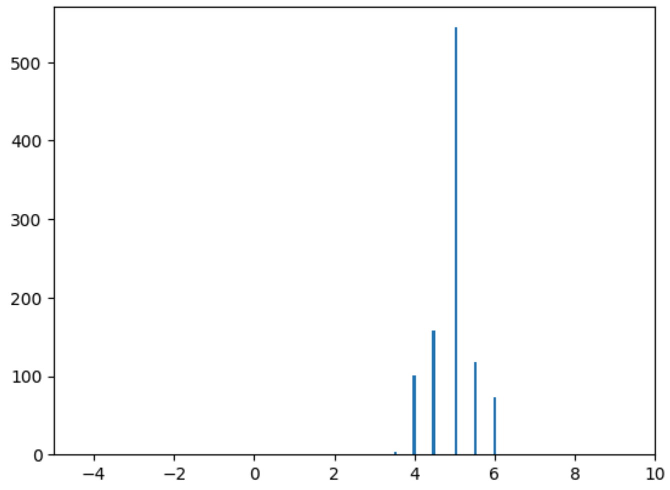


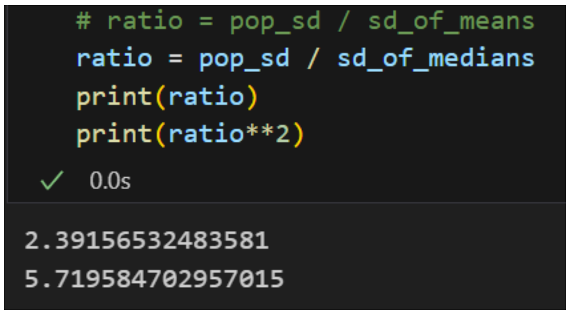
```
# ratio = pop_sd / sd_of_means
ratio = pop_sd / sd_of_medians
print(ratio)
print(ratio**2)
```

✓ 0.0s

```
2.611560366633334
6.820247548570033
```

Binomial:



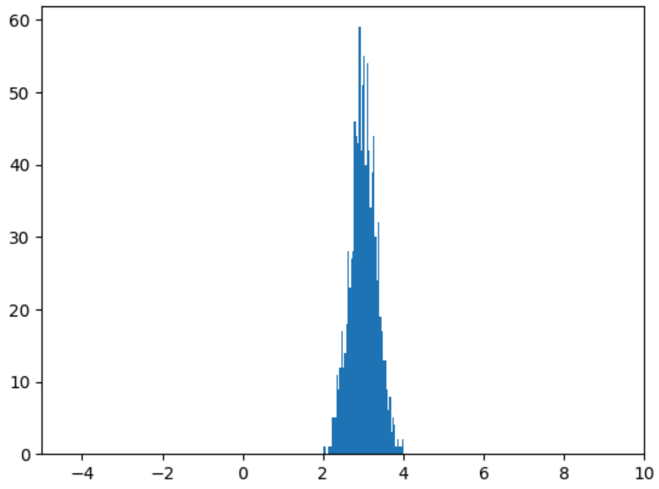


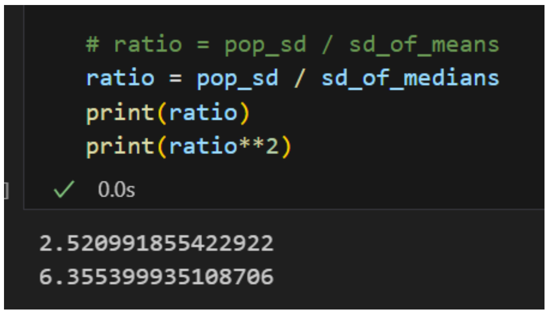
```
# ratio = pop_sd / sd_of_means
ratio = pop_sd / sd_of_medians
print(ratio)
print(ratio**2)
```

✓ 0.0s

```
2.39156532483581
5.719584702957015
```

Triangular:





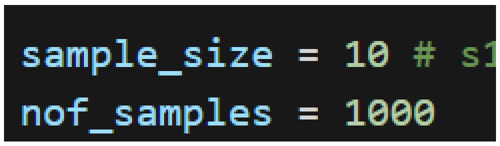
```
# ratio = pop_sd / sd_of_means
ratio = pop_sd / sd_of_medians
print(ratio)
print(ratio**2)
```

✓ 0.0s

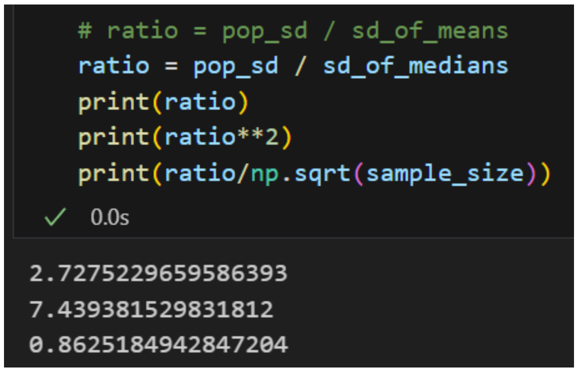
```
2.520991855422922
6.355399935108706
```

Looking at the above data, it seems the well-orderedness we had earlier has been messed up. We know from our earlier venture, that for a particular distribution and sample size, the ratio value will be constant approximately. But from the above exercise, it is clear that for different distributions even under the same value of the sample size, the ratio is simply not constant. It seems to have a dependence on the distribution and the distribution decides the value of the ratio too.

Now varying the sample size keeping a seed and the distribution constant(Gaussian):
Again, I am going to leave out the population distribution plot as it is already given in the previous questions and also it isn't needed for comparison here. Below are the results:



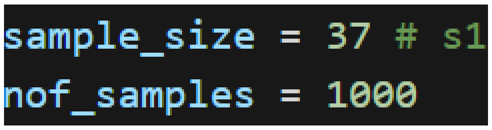
```
sample_size = 10 # s1
nof_samples = 1000
```



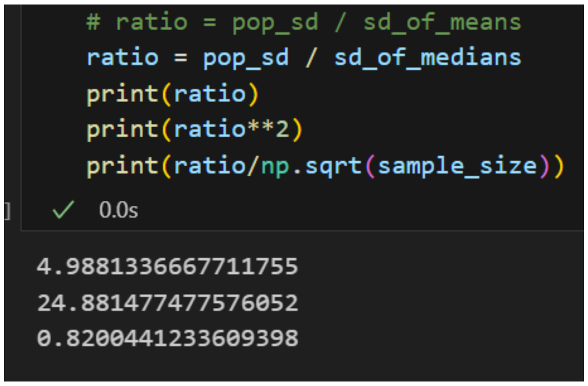
```
# ratio = pop_sd / sd_of_means
ratio = pop_sd / sd_of_medians
print(ratio)
print(ratio**2)
print(ratio/np.sqrt(sample_size))
```

✓ 0.0s

```
2.7275229659586393
7.439381529831812
0.8625184942847204
```



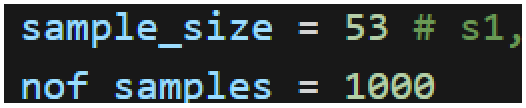
```
sample_size = 37 # s1
nof_samples = 1000
```



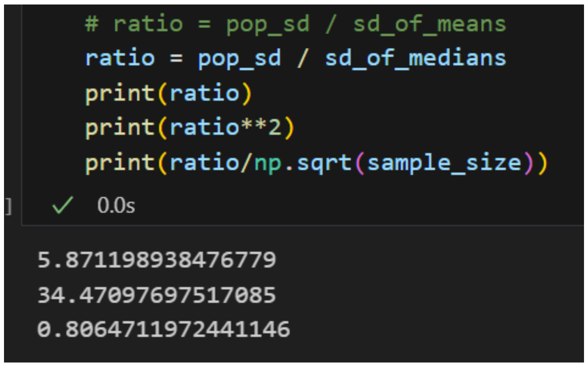
```
# ratio = pop_sd / sd_of_means
ratio = pop_sd / sd_of_medians
print(ratio)
print(ratio**2)
print(ratio/np.sqrt(sample_size))
```

✓ 0.0s

```
4.9881336667711755
24.881477477576052
0.8200441233609398
```



```
sample_size = 53 # s1,
nof_samples = 1000
```



```
# ratio = pop_sd / sd_of_means
ratio = pop_sd / sd_of_medians
print(ratio)
print(ratio**2)
print(ratio/np.sqrt(sample_size))
```

✓ 0.0s

```
5.871198938476779
34.47097697517085
0.8064711972441146
```

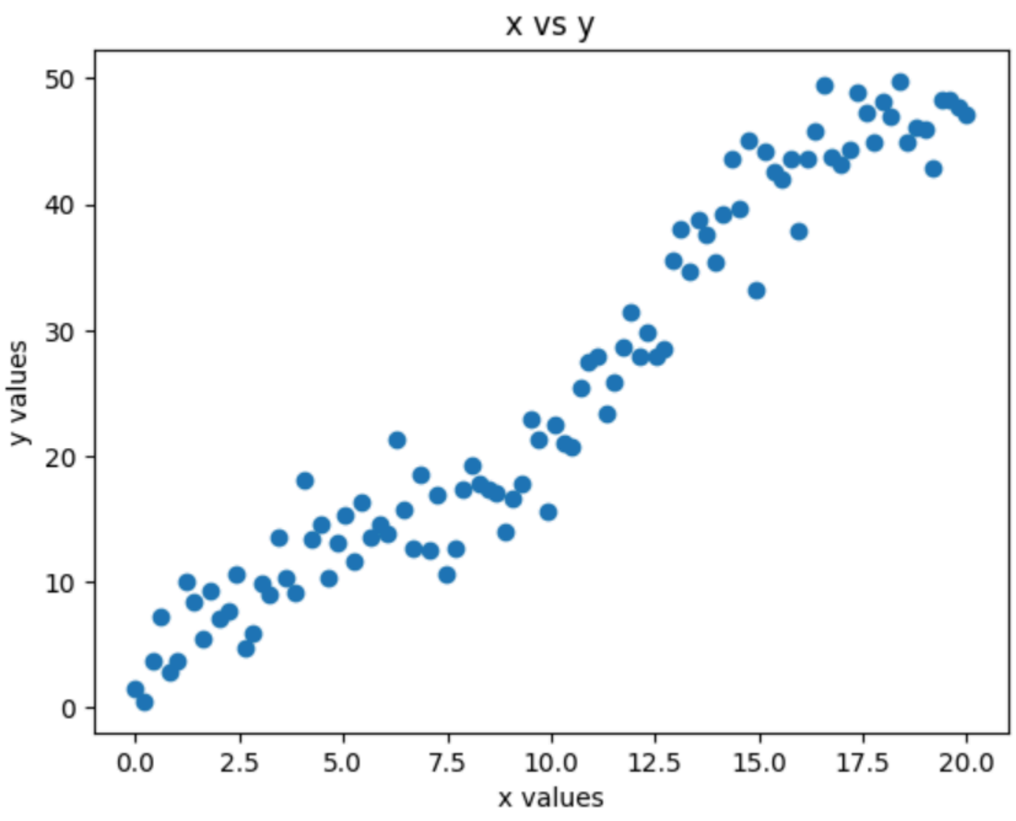
<u>Analysis:</u>
Earlier i only checked the constant behaviour of the ratio. But now i am checking the ratio/sqrt(sample size). This is done because i want to check whether the relation with sqrt(sample size) is still present or not. After checking that, I found out that the relation still persists because the ratio/sqrt gives a constant value close to 0.8. This means its just some extra constant that messes up the values.

According to all the above 3 exercises, I can safely conclude that, the SD of the sampling distribution(Medians) is inversely proportional to the sqrt(sample size) and directly proportional to the SD of the population and the only change from the earlier known equation is that, here, there is an extra constant term that seems to depend on the distribution chosen. This is because the constant value changes as per the distribution chosen. So in a general sense, I can define that the value is,

$$SD_{\text{sampling dist of median}} \approx \frac{SD_{\text{pop}}}{c\sqrt{n}}$$

This general solution makes sense. Here c is some constant that depends on the distribution of the population. Different the distribution, different the value. All the other quantities remain the same.

2. **SECTION 2:**
Qa:
I read the .csv file through the pandas library using the read_csv function. The made pandas arrays of the x and y values are used to make a scatter plot and the result is as follows.



Only using simple observation, the x and y values seem to have some kind of a relationship and it is quite obvious. The graph shape is linear and it seems as though x and y are almost proportional to each other as the constant term also seems to be close to 0. Simply put, the data features appear to have strong correlation with each other. This is my analysis, simply based upon visualization.

My idea would be to apply SLR to it as the plot looks quite linear. Though a small kink is seen in the middle, SLR would still perform a good job analysing it without increasing the complexity much and going into feature engineering.

Q.b:

Pearson's Correlation Coefficent is a method to check whether the features which are plotted out in the regression pattern actually are related to each other or is the regression line simply not able to get and pattern out of it, or maybe the line's prediction skills is just a matter of coincidence but the values aren't really correlated.

The formula is as given below,

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

I have simply done some algebra on the pandas arrays to get the data needed for the formula to compute the Pearson's coefficient.

```
xi_xbar_yi_ybar = 0
xi_xbar_sq = 0
yi_ybar_sq = 0

xbar = 0
ybar = 0

for i in range(0, 100):
    xbar += x_values[i]
    ybar += y_values[i]

xbar /= 100
ybar /= 100

for i in range(0, 100):
    xi_xbar_yi_ybar += (x_values[i] - xbar) * (y_values[i] - ybar)
    xi_xbar_sq += (x_values[i] - xbar)**2
    yi_ybar_sq += (y_values[i] - ybar)**2

pearson_corr_coeff = xi_xbar_yi_ybar / np.sqrt(xi_xbar_sq *
yi_ybar_sq)
```

The final values that i got was the following

Pearson's Correlation Coefficient for the above problem,

r = 0.9674196412653445

This proves that there is a strong correlation between the x and y values given in the .csv file and that my intuition from earlier looking at the plot was correct. A SLR would easily get out the pattern we want to see.

This is just a property that says that the model will work good on the data given.

<u>Q.c:</u>

This question has little explanation involved. But, one explanation I can provide is how I did the data construction.

As already given in the question, reduction of the SD of the y values is quite easy and only involves the multiplication of the ratio of the desired SD and the current SD. For this, I calculated the SD of the original y values and stored it in the variable SD_y. One important thing here is that I applied Bessel's correction as the data given is some sample only and not the population as a whole. Reduction can be done using the ratio multiplication itself because we are reducing the spread hence we must reduce the noise involved hence the ratio will do.

The increasing SD cases work by inflating the SD not by the ratio, but rather by inducing a noise, as the former wouldn't give the randomness associated with a higher value of SD. This is what i understood from reading about this more.

Some cases of what I did,

```
#SD reduced to 5
y_values_red5 = []

y_values_red5 = y_values * (5 / SD_y)
```

```
#SD increased to 20
mean_20 = 0
sigma_noise_20 = np.sqrt(20**2 - SD_y**2)

```

```
y_values_inc20 = []

y_values_inc20 = y_values + np.random.normal(mean_20,
sigma_noise_20, size = 100)
```

I didnt use any seed here. I didnt have any reason not to, but I preferred to keep no seed. I just wanted to keep a mention of that.

Nearby all these calculations, I have also put a print statement to verify the calculated SD values of the new data. It used the normal SD function but with the argument as ddof=1 which puts the Bessel correction into place as this is a sample and not a population.

For the seed I got at some time, I received this,

```
Reduced to 5: 5.000000000000002
Reduced to 10: 10.000000000000004
Original: 15.061425869440946
Increased to 20: 20.104206078564435
Increased to 25: 25.41653127046244
```

These values are close to our wanted values, hence the change did work.

<u>Train and Test Data:</u>

Making the train and test data was the place I had spent some time thinking. It was evident the data couldn't simply be used, the first 80 as train and last 20 as test. This was because the values were in increasing order. So a bias towards the lower values would be shown by the model which isn't great for the model's functioning.

Unlike excel where the shuffling was easier, in python, as i was using a pandas array, normal techniques produced an array with shuffled entries, but the indices were also shuffled, hence the i-th element still corresponded to the original array. This was fixed by introducing `.reset_index(drop=True)`. This, I had to take from the internet and it did take some digging. But this fixed that issue.

After getting the new shuffled lists, I applied the first 80 entries as the train data and the last 20 as the test data.

This was the code snippet for this part that i wrote,

```
#Choosing indices for train/test data was not easy, as the data
provided was not random, hence simply choosing the last 20 would not
work as all values are close together. Hence, I had to shuffle the
values. That is what is done below.
np.random.seed(25)

shuffle_indexes = np.random.permutation(100)

x_values_shuff = x_values[shuffle_indexes].reset_index(drop=True)

y_values_red5_shuff =
y_values_red5[shuffle_indexes].reset_index(drop=True)
y_values_red10_shuff =
y_values_red10[shuffle_indexes].reset_index(drop=True)
y_values_orig_shuff =
y_values_orig[shuffle_indexes].reset_index(drop=True)
y_values_inc20_shuff =
y_values_inc20[shuffle_indexes].reset_index(drop=True)
y_values_inc25_shuff =
y_values_inc25[shuffle_indexes].reset_index(drop=True)
```

Here, I did use a seed. I had put this there initially to check the values in the x and y lists and whether corresponding entries were the same. I checked it and it was the same but even after that, I had still kept it in.

Model Training and Calculation of Metrics asked for:
Now i had to train the model using the train data and test it using the test data.

I used the statsmodel.api library to apply the Linear Regression onto my model. I used this library which was said by sir rather than the other one as this library has many of the functions that sir asked for built in.

The below is the code snippet I wrote for the original code. It is a bit lengthy hence I am only providing the snippet for the code used to model the original x and y data. (Train data). It also calculates the various metric that was asked for in the question specifically.

```python
x_train_orig = x_values_shuff[0:80]
x_train_orig = sm.add_constant(x_train_orig)

y_train_orig = y_values_orig_shuff[0:80]

model = sm.OLS(y_train_orig, x_train_orig).fit()

y_train_pred_orig = model.predict(x_train_orig)

r2_train_orig = model.rsquared
rmse_train_orig = np.sqrt(model.mse_resid)
fval_train_orig = model.fvalue
fpval_train_orig = model.f_pvalue

print("--------------------------------------------------")
print("TRAIN DATA STATS:")
print(f"R2 on original train_data: {r2_train_orig}")
print(f"RMSE on original train_data: {rmse_train_orig}")
print(f"F statistic on original train_data: {fval_train_orig}")
print(f"F statistic p-value on original train_data:
{fpval_train_orig}")

#The below coefficients printing i took from the web(because i
wanted the CI also included). I did learn it though, i didnt blindly
cut copy paste it.....
print("\nCoefficients:")
for name in model.params.index:
    coef = model.params[name]
    pval = model.pvalues[name]
    ci = model.conf_int().loc[name]
    print(f"{name}: Value={coef}, P-value={pval}, CI=({ci[0]},
{ci[1]})")

x_test_orig = x_values_shuff[80:100]
x_test_orig = sm.add_constant(x_test_orig)
y_test_orig = y_values_orig_shuff[80:100]

y_test_pred_orig = model.predict(x_test_orig)
r2_test_orig = r2_score(y_test_orig, y_test_pred_orig)
```

```
rmse_test_orig = np.sqrt(mean_squared_error(y_test_orig,
y_test_pred_orig))

train_errors_orig = y_train_orig - y_train_pred_orig
test_errors_orig = y_test_orig - y_test_pred_orig

CI_0_width_orig = 0.0
CI_1_width_orig = 0.0

print("\n--------------------------------------------------")
print("TEST DATA STATS:")
print(f"R2 on original test_data: {r2_test_orig}")
print(f"RMSE on original test_data: {rmse_test_orig}")
print("\nRest of the statistics will be identical to the train data
as those are intinsic to the data used to train the model\n")
print(f"F statistic on original train_data: {fval_train_orig}")
print(f"F statistic p-value on original train_data:
{fpval_train_orig}")
print("\nCoefficients:")
for name in model.params.index:
    coef = model.params[name]
    pval = model.pvalues[name]
    ci = model.conf_int().loc[name]
    if name == 0:
        CI_0_width_orig = ci[1] - ci[0]
    else:
        CI_1_width_orig = ci[1] - ci[0]
    print(f"{name}: Value={coef}, P-value={pval}, CI=({ci[0]},
{ci[1]})")
print("--------------------------------------------------")

pval_0_orig = model.pvalues[0]
pval_1_orig = model.pvalues[1]
```

The functions that were used to calculate various metrics and Confidence Intervals(CIs) was taken from the online functions lists of the statmodels.api library. This was the case for the Train Data. Whereas for the Test Data, I used the sklearn library. I used the same method taught in class and that was in the

.ipynb file supplied but instead of the MSE, i calculated RMSE by taking the sqrt of the MSE.

The Train Data produced all the metrics asked for but the test data could only produce the R^2 and RMSE values. This was because the rest of the metrics were values intrinsic to each and every model. It was not possible to make those metrics for the test data, but I have provided the same metrics found in the Train case for the Test case.

The output for the original data:

```
--------------------------------------------------
TRAIN DATA STATS:
R2 on original train_data: 0.9349264426566539
RMSE on original train_data: 3.841057551874891
F statistic on original train_data: 1120.6435533015413
F statistic p-value on original train_data: 4.914874705727151e-48

Coefficients:
const: Value=0.5622642044778441, P-value=0.5230459481151172,
CI=(-1.1825433436417077, 2.3070717525973956)
x: Value=2.496386042404972, P-value=4.914874705727205e-48,
CI=(2.3479238405877005, 2.644848244222244)

--------------------------------------------------
TEST DATA STATS:
R2 on original test_data: 0.9362638860774684
RMSE on original test_data: 3.8115630761169683

Rest of the statistics will be identical to the train data as those
are intinsic to the data used to train the model

F statistic on original train_data: 1120.6435533015413
F statistic p-value on original train_data: 4.914874705727151e-48

Coefficients:
const: Value=0.5622642044778441, P-value=0.5230459481151172,
CI=(-1.1825433436417077, 2.3070717525973956)
x: Value=2.496386042404972, P-value=4.914874705727205e-48,
CI=(2.3479238405877005, 2.644848244222244)
--------------------------------------------------
```

The output for the reduced SD5 data:

```
--------------------------------------------------
TRAIN DATA STATS:
R2 on SD 5(reduced) train_data: 0.9349264426566539
RMSE on SD 5(reduced) train_data: 1.2751307828258973
F statistic on SD 5(reduced) train_data: 1120.643553301542
F statistic p-value on SD 5(reduced) train_data:
4.914874705727068e-48

Coefficients:
const: Value=0.18665703013506021, P-value=0.5230459481151184,
CI=(-0.3925735032966049, 0.7658875635667253)
x: Value=0.8287349630920549, P-value=4.914874705727066e-48,
CI=(0.7794493897657951, 0.8780205364183146)

--------------------------------------------------
TEST DATA STATS:
R2 on SD 5(reduced) test_data: 0.9362638860774684
RMSE on SD 5(reduced) test_data: 1.2653393872390537

Rest of the statistics will be identical to the train data as those
are intinsic to the data used to train the model

F statistic on SD 5(reduced) train_data: 1120.643553301542
F statistic p-value on SD 5(reduced) train_data:
4.914874705727068e-48

Coefficients:
const: Value=0.18665703013506021, P-value=0.5230459481151184,
CI=(-0.3925735032966049, 0.7658875635667253)
x: Value=0.8287349630920549, P-value=4.914874705727066e-48,
CI=(0.7794493897657951, 0.8780205364183146)
--------------------------------------------------
```

The output for the reduced SD10 data:

```
--------------------------------------------------
TRAIN DATA STATS:
R2 on SD 10(reduced) train_data: 0.9349264426566539
RMSE on SD 10(reduced) train_data: 2.5502615656517946
F statistic on SD 10(reduced) train_data: 1120.643553301542
F statistic p-value on SD 10(reduced) train_data:
4.914874705727068e-48

Coefficients:
const: Value=0.37331406027012043, P-value=0.5230459481151184,
CI=(-0.7851470065932098, 1.5317751271334505)
x: Value=1.6574699261841097, P-value=4.914874705727066e-48,
CI=(1.5588987795315903, 1.7560410728366291)

--------------------------------------------------
TEST DATA STATS:
R2 on SD 10(reduced) test_data: 0.9362638860774684
RMSE on SD 10(reduced) test_data: 2.5306787744781074

Rest of the statistics will be identical to the train data as those
are intinsic to the data used to train the model

F statistic on SD 10(reduced) train_data: 1120.643553301542
F statistic p-value on SD 10(reduced) train_data:
4.914874705727068e-48

Coefficients:
const: Value=0.37331406027012043, P-value=0.5230459481151184,
CI=(-0.7851470065932098, 1.5317751271334505)
x: Value=1.6574699261841097, P-value=4.914874705727066e-48,
CI=(1.5588987795315903, 1.7560410728366291)
--------------------------------------------------
```

The output for the increased SD20 data:

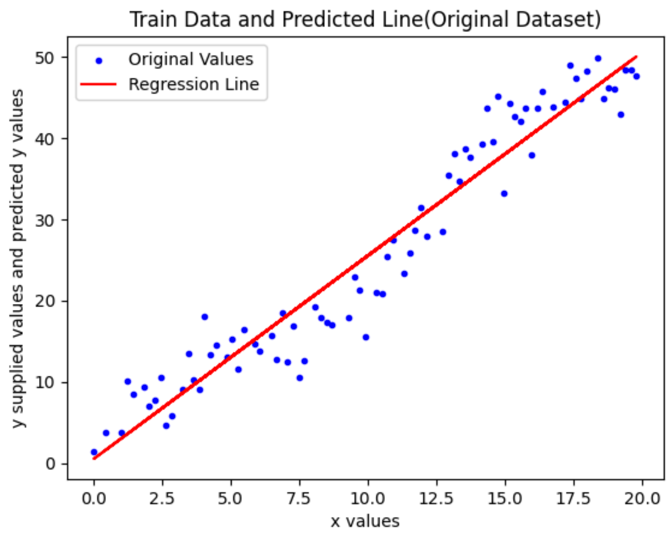
```
--------------------------------------------------
TRAIN DATA STATS:
R2 on SD 20(Increased) train_data: 0.5949676788609988
RMSE on SD 20(Increased) train_data: 13.66901503460519
F statistic on SD 20(Increased) train_data: 114.57722391303068
F statistic p-value on SD 20(Increased) train_data:
5.69809242989339e-17

Coefficients:
const: Value=-2.4934822675567783, P-value=0.42643731570455823,
CI=(-8.702657809875129, 3.715693274761573)
x: Value=2.8406226899730376, P-value=5.698092429893316e-17,
CI=(2.31229630035143, 3.3689490795946453)

--------------------------------------------------
TEST DATA STATS:
R2 on SD 20(Increased) test_data: 0.5503823149465026
RMSE on SD 20(Increased) test_data: 12.339299435540275

Rest of the statistics will be identical to the train data as those
are intinsic to the data used to train the model

F statistic on SD 20(Increased) train_data: 114.57722391303068
F statistic p-value on SD 20(Increased) train_data:
5.69809242989339e-17

Coefficients:
const: Value=-2.4934822675567783, P-value=0.42643731570455823,
CI=(-8.702657809875129, 3.715693274761573)
x: Value=2.8406226899730376, P-value=5.698092429893316e-17,
CI=(2.31229630035143, 3.3689490795946453)
--------------------------------------------------
```

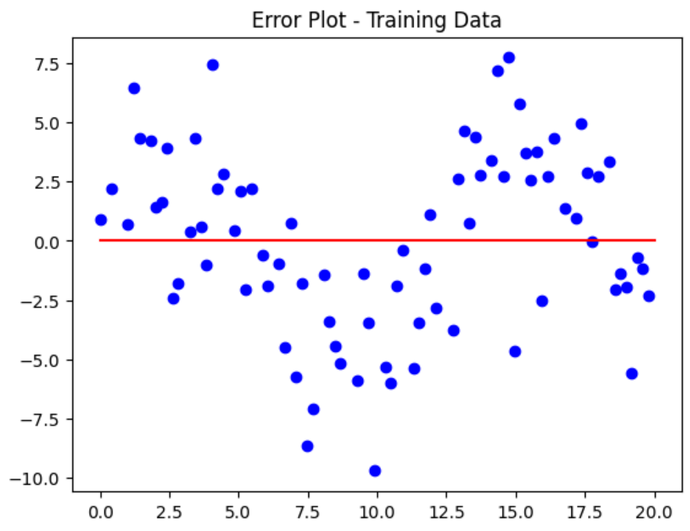
The output for the increased SD25 data:

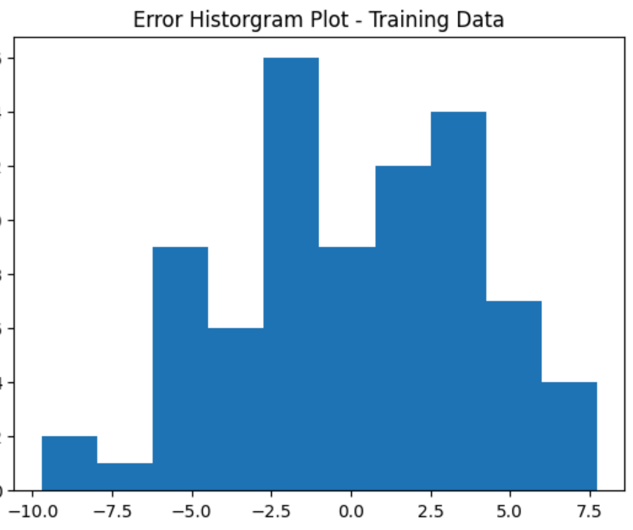
```
--------------------------------------------------
TRAIN DATA STATS:
R2 on SD 25(Increased) train_data: 0.3846830902648769
RMSE on SD 25(Increased) train_data: 19.934992668328604
F statistic on SD 25(Increased) train_data: 48.76394678244229
F statistic p-value on SD 25(Increased) train_data:
8.482180041088439e-10

Coefficients:
const: Value=-0.41696850363217886, P-value=0.9271950690277082,
CI=(-9.472476058427755, 8.638539051163399)
x: Value=2.702667924109346, P-value=8.482180041088353e-10,
CI=(1.9321528082120079, 3.473183040006684)

--------------------------------------------------
TEST DATA STATS:
R2 on SD 25(Increased) test_data: 0.46153088561981714
RMSE on SD 25(Increased) test_data: 21.157277834654025

Rest of the statistics will be identical to the train data as those
are intinsic to the data used to train the model

F statistic on SD 25(Increased) train_data: 48.76394678244229
F statistic p-value on SD 25(Increased) train_data:
8.482180041088439e-10

Coefficients:
const: Value=-0.41696850363217886, P-value=0.9271950690277082,
CI=(-9.472476058427755, 8.638539051163399)
x: Value=2.702667924109346, P-value=8.482180041088353e-10,
CI=(1.9321528082120079, 3.473183040006684)
--------------------------------------------------
```

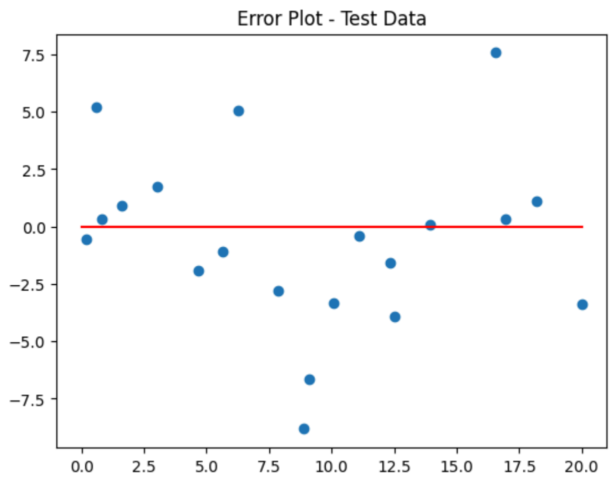
Extra Plots just for representation:

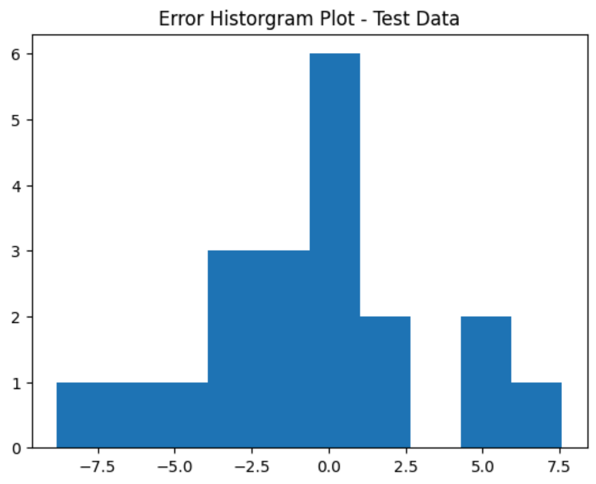
Original:



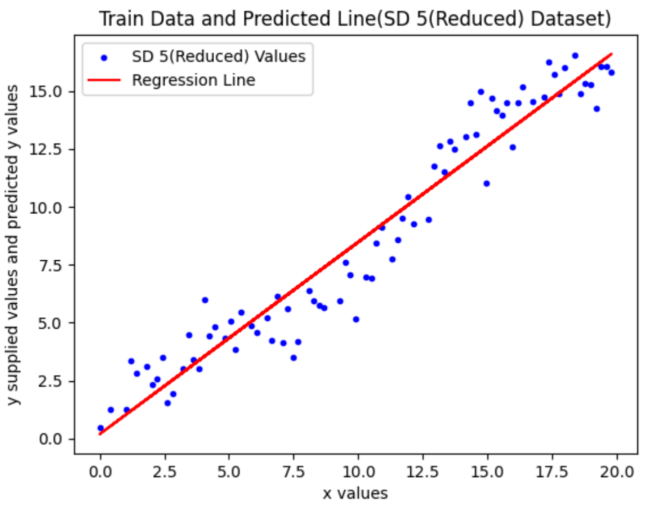


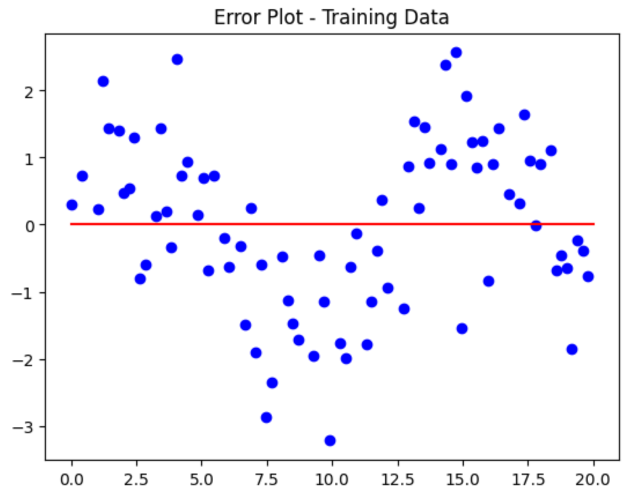


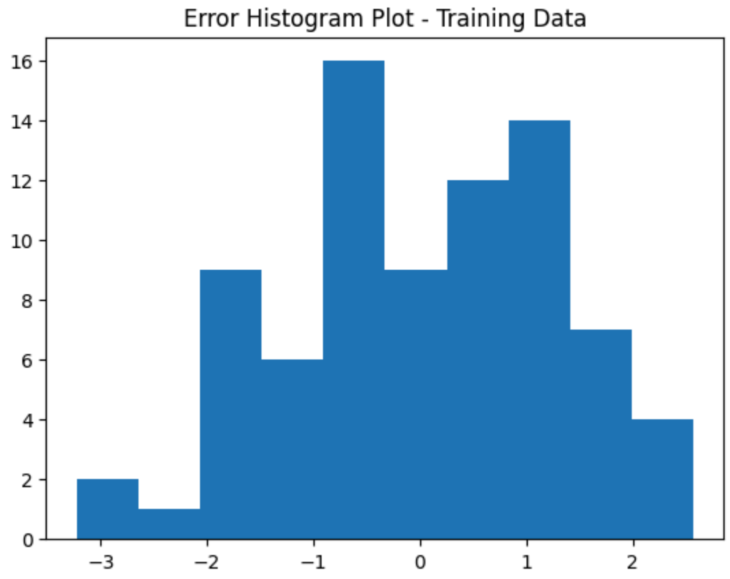


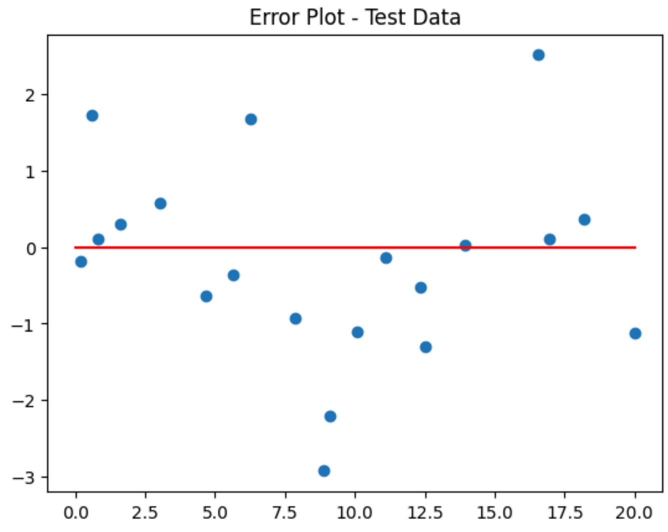


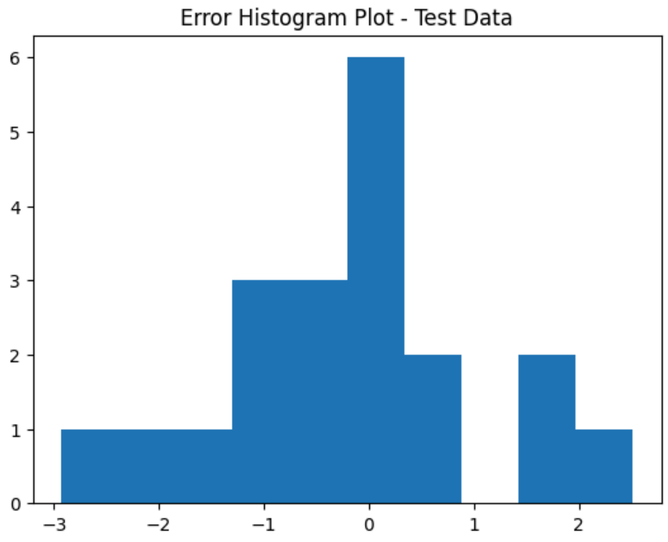
SD Reduced to 5:



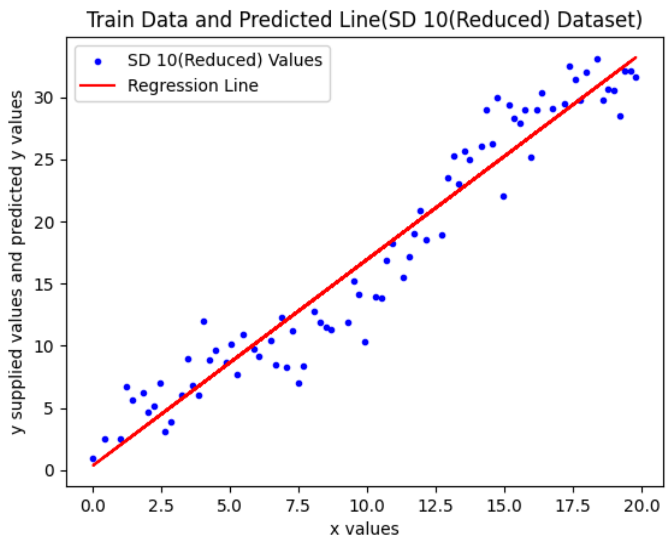


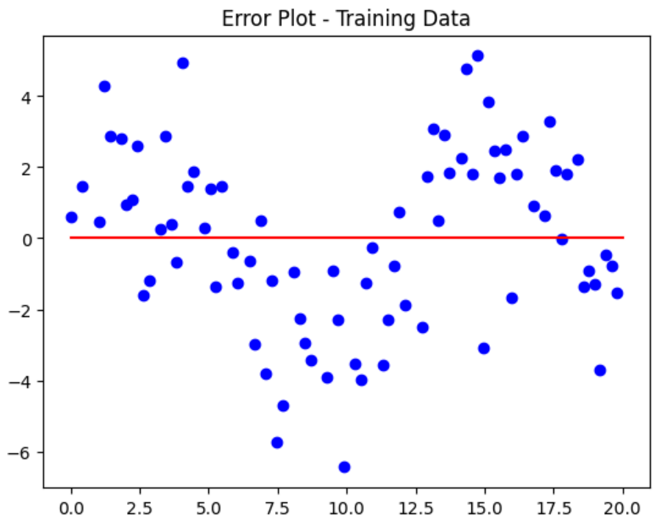


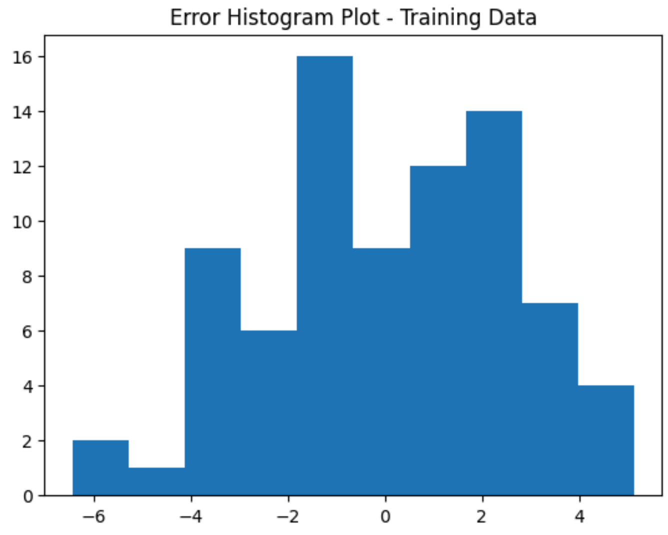


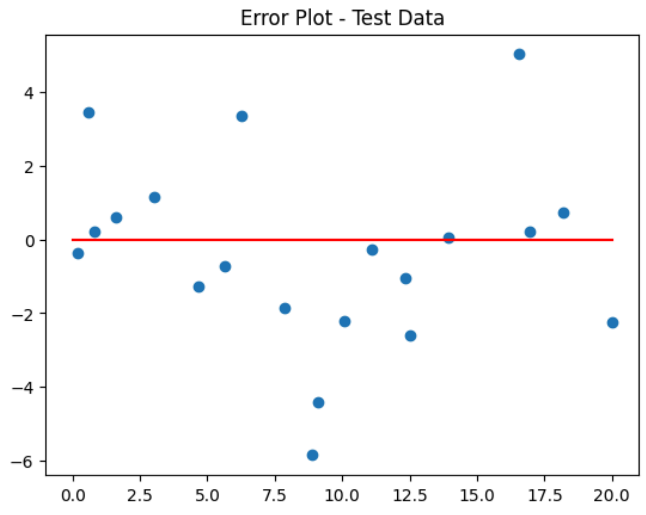


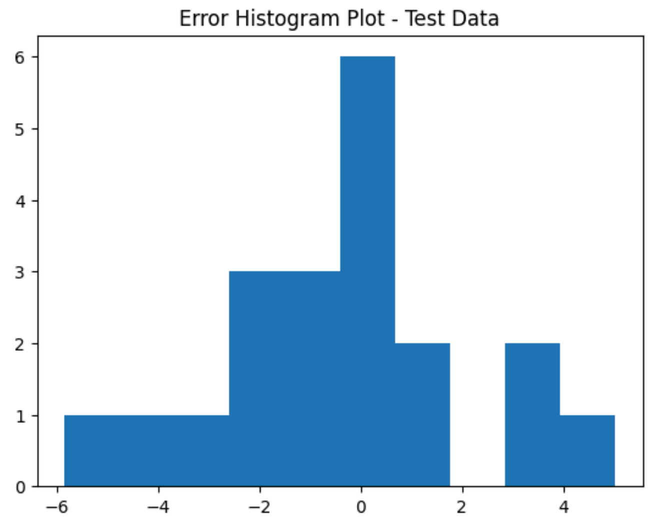
SD Reduced to 10:



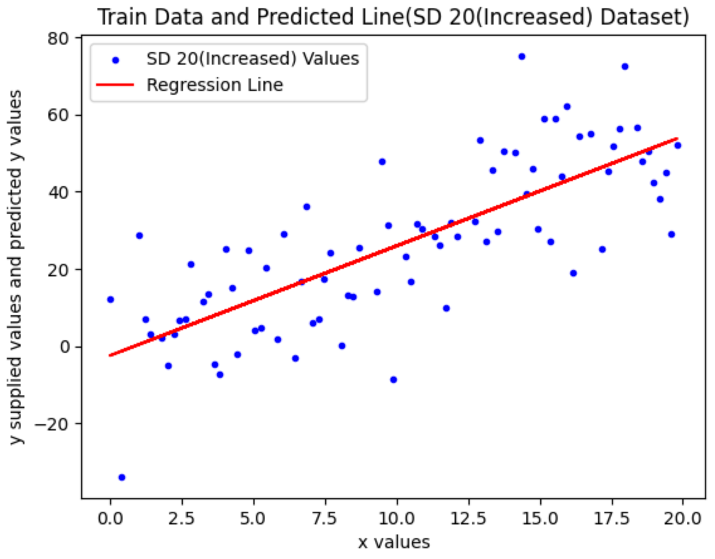


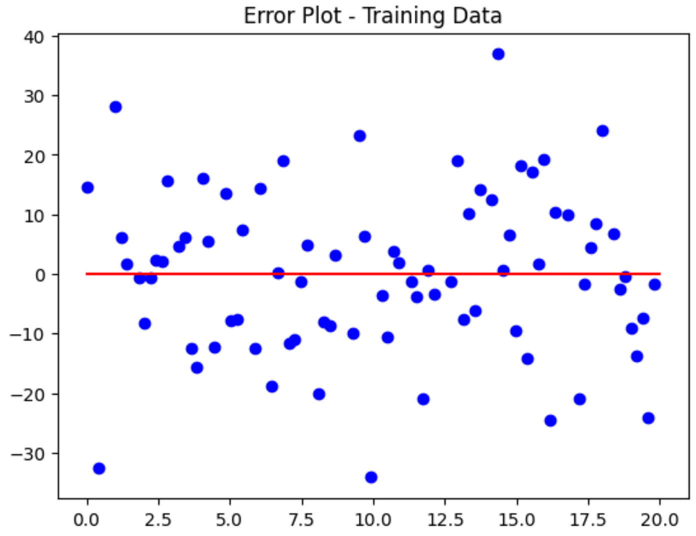


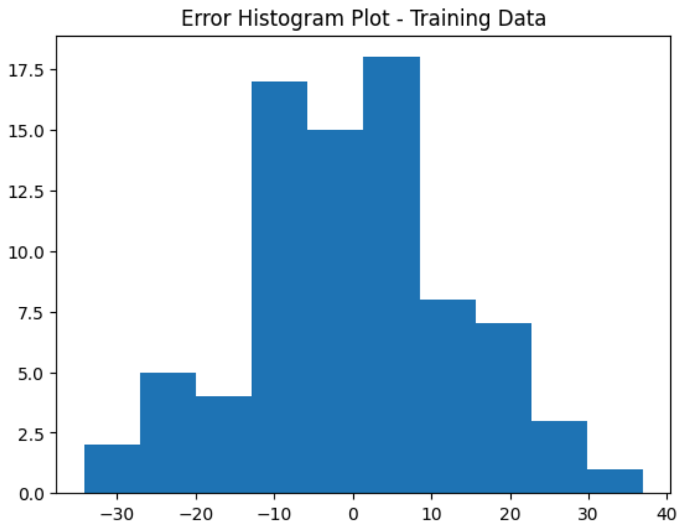


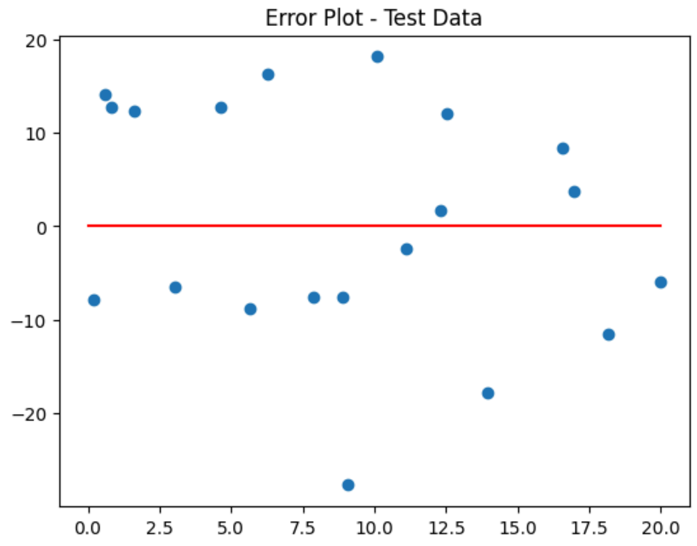


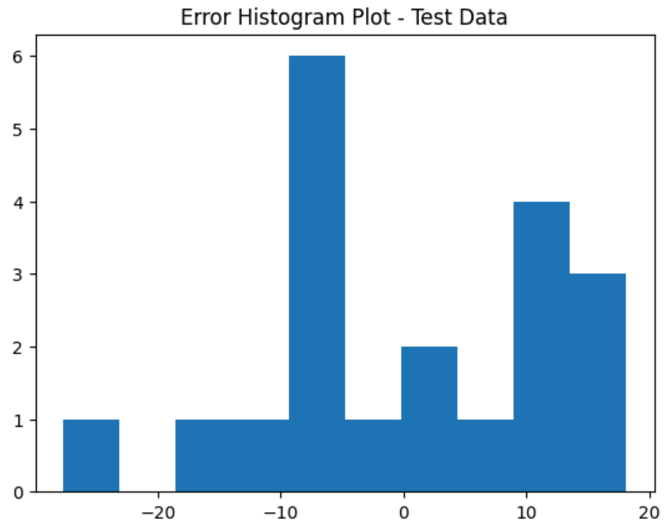
SD Increased to 20:



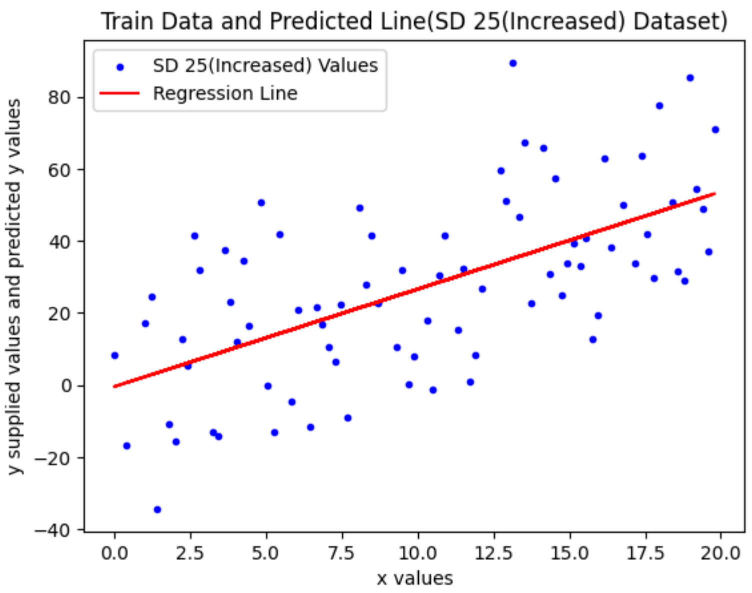


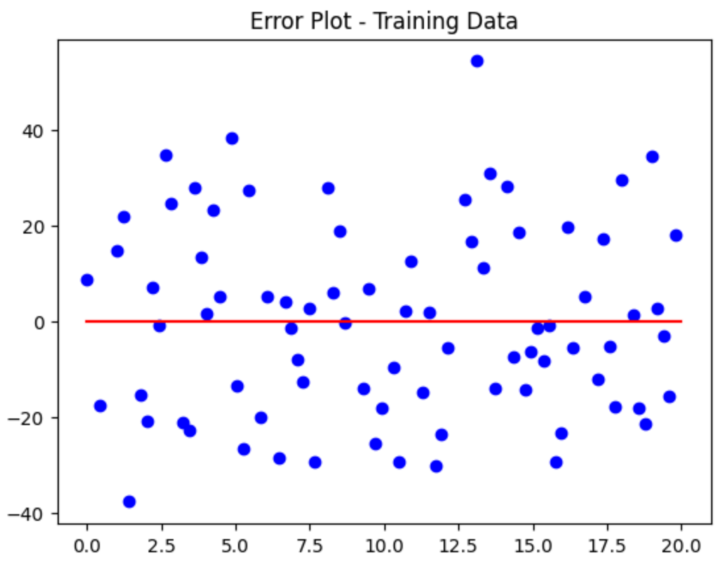


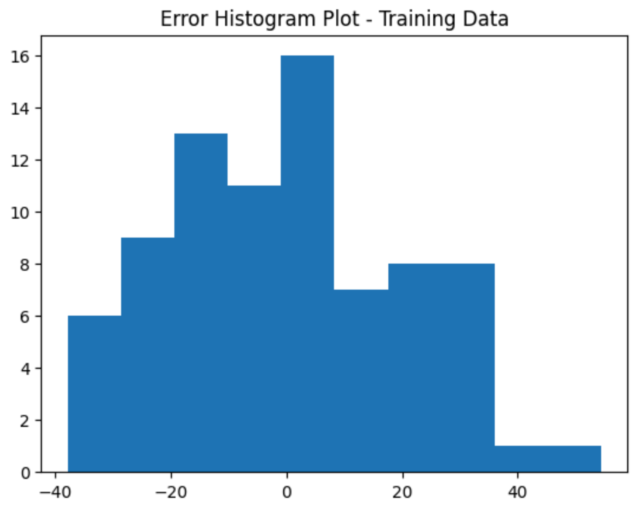


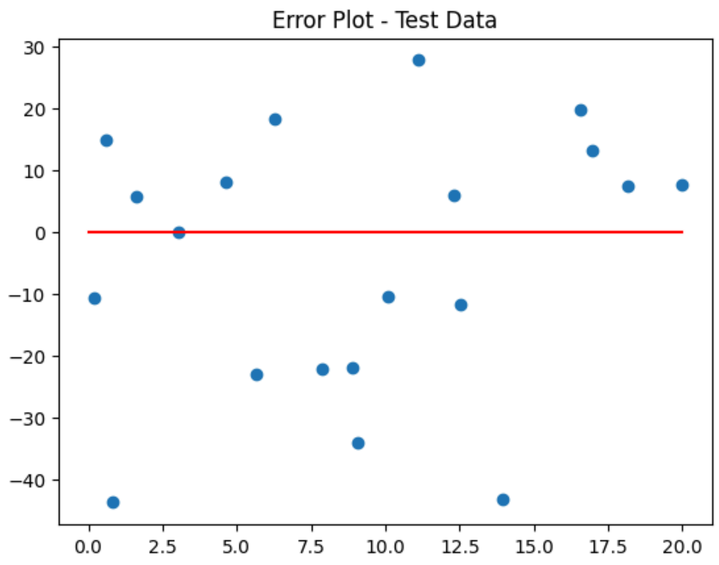


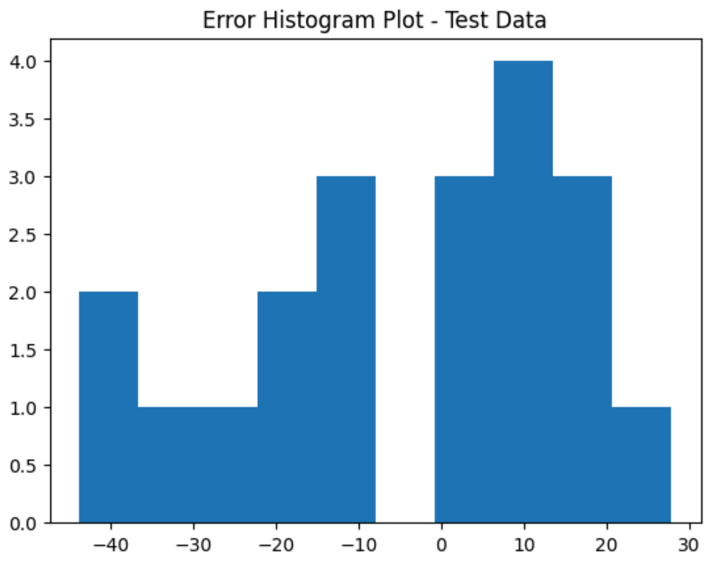
SD Increased to 25:









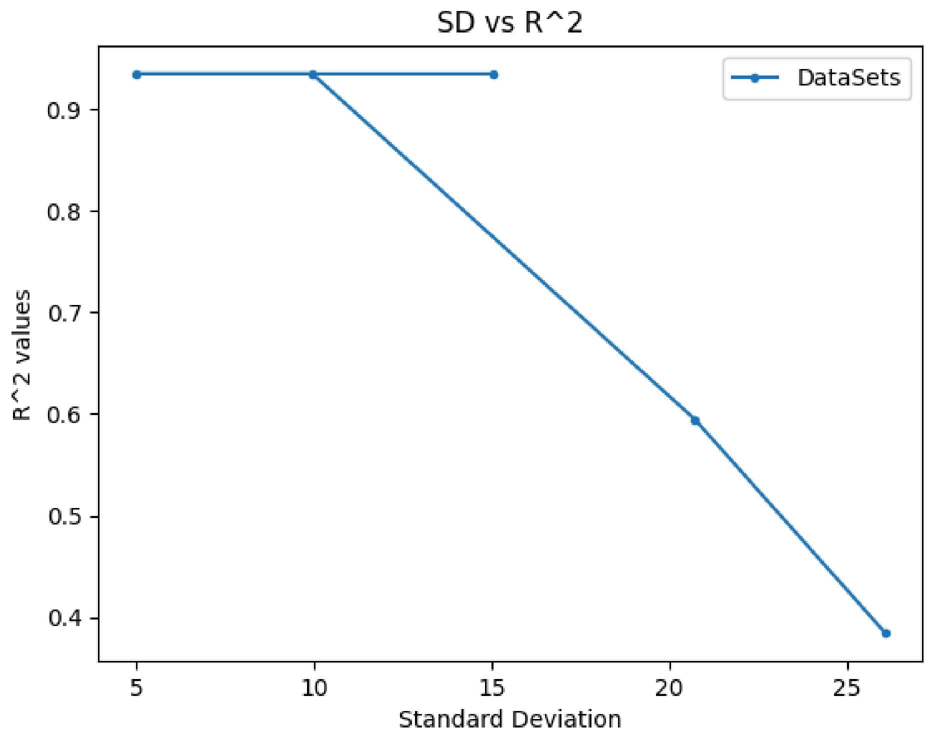


Analysis:
The major analysis will be given in Q.d). But an important detail i noticed from the data is that as the SD increases the spread becomes more, in the sense, if it feels as though a regression model won't accurately predict such a data set because it feels the errors will be large for any line.

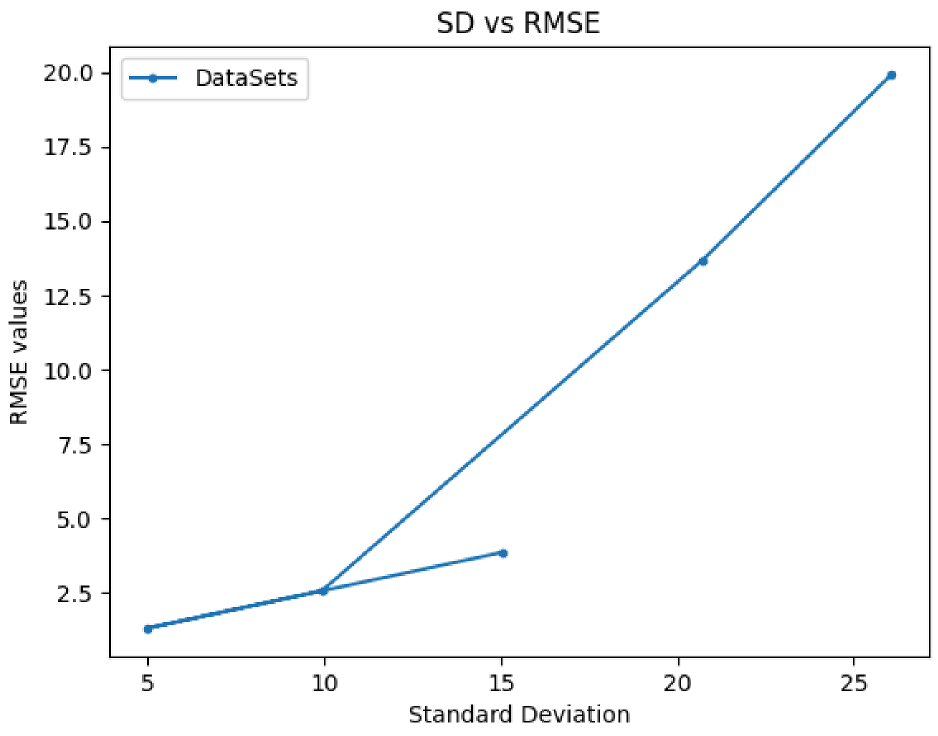
One thing I noticed about the residuals is that they have a pattern for the original and reduced cases. This is because the linear regression plot is an approximation and the kink i talked about earlier brings this issue. The pattern which was left out was randomized by the increased SD plots making it feel as though the residual plots for the increased SD case we better in some sense, but what it did was it blurred the y values in some sense.

Q.d)
To actually get a good comparison regarding the data, I have used a plot of SD vs all the above metrics. This gives us a fair idea on how good the model is for each case.

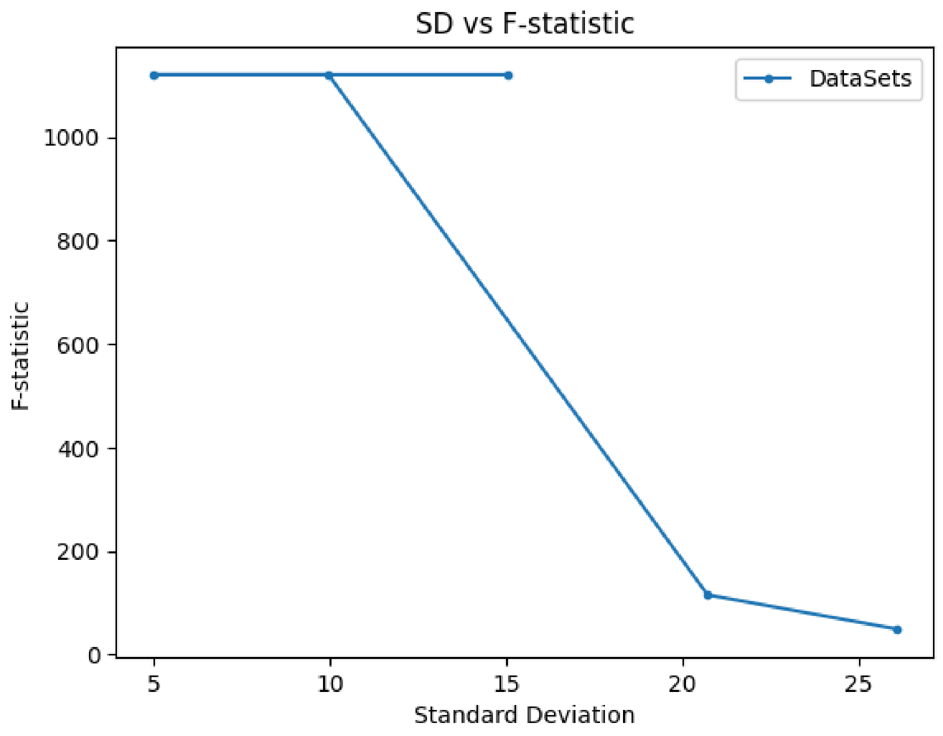
I have done this making arrays of the values from each dataset and plotting the SD list against the metric lists. The code is quite easy and nothing much is there in it for explanation, hence i haven't included it.



From this we find that the R^2 value stays approximately constant for the original and constant cases whereas the R^2 value decreases for the increased SD cases. This is mainly due to the inflation effect in the data points due to the presence of the noise added in the form of the normal distribution. This validates my earlier visual idea of the points being pretty scattered for the increased SD case. This just means that increasing the SD from the base value increases the residuals and therefore makes the R^2 value drop, making the model pretty bad.

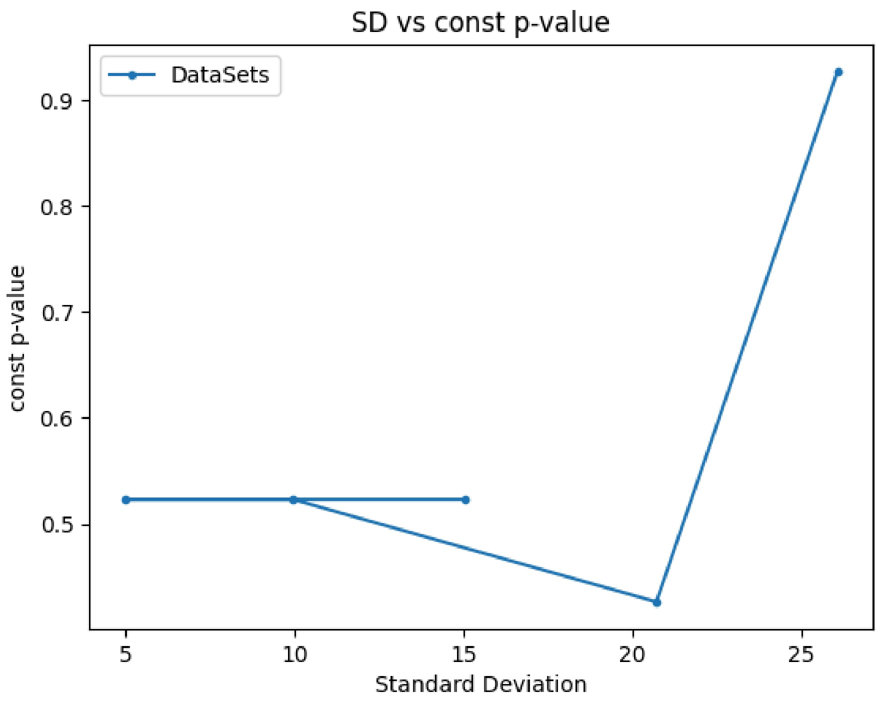


The above plot also tells a similar story. Increasing the SD in any manner, from any point increases the RMSE. This is because, lower the SD, lower is the spread of the values, ie: the points are closer together. So, when we increase the SD, the point spread out and added on top the increased from base value SDs actually have some noise also added, which increases the RMSE a lot.

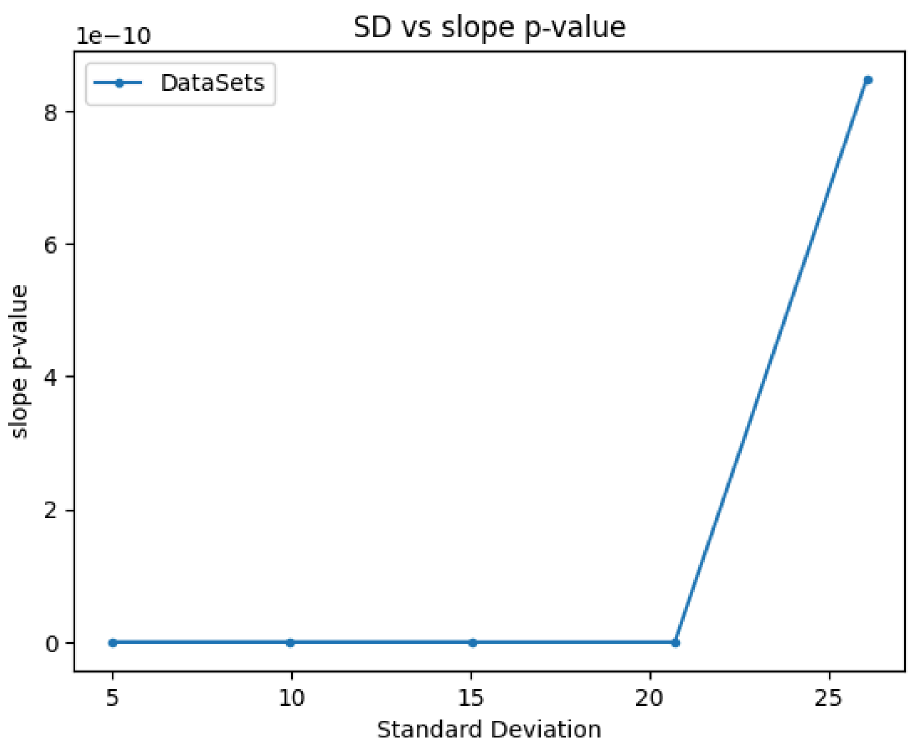


F-statistic measures how much of the variance in the model can be explained by the regression themselves rather than the residuals. Lesser the variance from the residuals, the better. F-statistic when higher, the better. So, from the above plot, we see that, higher the SD value, lower goes the F-statistic, even dropping below 100. Lower the F-statistic means more errors and noise in the model and that the model is not at all a good fit for the model. It just means that more the SD, more the spread of the points hence more of a bad fit. It is always best to keep the F-statistic as high as possible and the noise in the increased SD cases means the model cannot extract the pattern well. The lower SD cases, like all the previous plots show excellent metrics signifying they are very good fits.

I also have an F-statistic p-value plot but i had just found it as an extra plot to infer from. We can infer a similar result from that too that the higher SD cases introduce too much variability such that the model performs very badly.



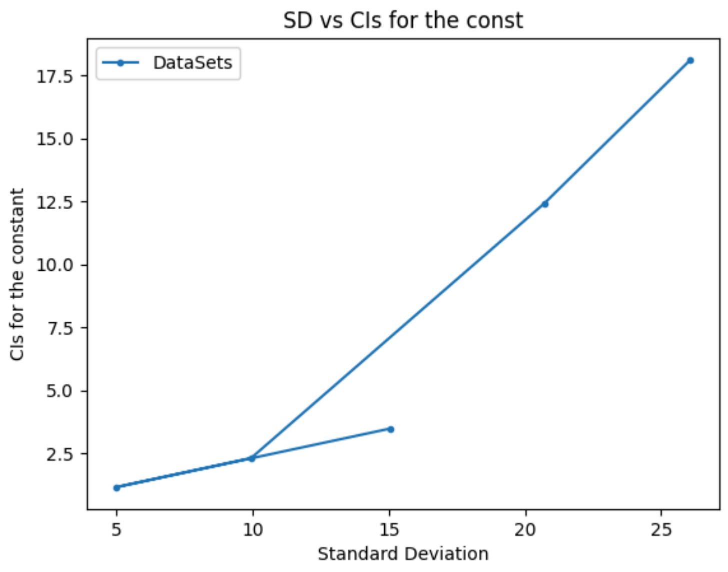
The p-values for all the SD values for the constant is more than 0.5, except for the SD20 case which seems to be a coincidence, because in the previous few runs i had encountered increased p-value for the SD20 case too. The p-value is close to 0.5 for almost all the models, indicating that the result may not be statistically significant. The origin is included in some of the CIs making is not a good choice. This plot doesn't reveal much but does give us a hint at the trend that the p-value increases with the increase in the SD, which means a larger CI and the inclusion of 0. An uncertainty is being added due to the noise addition.

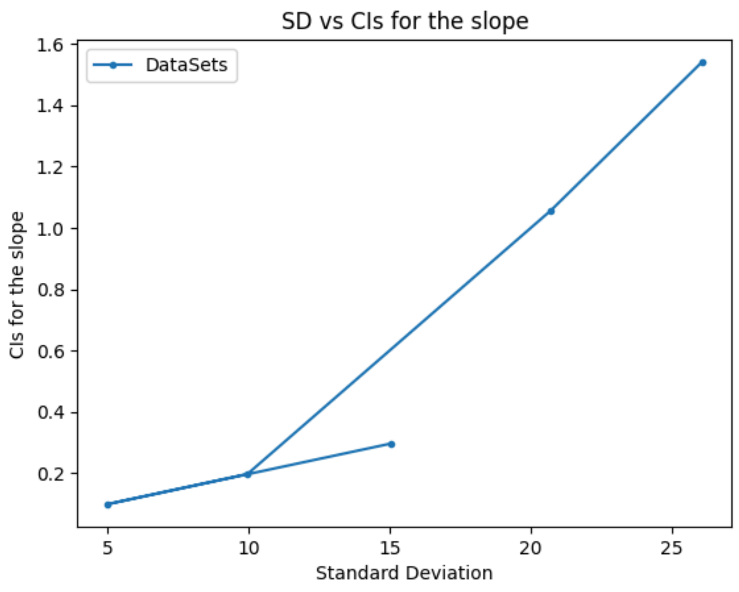


The slope p-value vs SD plot is shown above. The p-value shows a much better relation here. The increase in the SD shows that the p-value become more larger. This indicates an increase in the spread of the data points. This means there is more noise in the system and the system become less deterministic. The increase here is slight, the p-value is still statistically significant but according the plot above, the increase can be very fast and for higher values, the p-value can shoot up pretty fast and the model can be useless pretty fast.

Q.e)

This part deals with the confidence intervals. We already have an idea on the result from the p-values itself, but just to get a clear idea, we put in the following test. We check the width of the confidence interval. If it increases, it means the data shows more variability and is becoming more random and unpredictable in some sense.





The pattern I expected occurs again. We get quite similar plots but with the widths different. We again see here that, as the SD increases, the variability in the data points increases, this means more is the spread of the values. Hence the model defines larges slopes and constant in the CI to get a hold of all the points. This is just a solid proof that more the SD, more the variation due to addition of the noise. This means more is the parameter uncertainty.

LEARNINGS:

I learnt how to use python to code up regression lines and patterns.

The increasing SD and decreasing SD problem was a different type of problem and it really got me digging into why the above was true in the first place. After the read it seemed quite intuitive on how more the variation due to noise, more was the uncertainty.

I learnt how to properly choose the train and test data for any set using shufflers and also what is the optimum amount.

The main learning was how to intuitively predict how the pattern of regression might be simply looking at some of the metrics and also simple visual inspection.