Prediction of Stock Close Value

Drishti Chulani & Saloni Bera

Financial Elective Report

Submitted to the

School of Data Science and Business Intelligence

For the

Post Graduation Degree

In

Data Science and Business Analytics

April 2023

Abstract

The Stock Market is highly volatile, so understanding trends and accordingly predicting the value. Stock market prediction is the act of trying to determine the future value of a company stock or other financial instrument traded on an exchange. Machine learning and Data science helps a lot to find patterns in large data for predicting with a high degree of accuracy. The objective is to model the future close value of a stock that is traded in the market based on the data given. Therefore, an appropriate model is to be developed that captures the complexities and interactions of the variables that will be decided upon and is ultimately capable of producing an accurate prediction of the close price predicted.

Data Understanding

Data Exploration

The data was provided in a Comma Separated Values (CSV) file format. The data appears to be mostly cleaned and was given to us in 2 files (a train_test data file and validation data file). The dataset had 747 records consisting of 5 variables within them consisting of Date: to get the price of the stock at a particular date. Open: the price the stock opened at. High: the highest price during the day. Low: the lowest price during the day. Close: the closing price on the trading day. Adj Close: the closing price after adjustments for all applicable splits and dividend distributions. Volume: how many shares were traded.

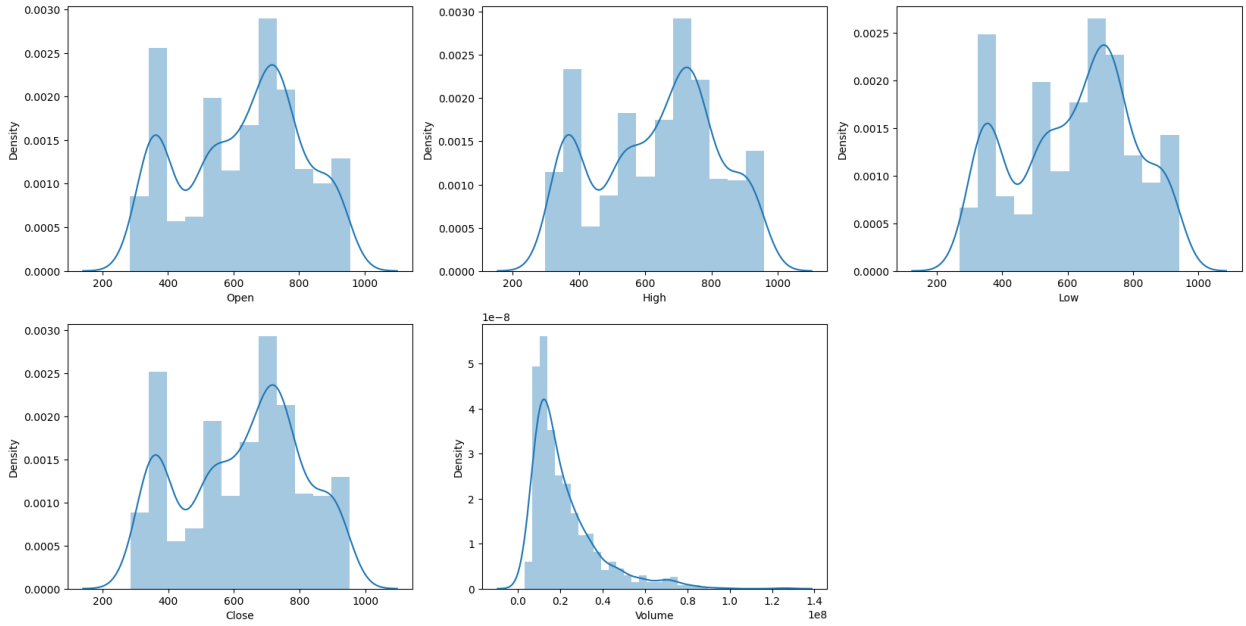
We assumed that the data was historical data for a particular stock with 3 years worth data starting from 01-01-2020 to 31-12-2022.

Data Importing and Cleaning

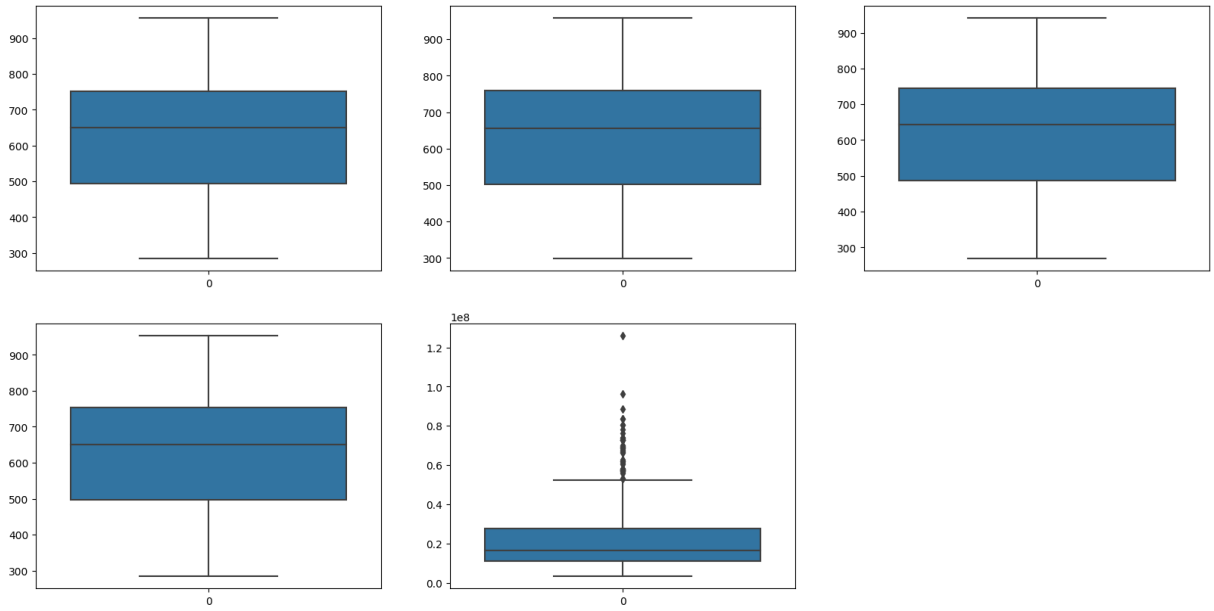
The files were imported into the colab notebook and were stored in the variable ‘df’. Codes were run to check the data, its structure, datatypes and missing values.

As the data shows the price of a stock, and given the stock market can be highly volatile at times we decided to keep and use all the data that was given to us without changing it. No missing values were found and outliers were studied. Apart from date and volume, the rest had float64 as their datatype, date had object and volume had integer datatype.

For further visualization, we have used distribution plots to get the continuous feature in the dataset. Date was indexed and Adj Close was removed.



In most of the distributions, we can see peaks which means the data has varied significantly in two regions whereas Volume data is left skewed because we assume there is more demand for stocks at low price, and to understand that more we plot the boxplot for the 5 variables.



In the box plot, we see there aren't many outliers in the first 4 variables but volume has a lot of outliers. Hence one solution is to take a log of volume to smooth out the data. Further details will be explained below.

Data Analysis

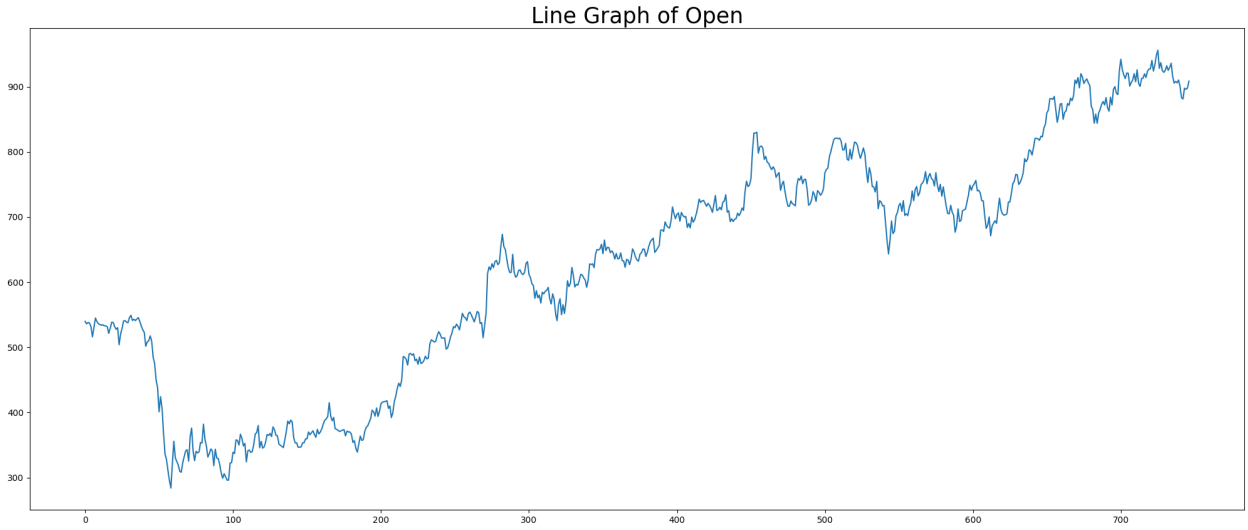
Variable Identification

After examining the structure of all the given variables, deleting redundant ones, the response and explanatory variables were determined. As previously known, Adj Close was deleted as it was not significantly important in the analysis. Naturally, the Close variable would be the dependent (response) variable, and the remaining 4 variables were potential independent (explanatory) variables.

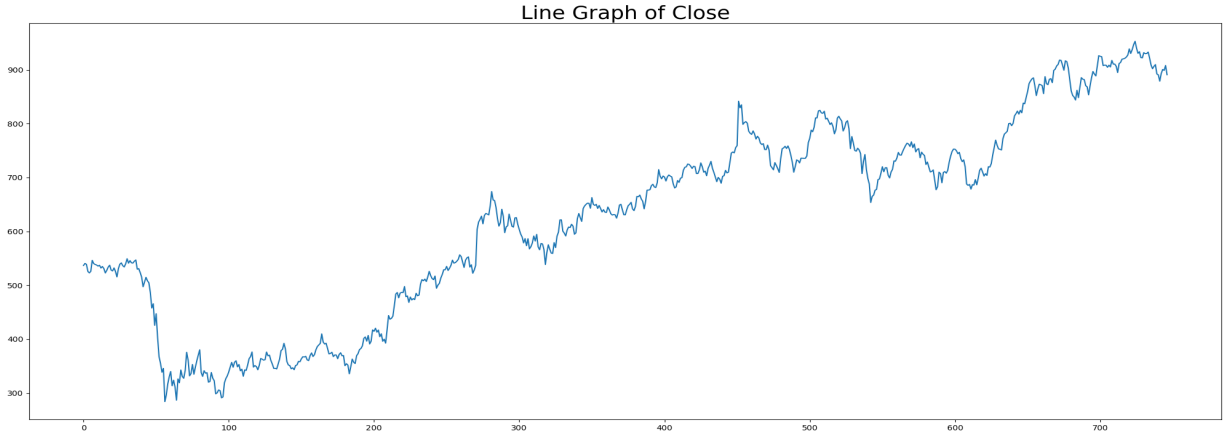
Data Exploratory

To gain a better insight of the data, graphs were created using packages like matplotlib (all codes are available in the attached .ipynb script). Following is a basic analysis of the data trends are shown below.

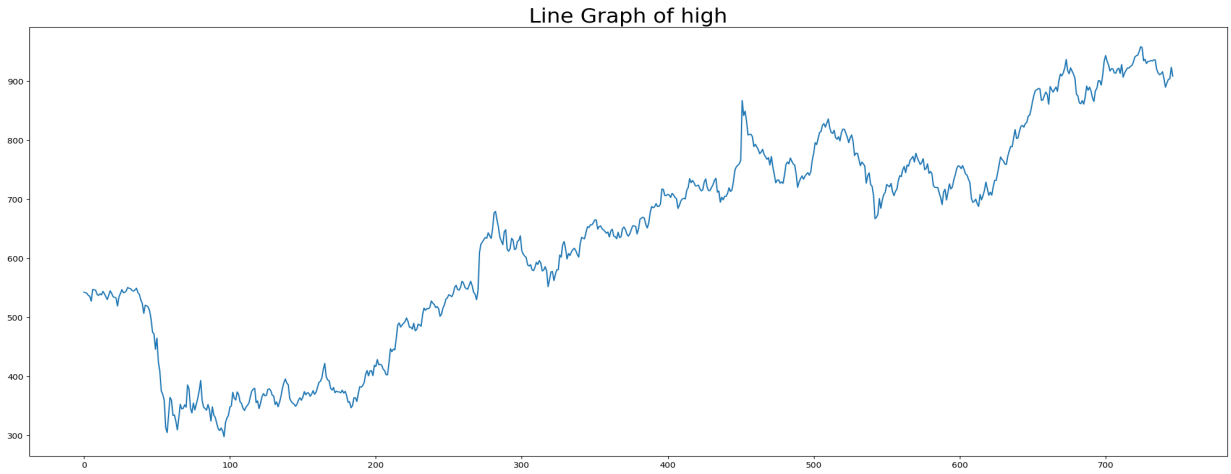
1) Open stock price fluctuation over 3 years



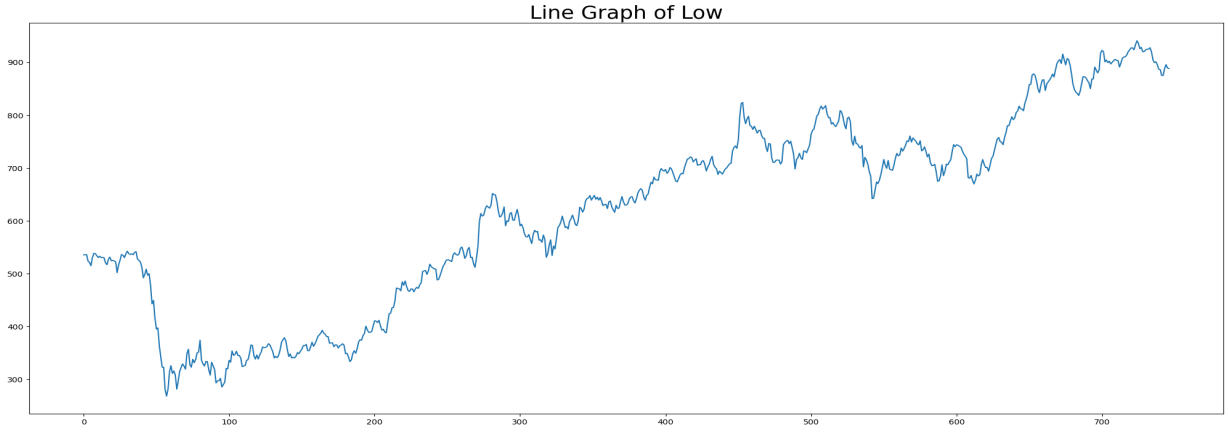
2) Close stock price fluctuation over 3 year



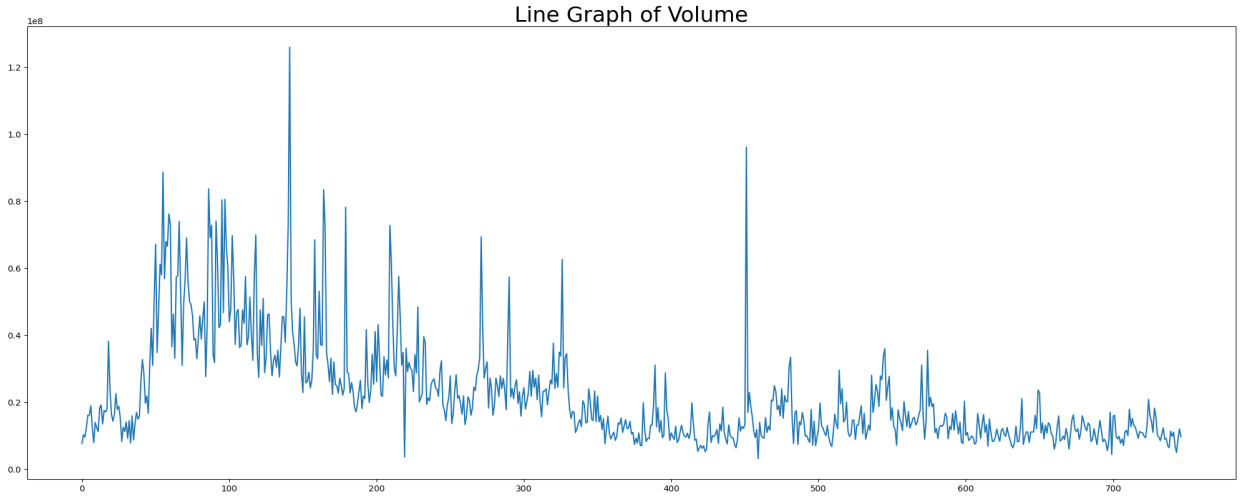
3) Highest stock price for the particular day fluctuation over 3 years



4) Lowest stock price for the particular day fluctuation over 3 years



5) Volume of the stock fluctuation over 3 years

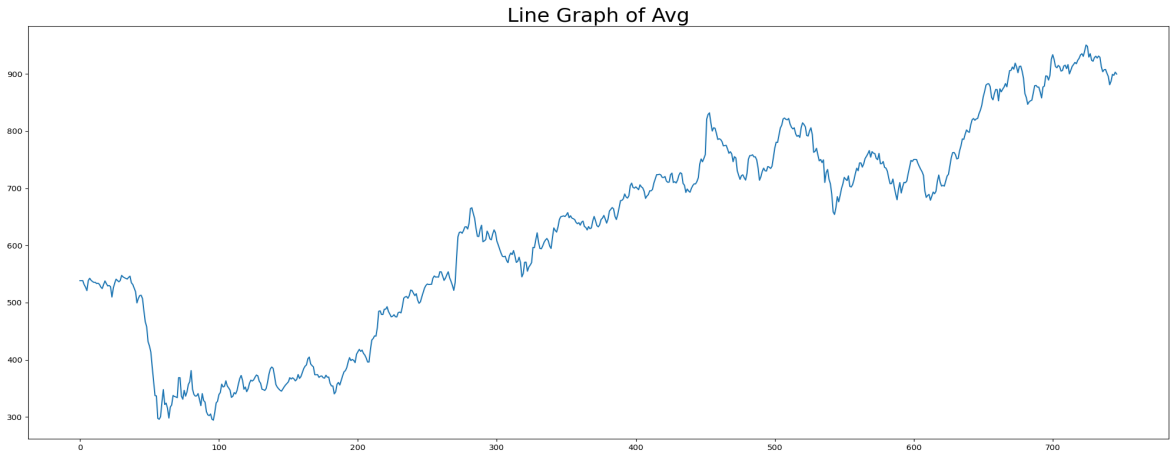


We also decided to create a few new variables namely, we added an **average price variable** and a **10-day moving average variable**. A moving average (MA) is a stock indicator that is commonly used in technical analysis. The reason for calculating the moving average of a stock is to help smooth out the price data over a specified period of time by creating a constantly updated average price. We calculate the moving average using the number of prices within a time periods and divide it by the number of total periods, for this case divide by 10.

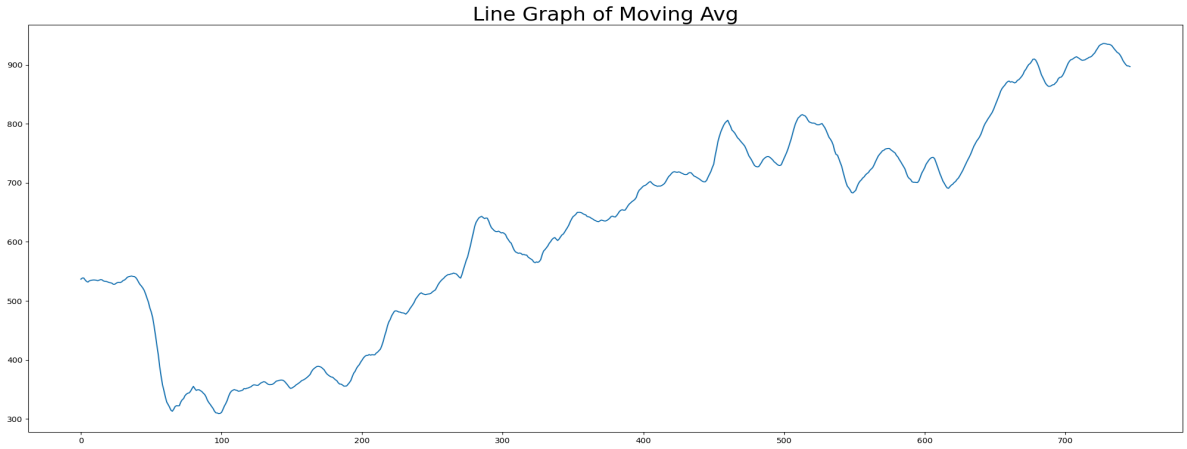
Furthermore, we also decided to take the log of the volume variable to help smoothen it out since there were a lot of fluctuations also, we took the log of other variables too. Log Transformation helps reduce the skewness in the data.

Plots

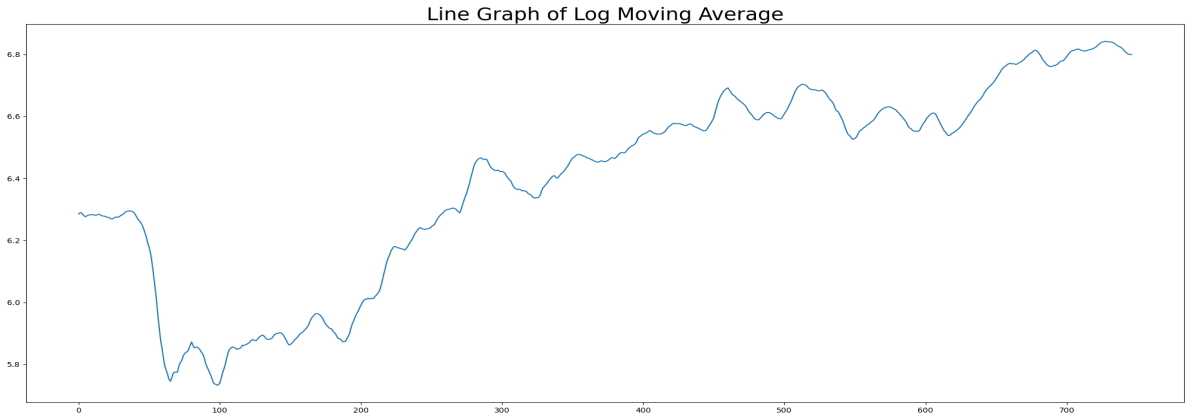
1) We have found the trend for the average price between Open and Close.

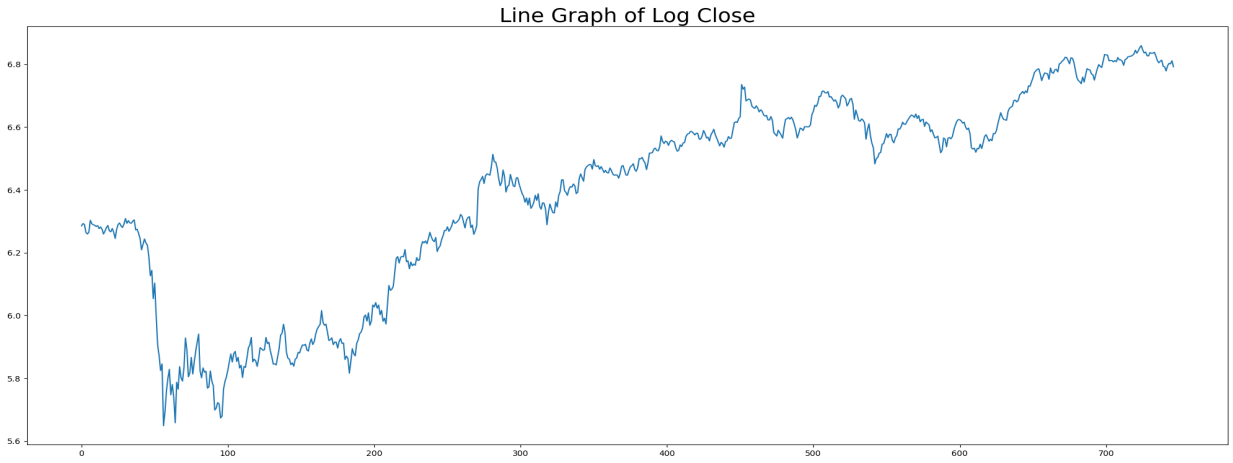


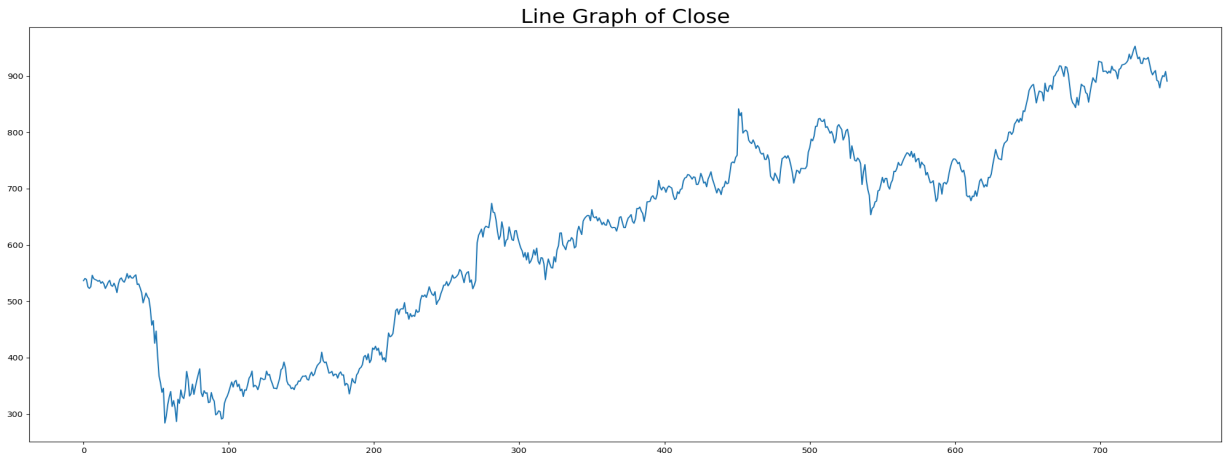
2) The trend for the moving average for the 3 years to help smoothening.



As we can see, the moving average variable has smoothened the price.



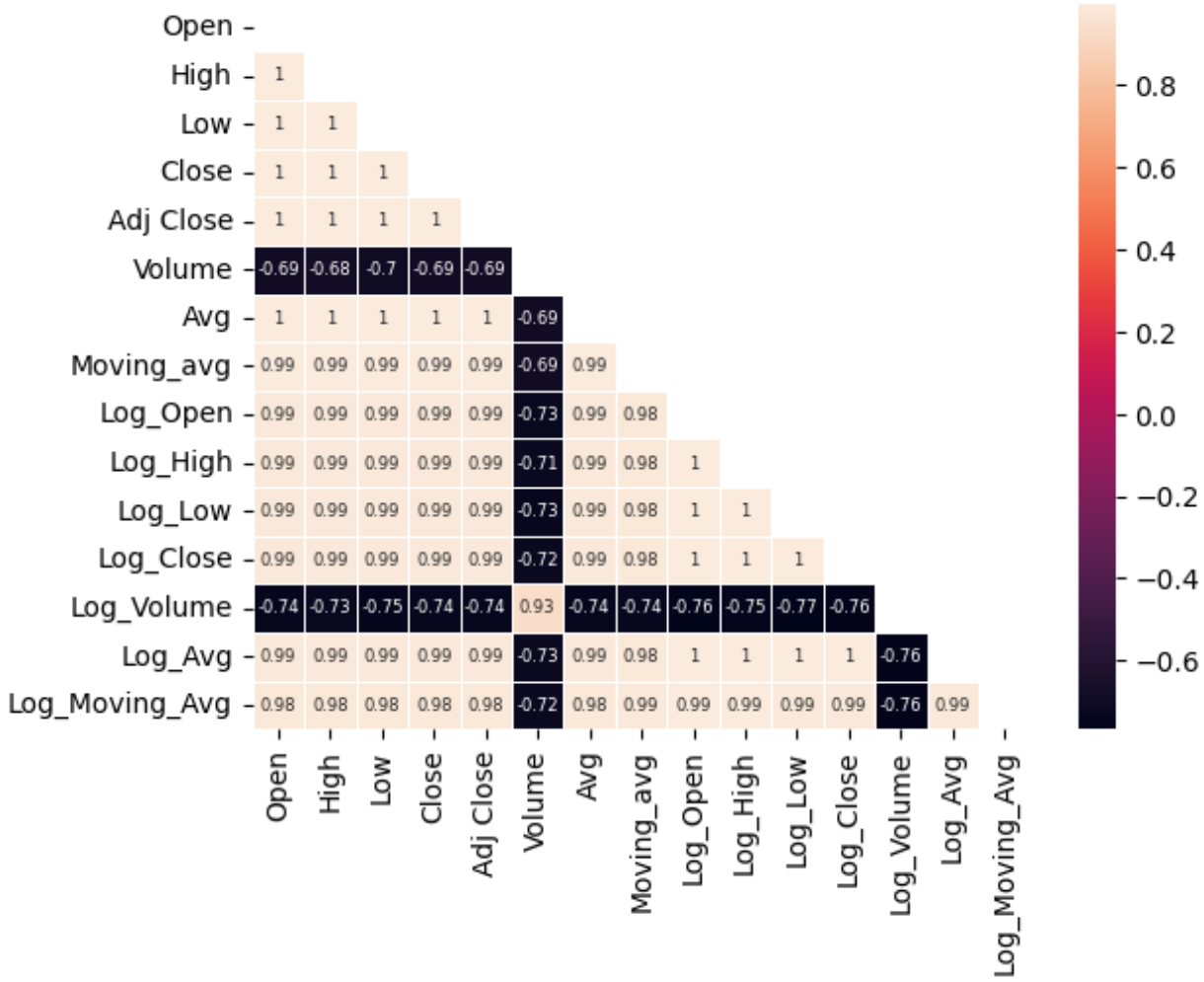




There is a slight difference between the moving average and log of moving average and close price and log of close price, and the scale of the Y-axis has also been adjusted.

A correlation heatmap:

A correlation heatmap is made to see the correlation between all the 15 variables (basic as well as log of these variables) has been made:



We can observe that all the variables have a negative correlation to the volume and log_volume variables. This means that as the price (for eg. Close) of the stock increase, the volume that is traded decreases.

Model Building

Data Presentation

As we have been asked to predict Close Price for the next day of the given data, we introduce a new variable called Close_Forecast having our next day Close price.The last row is removed.

```
df['Close_Forecast']=df['Close'].shift(-1)
df['Log_CF'] = np.log(df['Close_Forecast'])
```

```
df = df[:-1]
```

LINEAR REGRESSION

Iteration 1- High-Low Difference

Step 1: X and Y Variables

In this, our independent variable is Open, High, low, and the difference between High-low, this is for creating a simple linear regression without scaling the variables or log of the variables. The response variable is Close_Forecast.

```
x = df.loc[:,['Open', 'High', 'Low', 'H-L']]
```

```
y = df.loc[:,["Close_Forecast"]]
```

Step 2: Splitting the dataset

We need to split the data into two parts that is train and test data before fitting any model on our dataset. Where we fit our model on training data and evaluate our model on a test dataset. The split of train and test data can be in any ratio with the train dataset having more than 50% of the data. In our case, we have split the dataset into train and test data in a 7:3 ratio respectively with random state being zero. So our train set had 523 records and the test set had 224 records.

```
x_train = x[:523]
x_test =x[523:747] #70%
y_train = y[:523]
y_test =y[523:747]
```

Step 3: Fitting Linear Regression

Next we have to fit a linear regression to data. Linear Regression is a basic and commonly used type of predictive analysis. The overall idea of a linear regression is to examine how well a set of predictor variables does a good job in predicting an outcome variable. The regression estimates are used to explain the relationship between one dependent variable and one or more independent variables. The simplest form of the regression equation with one dependent variable and one independent variable is defined by the formula y = c + b*x, where y is an estimated dependent variable, c is intercept, b = regression coefficient and x is score on the independent variable. All the variables should be quantitative.

We fit the Linear Regression from the sklearn.linear_model library on the trainset.

```
model = LinearRegression()
```

```
model.fit(x_train, y_train)
```

```
▾ LinearRegression
LinearRegression()
```

The Intercept of the model is:

```
print(model.intercept_)
```

```
[2.93391591]
```

The coefficient of the model is:

```
print(model.coef_)
```

```
[[-5.20964152e-01 -1.55451316e+10  1.55451316e+10  1.55451316e+10]]
```

The model score of the train and test dataset

```
model.score(x_test, y_test)
```

```
0.9809935589014906
```

```
model.score(x_train, y_train)
```

```
0.9929763996918293
```

Step 4: Prediction

Using this model, we predict the values by using the code:

```
y_pred = model.predict(x_test)
```

And a table of Actual and predicted values are:

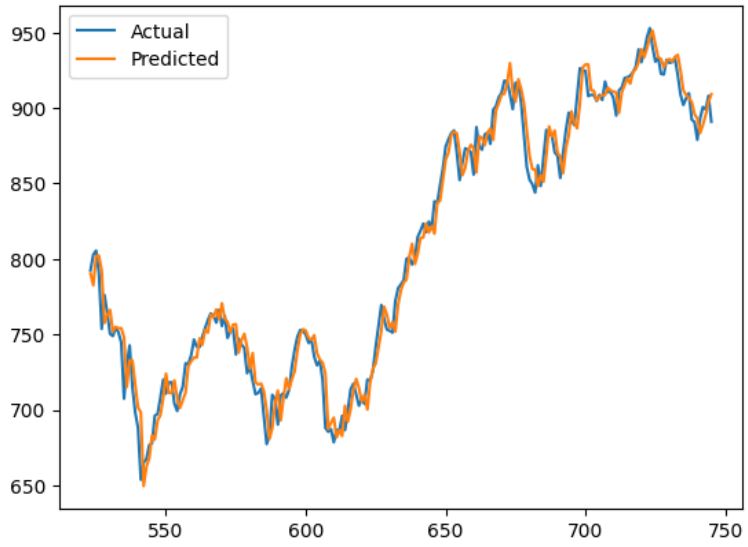
```
df_pred=pd.DataFrame(y_test.values,columns=['Actual'], index=y_test.index)
```

```
df_pred
```

	Actual	Predicted
523	792.500000	790.527666
524	802.799988	782.521807
525	805.500000	802.041338
526	790.799988	802.211260
527	753.700012	792.330400
...	...	...
741	893.200012	883.371416
742	900.650024	888.859697
743	898.950012	895.487383
744	908.049988	903.999849
745	890.849976	909.089495

223 rows × 2 columns

The plot to show the difference between the Actual and Predicted values



The code and output of the error

```
print("Mean Absolute Error:",metrics.mean_absolute_error(y_test,y_pred))
print("Mean Squared Error:",metrics.mean_squared_error(y_test,y_pred))
print("Root Mean Squared Error:",np.sqrt(metrics.mean_squared_error(y_test,y_pred)))
```

```
Mean Absolute Error: 9.211704415825864
Mean Squared Error: 143.1640535459512
Root Mean Squared Error: 11.965118200249892
```

Based on the given statistics, it appears that the model's performance is evaluated using three common metrics: Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE).

1. Mean Absolute Error (MAE): The MAE is a measure of the average absolute difference between predicted and actual values. In this case, the MAE is 9.211, which indicates that on average, the model's predictions deviate from the actual values by approximately 9.21 units. A lower MAE value suggests better accuracy of the model.

2. Mean Squared Error (MSE): The MSE is a measure of the average squared difference between predicted and actual values. In this case, the MSE is 143.164, which indicates that on average, the model's predictions deviate from the actual values by approximately 143.16 units squared. MSE is more sensitive to outliers as it involves squaring the errors, and higher MSE values indicate higher prediction errors.

3. Root Mean Squared Error (RMSE): The RMSE is the square root of the MSE and is commonly used to interpret the prediction error in the same unit as the original data. In this case, the RMSE is 11.9651, which indicates that on average, the model's predictions deviate from the actual values by approximately 11.97 units. Like the MAE, a lower RMSE value suggests better accuracy of the model.

In summary, the given statistics suggest that the model's predictions have an average error of around 9.21 units (MAE), with some higher errors indicated by the MSE of 143.16 units squared. The RMSE of 11.97 units provides an easily interpretable measure of the prediction error in the original unit of the data. Overall, the model's performance is not so good hence we will try some more iterations.

Iteration 2- Log_Average

Step 1: X and Y Variables

With our previous preparation, our response variable is Close_Forecast, now since we have log all our variables our response variable comes out to be Log_CF and independent variable is Log_Avg, this log of average of open and close price

```
X1 = df['Log_Avg'].values.reshape(-1,1)
Y1 = df['Log_CF'].values.reshape(-1,1)
```

Step 2: Splitting the dataset

We need to split the data into two parts that is train and test data before fitting any model on our dataset. Where we fit our model on training data and evaluate our model on a test dataset. The split of train and test data can be in any ratio with the train dataset having more than 50% of the data. We split the dataset into train and test data in a ratio of 6:1 respectively with random state being zero. So our train set had 621 records and the test set had 125 records.

```
#Splitting data into Train & Test
X_train1,X_test1,Y_train1,Y_test1=train_test_split(X1,Y1,test_size=(1/6),random_state=0)
```

Step 3: Fitting Linear Regression

We fit the Linear Regression from the sklearn.linear_model library on the trainset.

```
regressor1 = LinearRegression()
regressor1.fit(X_train1,Y_train1)
```

```
▾ LinearRegression
LinearRegression()
```

The Intercept of our model:

```
print(regressor1.intercept_)
```

```
[0.00416159]
```

The Coefficient of our model:

```
print(regressor1.coef_)
```

```
[[0.99957594]]
```

Step 4: Prediction

Using this model, we predict the values by using the code:

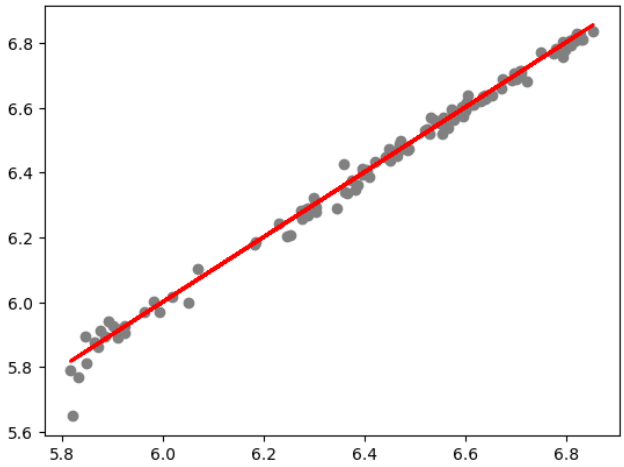
```
Y_pred1 = regressor1.predict(X_test1)
```

And a table of Actual and predicted values are:

```
dataset1 = pd.DataFrame({"Actual":Y_test1.flatten(),"Predicted":Y_pred1.flatten()})
dataset1.head(10)
```

	Actual	Predicted
0	6.782815	6.780356
1	6.686672	6.693066
2	6.259295	6.278143
3	6.302253	6.302275
4	5.907403	5.926506
5	5.940566	5.892548
6	6.408693	6.407500
7	5.813683	5.849853
8	6.812125	6.835238
9	6.280302	6.290653

A Scatter plot



The code and output for measuring the errors

```
print("Mean Absolute Error:",metrics.mean_absolute_error(Y_test1,Y_pred1))
print("Mean Squared Error:",metrics.mean_squared_error(Y_test1,Y_pred1))
print("Root Mean Squared Error:",np.sqrt(metrics.mean_squared_error(Y_test1,Y_pred1)))
```

```
Mean Absolute Error: 0.01711965810886071
Mean Squared Error: 0.0006845246525655191
Root Mean Squared Error: 0.026163422034694146
```

In summary, the given statistics suggest that the model's predictions have a very low average error of around 0.0171 units (MAE), with even lower errors indicated by the MSE of 0.0006845 units squared. The RMSE of 0.0262 units provides an easily interpretable measure of the prediction error in the original unit of the data. Overall, the model's performance is very good, with very low prediction errors, indicating high accuracy in predicting the actual values.

Subsequent Models

Now to compare our model with other models.

Iteration 3- Log of all Variables

Step 1: X and Y Variables

Now to compare, here we built another model where we had all the Log variables as independent variables and Log_CF as dependent variables

```
X=df.iloc[:,[3,4,5,7,8,9]]
Y=df['Log_CF'].values.reshape(-1,1)
```

Step 2: Splitting the dataset

We split the dataset into train and test data in a ratio of 6:1 respectively with random state being zero. So our train set had 621 records and the test set had 125 records.

```
X_train,X_test,Y_train,Y_test=train_test_split(X,Y,test_size=(1/6),random_state=0)
```

```
X_train.shape , Y_train.shape
```

```
((621, 6), (621, 1))
```

Step 3: Fitting Linear Regression

We fit the Linear Regression from the sklearn.linear_model library on the trainset.

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

```
▾ LinearRegression
LinearRegression()
```

The intercept of the model is

```
print(regressor.intercept_)
```

```
[0.05092244]
```

The coefficient of the model is

```
print(regressor.coef_)
```

```
[[-1.02900531e+00  2.27677539e-01  2.55522500e-01 -1.03836330e-03
   1.49061286e+00  5.02947703e-02]]
```

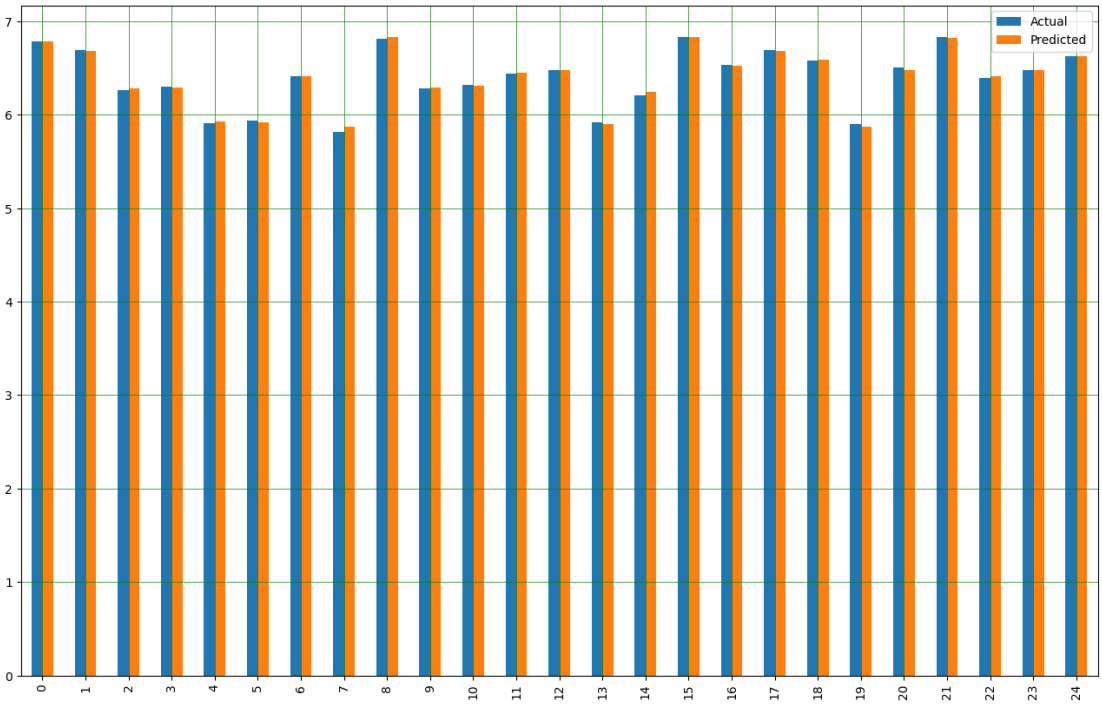
The table of actual and predicted values are:

```
dataset=pd.DataFrame({"Actual":Y_test.flatten(),"Predicted":Y_pred.flatten()})
dataset.head(10)
```

	Actual	Predicted
0	6.782815	6.787743
1	6.686672	6.679180
2	6.259295	6.280952
3	6.302253	6.294131
4	5.907403	5.926400
5	5.940566	5.920342
6	6.408693	6.411830
7	5.813683	5.869379
8	6.812125	6.825948
9	6.280302	6.287352

The bar plot of actual and predicted of first 25 row is:

```
dataset_1=dataset.head(25)
dataset_1.plot(kind="bar",figsize=(16,10))
plt.grid(which="major",linestyle="-",linewidth="0.5",color="green")
plt.grid(which="minor",linestyle=":",linewidth="0.5",color="black")
plt.show()
```



The code and output for errors

```
print("Mean Absolute Error:",metrics.mean_absolute_error(Y_test,Y_pred))
print("Mean Squared Error:",metrics.mean_squared_error(Y_test,Y_pred))
print("Root Mean Squared Error:",np.sqrt(metrics.mean_squared_error(Y_test,Y_pred)))
```

```
Mean Absolute Error: 0.01640306475070111
Mean Squared Error: 0.0008422574777604763
Root Mean Squared Error: 0.0290216725527747
```

In summary, the given statistics suggest that the model's predictions have a very low average error of around 0.0164 units (MAE), with relatively low errors indicated by the MSE of 0.0008422 units squared. The RMSE of 0.0290 units provides an easily interpretable measure of the prediction error in the original unit of the data. Overall, the model's performance is very good, with low prediction errors, indicating high accuracy in predicting the actual values and better than iteration 2.

Iteration 4- Log of all variables except Log_Avg & Log_Moving_Avg

Step 1: X and Y Variables

Now to compare, here we built another model where we had all the Log variables as independent variables apart from Log_Avg & Log_Moving_Avg and Log_CF as dependent variables.

```
x=df.iloc[:,[3,4,5,7]]
y=df['Log_CF'].values.reshape(-1,1)
```

Step 2: Splitting the dataset

We split the dataset into train and test data in a ratio of 6:1 respectively with random state being zero. So our train set had 621 records and the test set had 125 records.

```
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=(1/6),random_state=0)
```

```
x_train.shape , y_train.shape
```

```
((621, 4), (621, 1))
```

Step 3: Fitting Linear Regression

We fit the Linear Regression from the sklearn.linear_model library on the trainset.

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

```
▾ LinearRegression
LinearRegression()
```

The intercept of the model is

```
print(regressor.intercept_)
```

```
[0.07568957]
```

The coefficient of the model is

```
print(regressor.coef_)
```

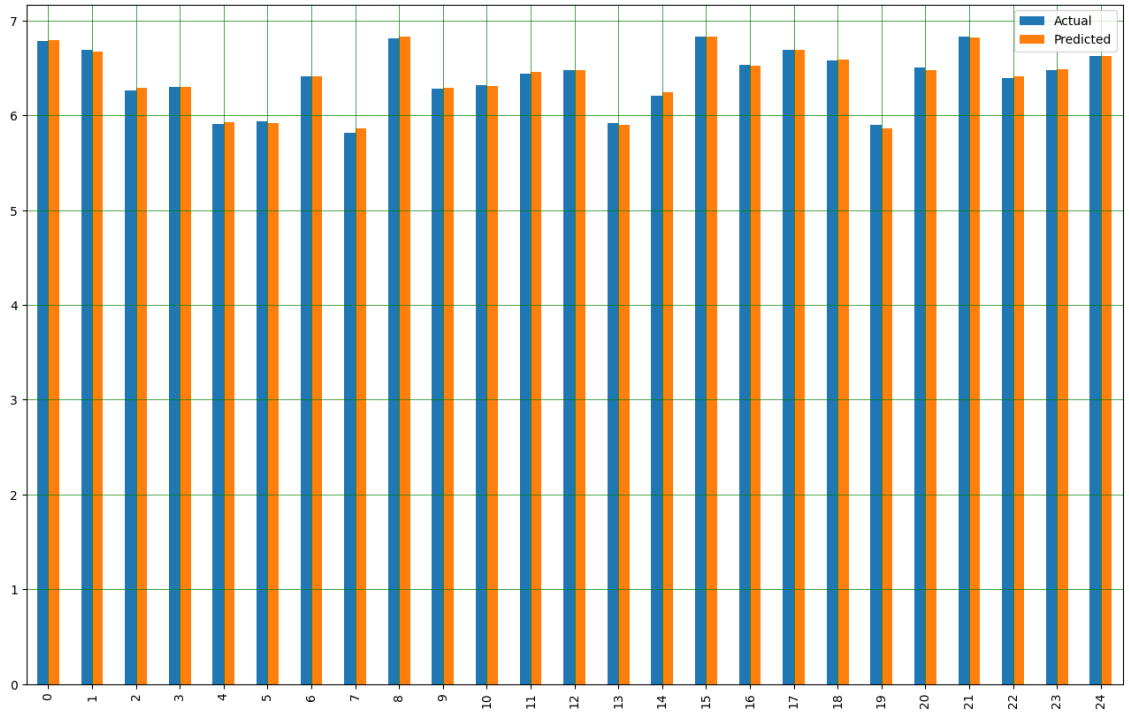
```
[[-0.62500528  0.79655673  0.82173495 -0.00181307]]
```

The table of actual and predicted values are:

```
dataset_3=pd.DataFrame({"Actual":y_test.flatten(),"Predicted":y_pred.flatten()})
dataset_3.head(10)
```

	Actual	Predicted
0	6.782815	6.789483
1	6.686672	6.673279
2	6.259295	6.288787
3	6.302253	6.295729
4	5.907403	5.924620
5	5.940566	5.919255
6	6.408693	6.409831
7	5.813683	5.862146
8	6.812125	6.827646
9	6.280302	6.290200

The bar plot of actual and predicted of first 25 row is:



The code and output for errors

```
print("Mean Absolute Error:",metrics.mean_absolute_error(y_test,y_pred))
print("Mean Squared Error:",metrics.mean_squared_error(y_test,y_pred))
print("Root Mean Squared Error:",np.sqrt(metrics.mean_squared_error(y_test,y_pred))
```

```
Mean Absolute Error: 0.017477170027702188
Mean Squared Error: 0.0008363423587635003
Root Mean Squared Error: 0.0289195843463128
```

We removed Log_Avg & Log_Moving_Avg as it is dependent on the close price. As our validation dataset does not consists close price this variable cannot be added to the test hence, this iteration was added.

In summary, the given statistics suggest that the model's predictions have a relatively low average error of around 0.0175 units (MAE), with relatively low errors indicated by the MSE of 0.0008363 units squared. The RMSE of 0.0289 units provides an easily interpretable measure of the prediction error in the original unit of the data. Overall, the model's performance is good, with low prediction errors, indicating high accuracy in predicting the actual values.

Ridge Estimator

Ridge regression is a model tuning method that is used to analyze any data that suffers from multicollinearity. When the issue of multicollinearity occurs, least-squares are unbiased, and variances are large, this results in predicted values to be far away from the actual values.

Ridge helps in penalizing the sum of squared coefficients (L2 penalty) meaning shrinking the coefficients for those input variables that do not contribute much to the prediction task.
The ridge problem penalizes large regression coefficients, and the larger the parameter lambda is, the larger the penalty. The estimator bias increases with λ and the estimator variance decreases with λ. The optimal level for λ is the one that minimizes the root mean squared error (RMSE) or R-squared. In order to fit a ridge model, we'll use the Ridge() function.

Iteration 3 for Ridge:

The code and output:

```
#Iteration 3
ridgeR = Ridge()
ridgeR.fit(X_train, Y_train)
ridgeR_y_pred = ridgeR.predict(X_test)
print('Ridge Mean Square Error:',metrics.mean_squared_error(Y_test,ridgeR_y_pred))
print('Ridge Root Mean Square Error:',np.sqrt(metrics.mean_squared_error(Y_test,ridgeR_y_pred)))
```

```
Ridge Mean Square Error: 0.0008186457801252346
Ridge Root Mean Square Error: 0.028611986651143863
```

Iteration 4 for Ridge

```
# Iteration 4
ridgeR = Ridge()
ridgeR.fit(x_train, y_train)
ridgeR_y_pred_1 = ridgeR.predict(x_test)
print('Ridge Mean Square Error:',metrics.mean_squared_error(y_test,ridgeR_y_pred_1))
print('Ridge Root Mean Square Error:',np.sqrt(metrics.mean_squared_error(y_test,ridgeR_y_pred_1)))
```

```
Ridge Mean Square Error: 0.0007314649413900855
Ridge Root Mean Square Error: 0.027045608541685386
```

Iteration 4 has a better Ridge regression model than Iteration 3 because of its good performance with low prediction errors, as indicated by the low values of Ridge MSE and Ridge RRMSE.

Lasso

LASSO stands for Least Absolute Shrinkage and Selection Operator. Lasso regression performs L1 regularization, i.e. it adds a factor of sum of absolute value of coefficients in the optimization objective. The parameter α (alpha) works similar to that of ridge and provides a trade-off between balancing RSS and magnitude of coefficients. Like that of a ridge, α can take various values. After using ridge, we will check whether the lasso can yield either a more accurate or a more interpretable model than ridge regression. In order to fit a lasso model, we'll use the Lasso() function. lasso adds a penalty for coefficients, but instead of penalizing the sum of squared coefficients (L2 penalty), lasso penalizes the sum of absolute values (L1 penalty).

Iteration 3 for Lasso:

The code and output:

```
#Iteration 3
Lasso = Lasso(alpha=0.1)
Lasso.fit(X_train, Y_train)
Lasso_y_pred = Lasso.predict(X_test)
print('Lasso Mean Square Error:',metrics.mean_squared_error(Y_test,Lasso_y_pred))
print('Lasso Root Mean Square Error:',np.sqrt(metrics.mean_squared_error(Y_test,Lasso_y_pred)))
```

```
Lasso Mean Square Error: 0.06313566021746744
Lasso Root Mean Square Error: 0.25126810425811597
```

Iteration 4 for Lasso:

```
# Iteration 4
Lasso.fit(x_train, y_train)
Lasso_y_pred_1 = Lasso.predict(x_test)
print('Lasso Mean Square Error:',metrics.mean_squared_error(y_test,Lasso_y_pred_1))
print('Lasso Root Mean Square Error:',np.sqrt(metrics.mean_squared_error(y_test,Lasso_y_pred_1)))
```

```
Lasso Mean Square Error: 0.06313566021746744
Lasso Root Mean Square Error: 0.25126810425811597
```

These statistics suggest that the Lasso regression model has relatively higher prediction errors compared to the Ridge regression model (as shown in the previous response), as indicated by the higher values of Lasso MSE and Lasso RMSE.

Elastic Net

Elastic net is a popular type of regularized linear regression that combines two popular penalties, ridge and lasso, the L1 and L2 penalty functions. Elastic Net combines the penalties of ridge and lasso to get the best of both worlds. Elastic Net also allows us to tune the alpha parameter where alpha = 0 corresponds to Ridge regression and alpha = 1 to Lasso regression. In order to fit Elastic net, we use “ElasticNet()” function.

Iteration 3 for Elastic

The code and output:

```
#iteration 3
elastic = ElasticNet(alpha=0.1)
elastic.fit(X_train, Y_train)
elastic_y_pred = elastic.predict(X_test)
print('ElasticNet Mean Square Error:',metrics.mean_squared_error(Y_test,elastic_y_pred))
print('ElasticNet Root Mean Square Error:',np.sqrt(metrics.mean_squared_error(Y_test,elastic_y_pred)))
```

```
ElasticNet Mean Square Error: 0.03085389430872103
ElasticNet Root Mean Square Error: 0.17565276629965448
```

Iteration 4 for Elastic

```
#Iteration 4
elastic = ElasticNet(alpha=0.1)
elastic.fit(x_train, y_train)
elastic_y_pred_1 = elastic.predict(x_test)
print('ElasticNet Mean Square Error:',metrics.mean_squared_error(y_test,elastic_y_pred_1))
print('ElasticNet Root Mean Square Error:',np.sqrt(metrics.mean_squared_error(y_test,elastic_y_pred_1)))
```

```
ElasticNet Mean Square Error: 0.03262754647796022
ElasticNet Root Mean Square Error: 0.18063096766047682
```

These statistics suggest that the ElasticNet regression model has intermediate performance compared to the Ridge and Lasso regression models (as shown in the previous responses), as indicated by the moderate values of ElasticNet MSE and ElasticNet RMSE.

Final Model Selection

Iteration 4 has the best results in all Ridge, Lasso and Elastic net.

Overall, these statistics suggest that the final model has good accuracy with low prediction errors, as indicated by the low values of MAE, MSE, and RMSE. However, it's important to consider the specific context and requirements of the problem being solved, and evaluate the model's performance in comparison to the desired level of accuracy or industry standards. Additionally, other factors such as model interpretability, scalability, and applicability to real-world scenarios should also be taken into consideration in determining the suitability of the final model.

For your validation,

```
ridgeR = Ridge()
ridgeR.fit(x_train, y_train)
ridgeR_y_prediction = ridgeR.predict(x_test)
```

We got the predicted and actual values in our dataset and inverted our log of the variables so that our calculation can be more accurate and have shifted the price one day later.

```
final_data['Invlog_Actual'] = np.exp(final_data['Actual'])
final_data['Invlog_Predicted'] = np.exp(final_data['Predicted'])
final_data
```

	Actual	Predicted	Invlog_Actual	Invlog_Predicted
0	6.782815	6.777869	882.549988	878.194984
1	6.686672	6.682997	801.650024	798.709463
2	6.259295	6.286642	522.849976	537.346077
3	6.302253	6.308098	545.799988	548.999733
4	5.907403	5.927077	367.750000	375.056612
...	...	...	...	...
120	5.893990	5.883696	362.850006	359.134044
121	6.606853	6.598946	740.150024	734.320691
122	6.001539	5.978686	404.049988	394.921010
123	6.638633	6.657199	764.049988	778.367763
124	6.274668	6.287851	530.950012	537.995749

125 rows × 4 columns

And this was then converted into an excel file.

Conclusion and Insights

- Stock market is too volatile; hence the presence of the outliers is justified and no data preparation was done on them..
- In the Data Exploration section we had checked the data for missing values and duplicate values, and we found none.
- There was a drop in the prices in the second quarter of 2020 which we assumed to be an impact from the COVID-19.
- After doing all the data exploration checks, no significant insights could be drawn. And we wanted to add a single variable that could be the representative of the price, as Open, Close, High and Low are just variations of the same price. To combat this, we added a new variable “Average”, which is the average of open and close prices.
- New variable known as “moving average” is added to smoothen the fluctuations in the data.
- Log of all variables: There was a huge difference between the ranges of the dependent variables and our response variable. For example, our open and close prices are in ranges of hundred’s and volume is in the range of lakhs. Hence, to bring them to a similar range, log of the variables was the best suited option.
- Iteration 4 is the best suited model after Comparing the MSE and RMSE.

THE END