Project Name - **Yes Bank Stock Price Prediction**

Project Type - Regression In Machine Learning

--->By Krishna from cohort Jerusalem in Almabetter

Contribution - **Individual**

Dataset Link-- https://drive.google.com/file/d/1u8kSz309mrZULVPrxRvZ6VRNDj1dBfTM/view?usp=share_link

Github Link--

Introduction

YES bank stands for Youth Enterprise Scheme Bank. Stock market is one of the major fields that attracts people, thus stock market price prediction is always a hot topic for researchers from both financial and technical domains. In our project our objective is to build a prediction model for close price prediction. A stock market is a public market where you can buy and sell shares for publicly listed companies. Stock Price Prediction using machine learning helps you get an estimate of value of company stock going forward and other financial assets traded on an exchange. The entire idea of predicting stock prices is to gain significant profits. Predicting how the stock market will perform is a hard task to do. There are numerous other factors involved in the prediction, such as the psychological factor – namely crowd behavior etc. All these factors combine to make share prices very difficult to predict with high accuracy.

PROBLEM STATEMENT---

Yes Bank is a well-known Indian bank headquartered in Mumbai, India and was founded by Rana Kapoor and Ashok Kapoor in 2004. It offers wide range of differentiated products for corporate and retail customers through retail banking and asset management services. Yes Bank is a publicly traded company listed on the stock market and is therefore subject to the ups and downs of the stock market cycle. The stock market is driven by speculation. The investors decide on buying or selling shares of a company based on its performance and its reputation. Public opinion has a huge impact on stock market prices. Which is why when the news of fraud case involving Rana Kapoor broke in 2018, stock price of Yes bank went down significantly. Here we are presented with the stock market price data of Yes bank and our job is to try and predict stock's closing price of the month. This data contains the date, lowest, highest and closing price details. Our approach is to fit a machine learning model on this past data and try to predict the closing price for new unseen data using the parameters learned during training. This way, we can get our model to learn the trends present in the data during training and use that information during prediction. We will apply various Regression Models for this task such as : Linear Regression, Lasso Regression, Ridge Regression, Elastic Net Regression.

Loading the libraries and the data--

```
# importing the libraries we'll need.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')

# importing LinearRegression model and the metrics that we will use for evaluating differe
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error

# Importing Lasso model.
from sklearn.linear_model import Lasso

# importing ridge regressor model.
from sklearn.linear_model import Ridge
ridge = Ridge()

# importing and initializing Elastic-Net Regression.
from sklearn.linear_model import ElasticNet
```

```
# Mounting google drive to load the data.
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# Loading our dataset.
df = pd.read_csv('/content/drive/MyDrive/csvfile/data_YesBank_StockPrices.csv')
df
```

	Date	Open	High	Low	Close
0	Jul-05	13.00	14.00	11.25	12.46
1	Aug-05	12.58	14.88	12.55	13.42
2	Sep-05	13.48	14.87	12.27	13.30
3	Oct-05	13.20	14.47	12.40	12.99
4	Nov-05	13.35	13.88	12.88	13.41
...	...	...	...	...	...
180	Jul-20	25.60	28.30	11.10	11.95
181	Aug-20	12.00	17.16	11.85	14.37
182	Sep-20	14.30	15.34	12.75	13.15
183	Oct-20	13.30	14.01	12.11	12.42
184	Nov-20	12.41	14.90	12.21	14.67

```
# Taking a look at the data.
df.head()          # displays first five instances of the dataframe.
```

	Date	Open	High	Low	Close
0	Jul-05	13.00	14.00	11.25	12.46
1	Aug-05	12.58	14.88	12.55	13.42
2	Sep-05	13.48	14.87	12.27	13.30
3	Oct-05	13.20	14.47	12.40	12.99
4	Nov-05	13.35	13.88	12.88	13.41

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 185 entries, 0 to 184
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Date    185 non-null    object 
 1   Open    185 non-null    float64
 2   High    185 non-null    float64
 3   Low     185 non-null    float64
 4   Close   185 non-null    float64
dtypes: float64(4), object(1)
memory usage: 7.4+ KB
```

Explaining the data:- We have a dataset containing values of Yes bank monthly stock prices as mentioned in our problem statement.

Explaining the features present :-

Date :- The date (Month and Year provided)

Open :- The price of the stock at the beginning of a particular time period.

High :-The Peak(Maximum) price at which a stock traded during the period.

Low :-The Lowest price at which a stock traded during the period.

Close :- The trading price at the end (in this case end of the month).

```
df.describe()
```

	Open	High	Low	Close
count	185.000000	185.000000	185.000000	185.000000
mean	105.541405	116.104324	94.947838	105.204703
std	98.879850	106.333497	91.219415	98.583153
min	10.000000	11.240000	5.550000	9.980000
25%	33.800000	36.140000	28.510000	33.450000
50%	62.980000	72.550000	58.000000	62.540000
75%	153.000000	169.190000	138.350000	153.300000
max	369.950000	404.000000	345.500000	367.900000

Data Cleaning--

```
# Checking for null values.
df.isna().sum()
```

```
Date     0
Open     0
High     0
Low      0
Close    0
dtype: int64
```

```
# So there are no null values in our dataset.
# Getting information about our data - its datatypes, its size etc. also printing the shap
df.info()
print('\n', f'The shape of the dataset is : {df.shape}')
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 185 entries, 0 to 184
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Date    185 non-null    object 
 1   Open    185 non-null    float64
```

```
 2   High    185 non-null    float64
 3   Low     185 non-null    float64
 4   Close   185 non-null    float64
dtypes: float64(4), object(1)
memory usage: 7.4+ KB

 The shape of the dataset is : (185, 5)
```

```
# getting descriptive statistics of the data.
df.describe(include='all')
```

	Date	Open	High	Low	Close
count	185	185.000000	185.000000	185.000000	185.000000
unique	185	NaN	NaN	NaN	NaN
top	Jul-05	NaN	NaN	NaN	NaN
freq	1	NaN	NaN	NaN	NaN
mean	NaN	105.541405	116.104324	94.947838	105.204703
std	NaN	98.879850	106.333497	91.219415	98.583153
min	NaN	10.000000	11.240000	5.550000	9.980000
25%	NaN	33.800000	36.140000	28.510000	33.450000
50%	NaN	62.980000	72.550000	58.000000	62.540000
75%	NaN	153.000000	169.190000	138.350000	153.300000
max	NaN	369.950000	404.000000	345.500000	367.900000

```
# Let us now preserve the original data before we operate on it.
preserved_stock_data = df.copy()
```

```
# Checking for duplicate instances.
df[df.duplicated()==True]
```

Date	Open	High	Low	Close

```
# So there is no duplicate data in our dataframe.
# checking the datatypes once more.
df.dtypes
```

```
Date      object
Open     float64
High     float64
Low      float64
Close    float64
dtype: object
```

```
# as we can see, Date column has the object datatype.
df['Date']
```

```
0      Jul-05
1      Aug-05
2      Sep-05
3      Oct-05
4      Nov-05
        ...
180    Jul-20
181    Aug-20
182    Sep-20
183    Oct-20
184    Nov-20
Name: Date, Length: 185, dtype: object
```

```
# we need to modify this before passing it to a model.
# lets convert Date column to a proper datetime datatype.
from datetime import datetime
df['Date'] = pd.to_datetime(df['Date'].apply(lambda x: datetime.strptime(x, '%b-%y')))
```

```
df.head()
```

	Date	Open	High	Low	Close
0	2005-07-01	13.00	14.00	11.25	12.46
1	2005-08-01	12.58	14.88	12.55	13.42
2	2005-09-01	13.48	14.87	12.27	13.30
3	2005-10-01	13.20	14.47	12.40	12.99
4	2005-11-01	13.35	13.88	12.88	13.41

Since we are trying to track variation in stock price on different dates, it makes sense to set this column as index.

```
df.set_index('Date', inplace=True)           # setting Date column as index.
```

```
# checking the data.
df.head()
```

	Open	High	Low	Close

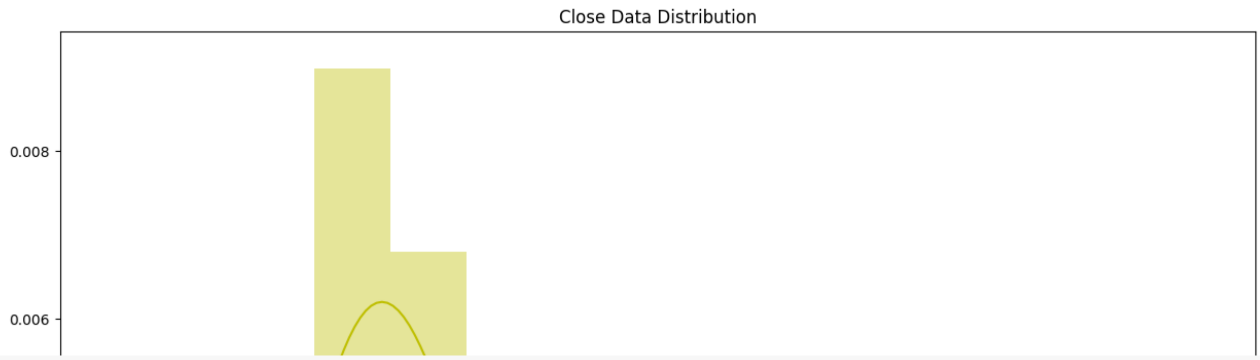
We can see from the dataframe above, all the columns we have contain numerical data. There is no categorical data present.

2005-08-01	12.58	14.88	12.55	13.42

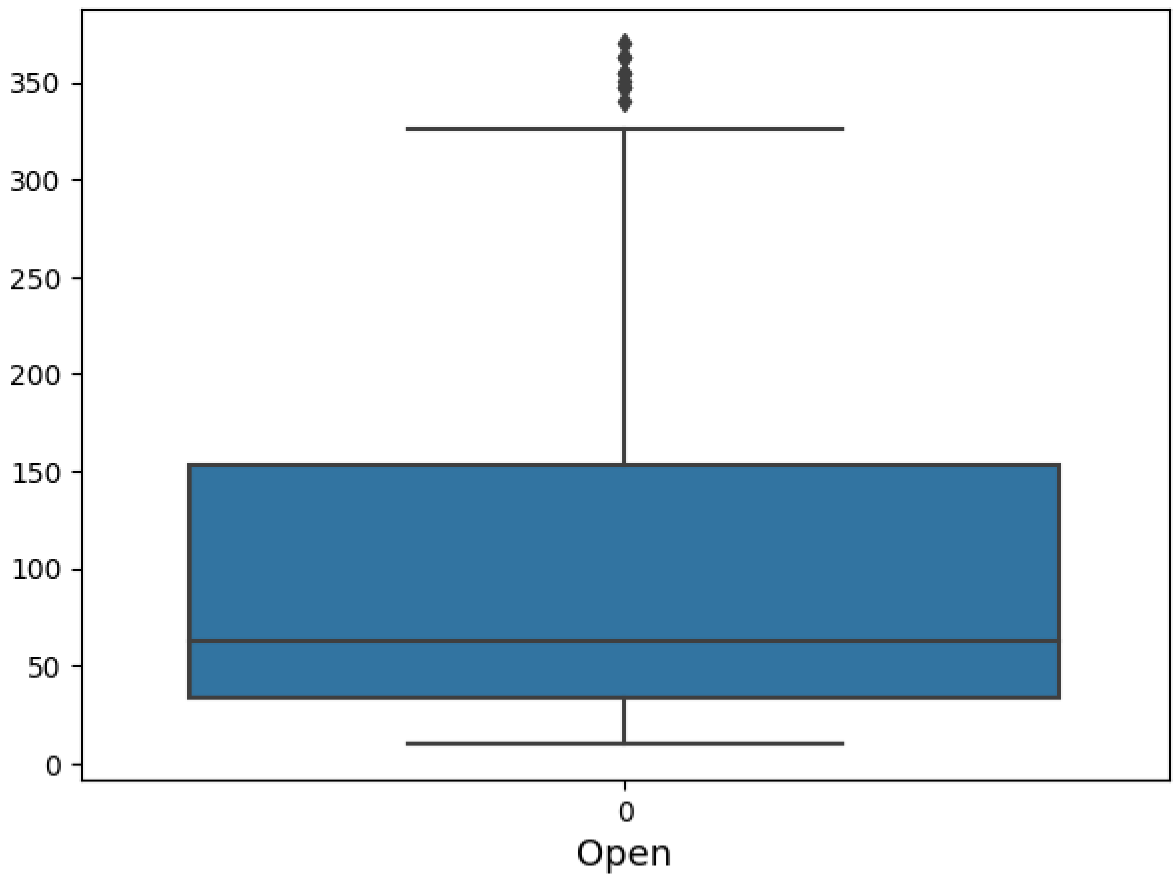
Data Visulazation--

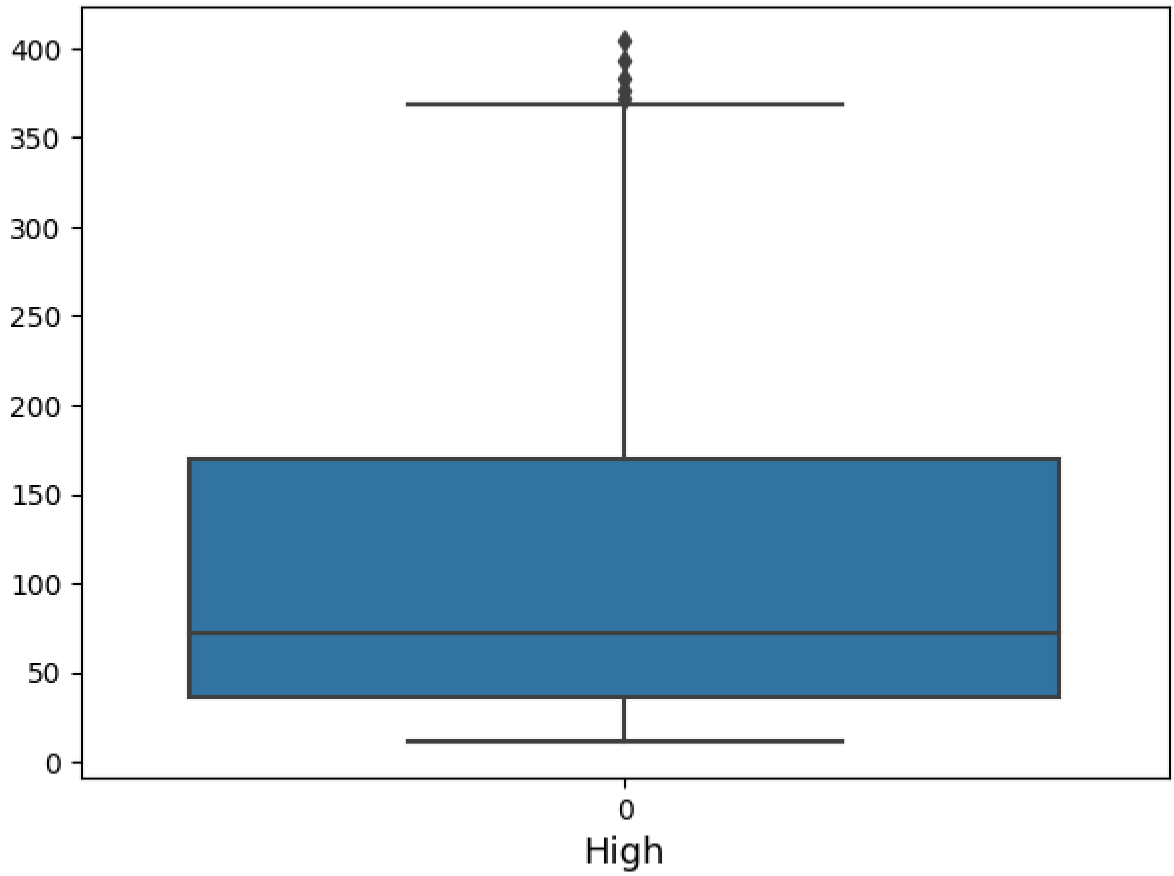
2005-11-01	13.35	13.88	12.88	13.41

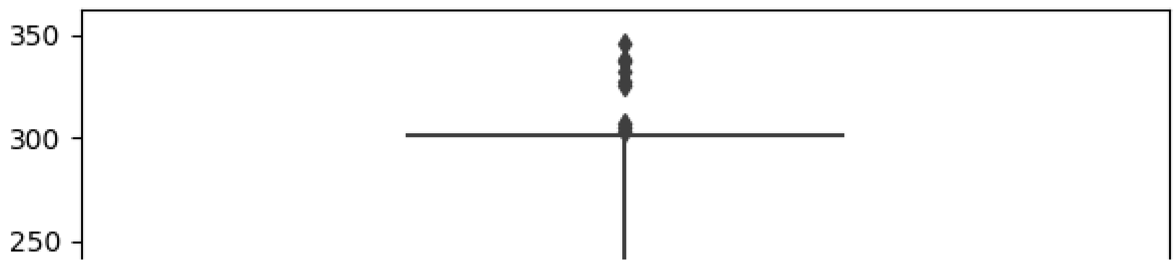
```
# Dependent variable 'Closing price'
plt.figure(figsize=(15,10))
sns.distplot(df['Close'],color="y")
plt.title('Close Data Distribution')
plt.xlabel('Closing Price')
plt.show()
```




```
# Checking all features for presence of outliers.
for col in df.columns:
  plt.figure(figsize=(7,5))
  sns.boxplot(df[col])
  plt.xlabel(col, fontsize=13)
  plt.show()
```


350
300
250
200
150
100
50
0
0
Open


400
350
300
250
200
150
100
50
0
0
High


350
300
250

As we can see there are some outliers present in our data. We will need to deal with these before proceeding to modelling.

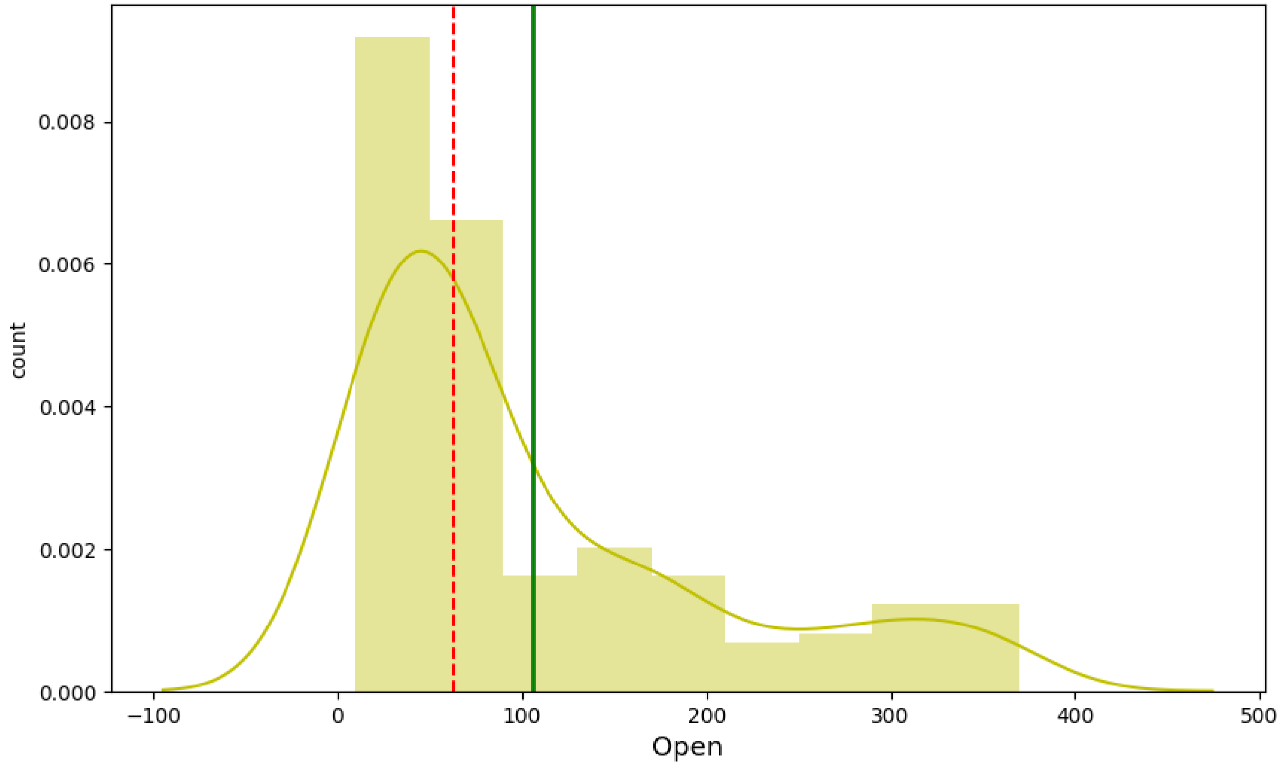
```
# Separating the dependent and independent variables.
independent_variables = df.columns.tolist()[:-1]
dependent_variable = ['Close']

print(independent_variables)
print(dependent_variable)
```

```
['Open', 'High', 'Low']
['Close']
```

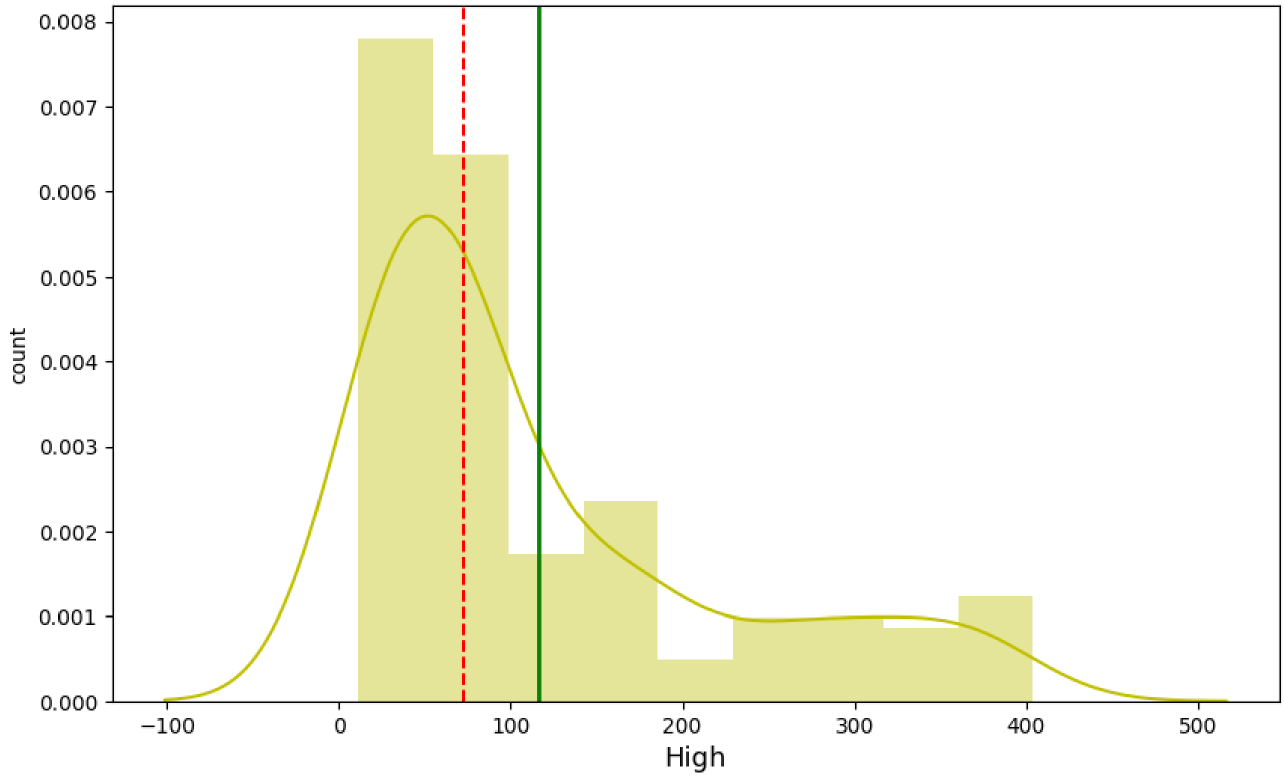
```
# Plotting the dependent variable .
plt.figure(figsize=(12,7))
df['Close'].plot(color = 'r')
plt.grid(which='major', linestyle='-', linewidth='0.5', color='green')
plt.grid(which='minor', linestyle=':', linewidth='0.5', color='green')
plt.xlabel('Date')
plt.ylabel('Closing Price')
plt.title('Closing Price with Date')
plt.show()
```

We can see that the stock price is rising up until 2018 when the fraud case involving Rana Kapoor happened after which the stock price has had a sharp decline.

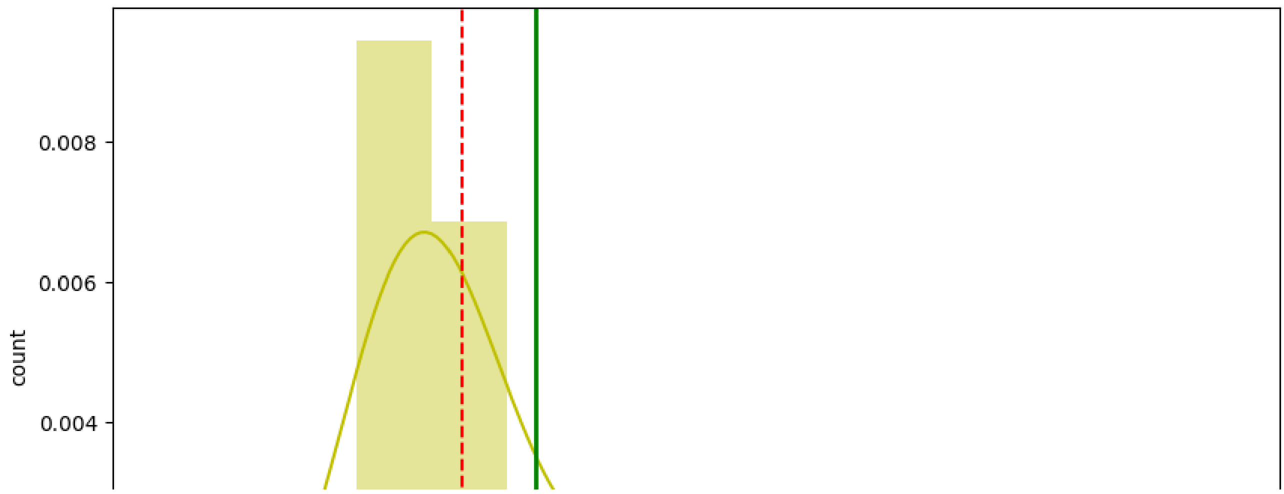
```
# Plotting the distributions of all features.
for col in df.columns:
  plt.figure(figsize=(10,6))
  sns.distplot(df[col], color='y')
  plt.xlabel(col, fontsize=13)
  plt.ylabel('count')

  # Plotting the mean and the median.
  plt.axvline(df[col].mean(),color='green',linewidth=2)                           # axvli
  plt.axvline(df[col].median(),color='red',linestyle='dashed',linewidth=1.5)
  plt.show()
```



count
0.000
0.002
0.004
0.006
0.008
−100
0
100
200
300
400
500
Open



count
0.000
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008
−100
0
100
200
300
400
500
High

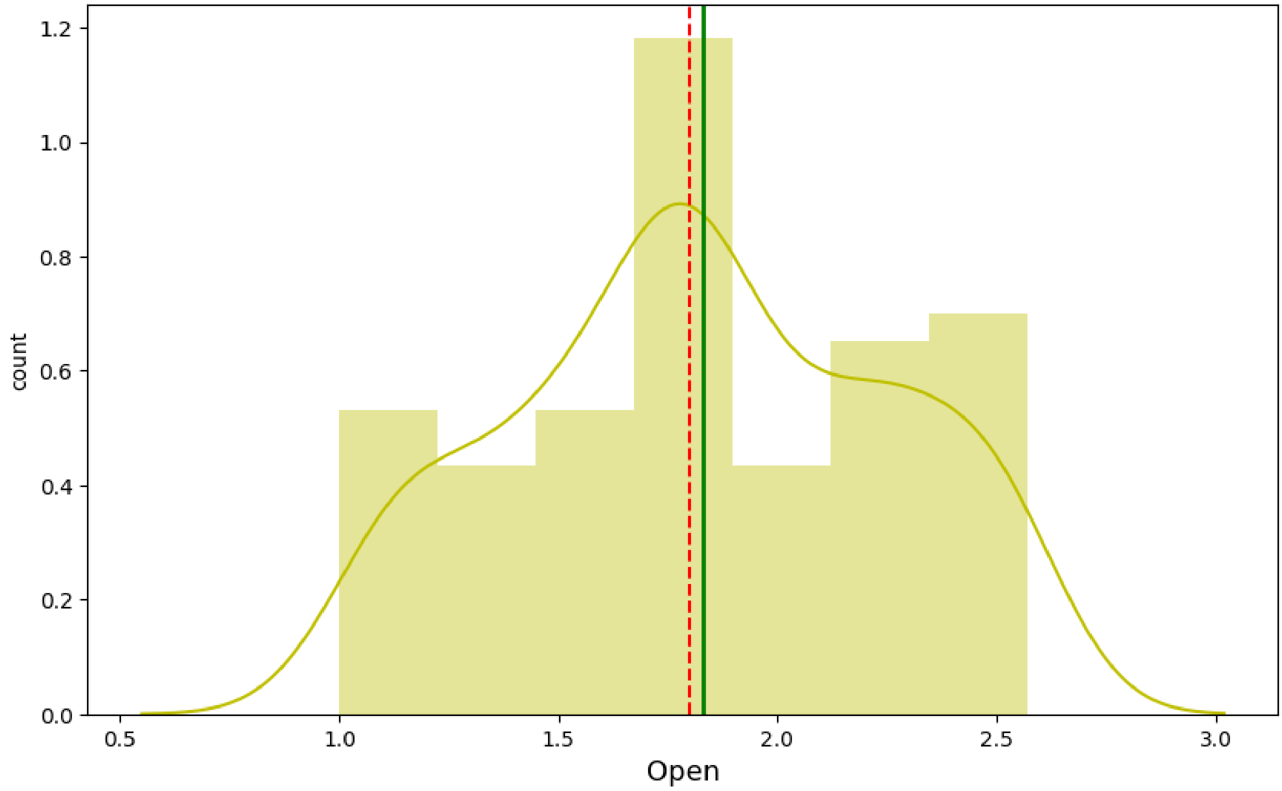


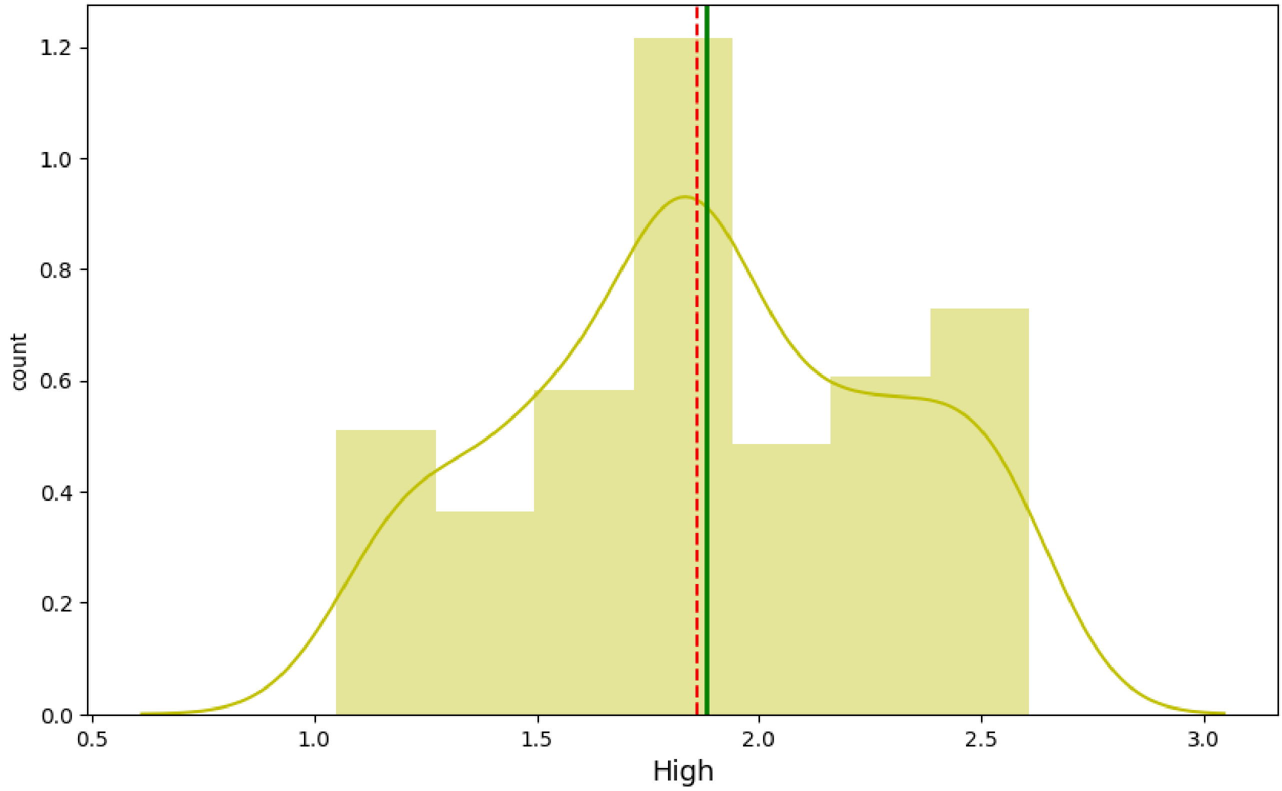
count
0.004
0.006
0.008

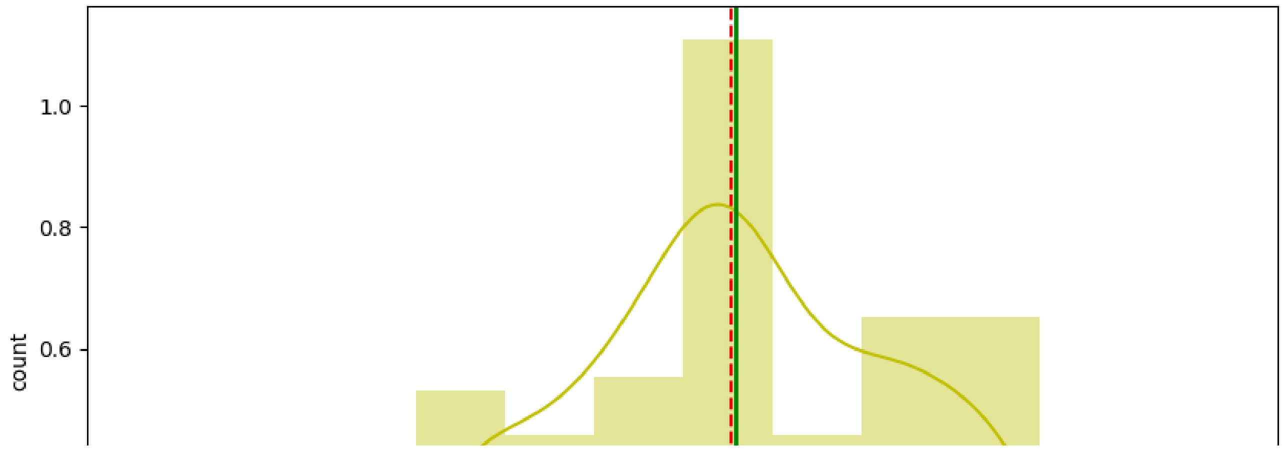
We can clearly see that ***these distributions are positively skewed***. The mean and median are at significant distance from each other.

So we need to transform them into something close to a Normal Distribution as our models give optimal results that way.

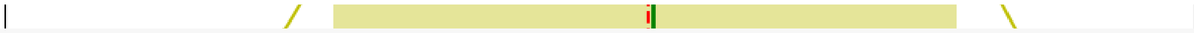
```
# Lets use log transformation on these features using np.log() and plot them.
for col in df.columns:
  plt.figure(figsize=(10,6))
  sns.distplot(np.log10(df[col]), color='y')
  plt.xlabel(col, fontsize=13)
  plt.ylabel('count')

  # Plotting the mean and the median.
  plt.axvline(np.log10(df[col]).mean(),color='green',linewidth=2)
  plt.axvline(np.log10(df[col]).median(),color='red',linestyle='dashed',linewidth=1.5)
  plt.show()
```


1.2
1.0
0.8
0.6
0.4
0.2
0.0
count
0.5
1.0
1.5
2.0
2.5
3.0
Open

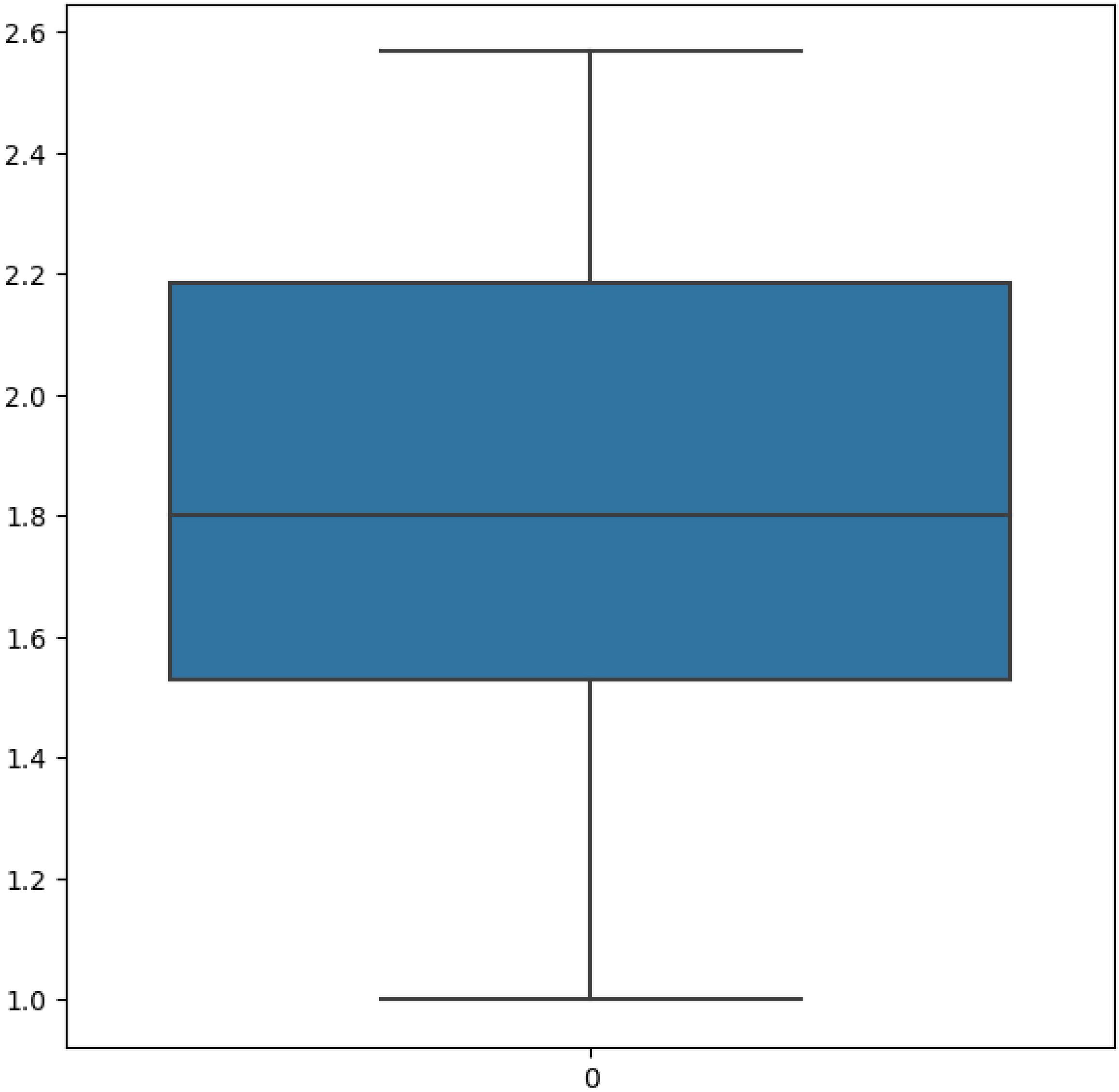

1.2
1.0
0.8
0.6
0.4
0.2
0.0
count
0.5
1.0
1.5
2.0
2.5
3.0
High


1.0
0.8
0.6
count

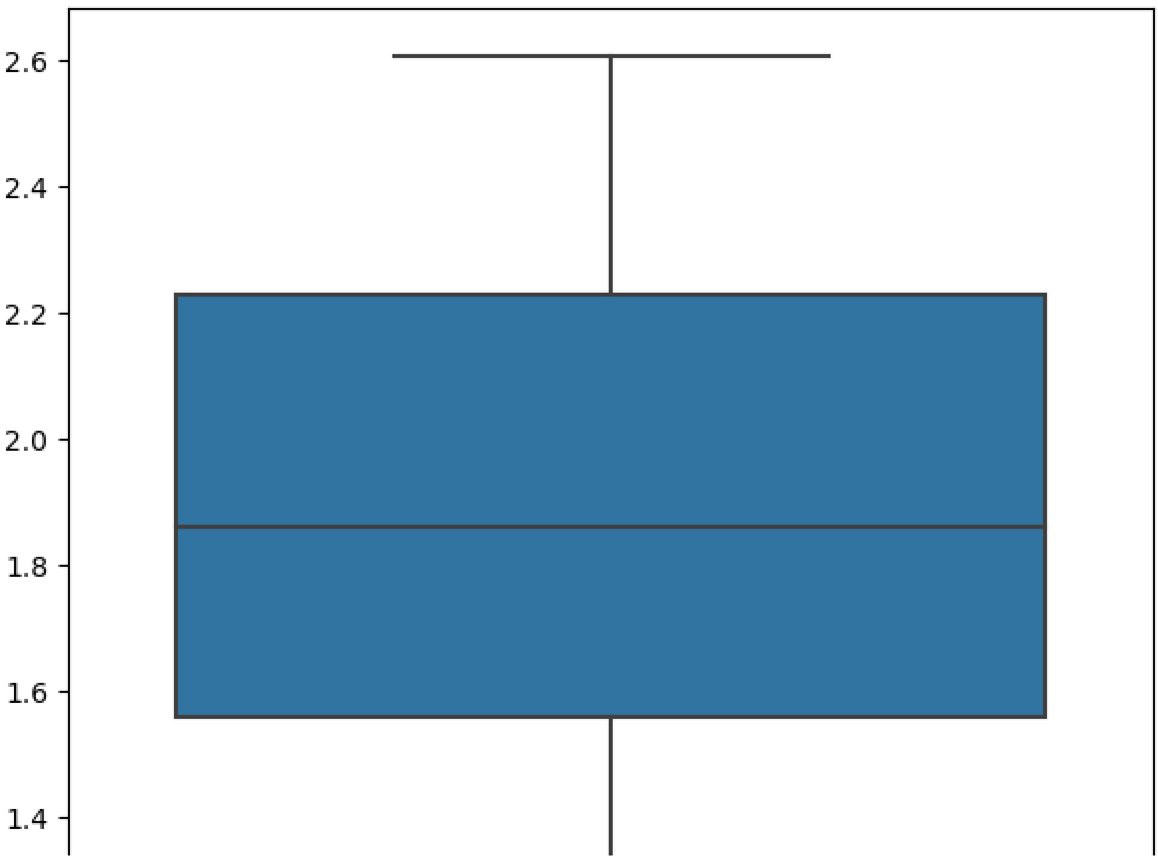
Now, the distributions are **very similar to Normal distribution**. The mean and median values are nearly same.*



```
# Let's check for outliers now in the transformed variable data.
for col in df.columns:
  plt.figure(figsize=(7,7))
  sns.boxplot(np.log10(df[col]))
  plt.xlabel(col, fontsize=13)
  plt.show()
```



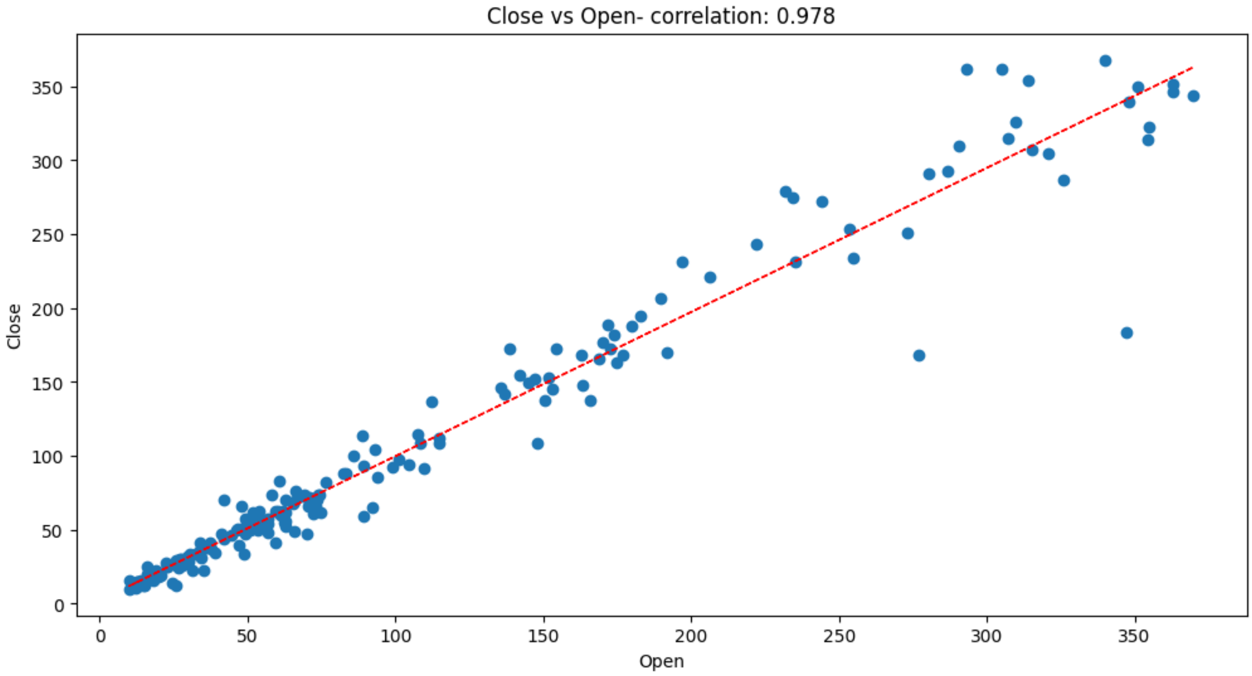
Open

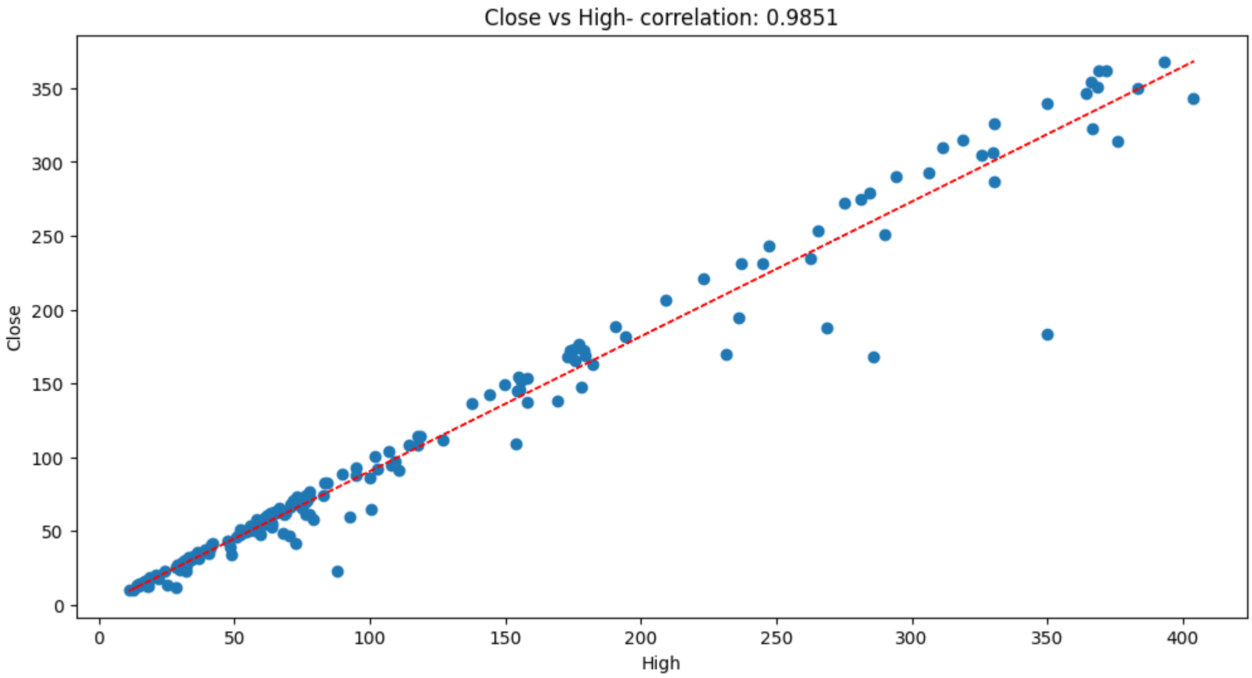


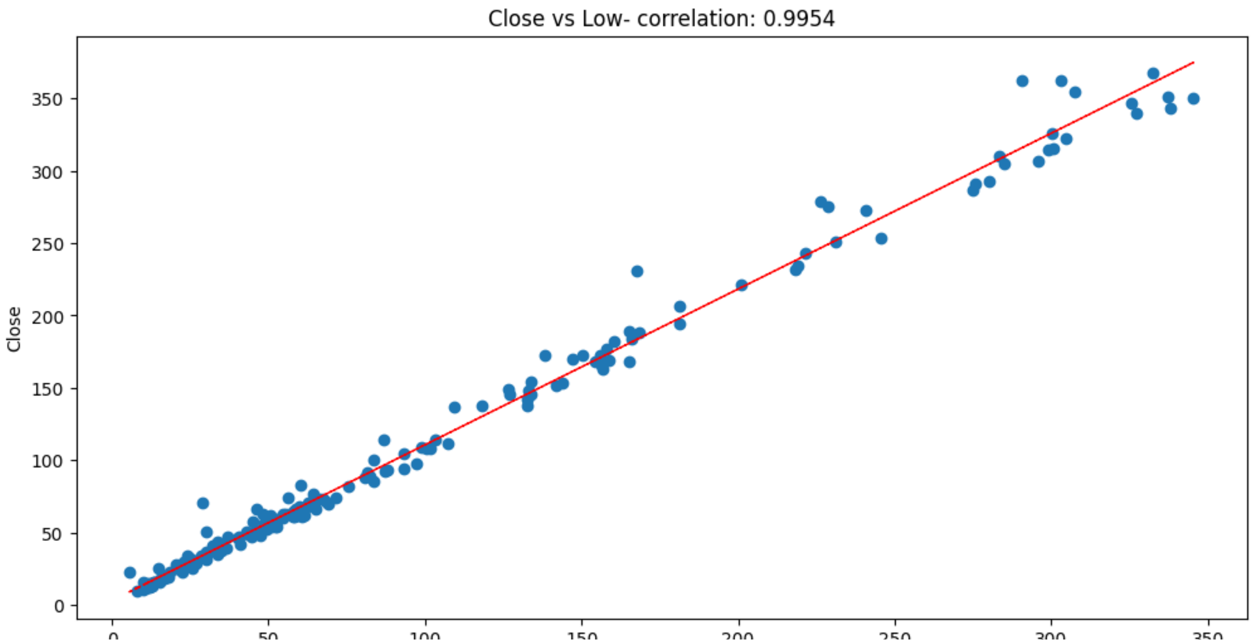
Now, we have no outliers anymore. Log transformation diminishes the outlier's effect.

Since we have a very small dataset to work with, dropping the outliers completely is not a good idea. So this is how we are going to leave them.

```
# Plotting the independent variables against dependent variable close and also checking th
for col in independent_variables:

  fig = plt.figure(figsize=(12, 6))
  ax = fig.gca()
  feature = df[col]
  label = df['Close']
  correlation = feature.corr(label)        # calculating the correlation between dependent
  plt.scatter(x=feature, y=label)          # plotting dependent variables against independ

  # Setting the x,y labels and the title.
  plt.xlabel(col)
  plt.ylabel('Close')
  ax.set_title('Close vs ' + col + '- correlation: ' + str(round((correlation),4)))

  z = np.polyfit(df[col], df['Close'], 1)
  y_hat = np.poly1d(z)(df[col])

  plt.plot(df[col], y_hat, "r--", lw=1)

plt.show()
```


Close vs Open- correlation: 0.978
Close
Open

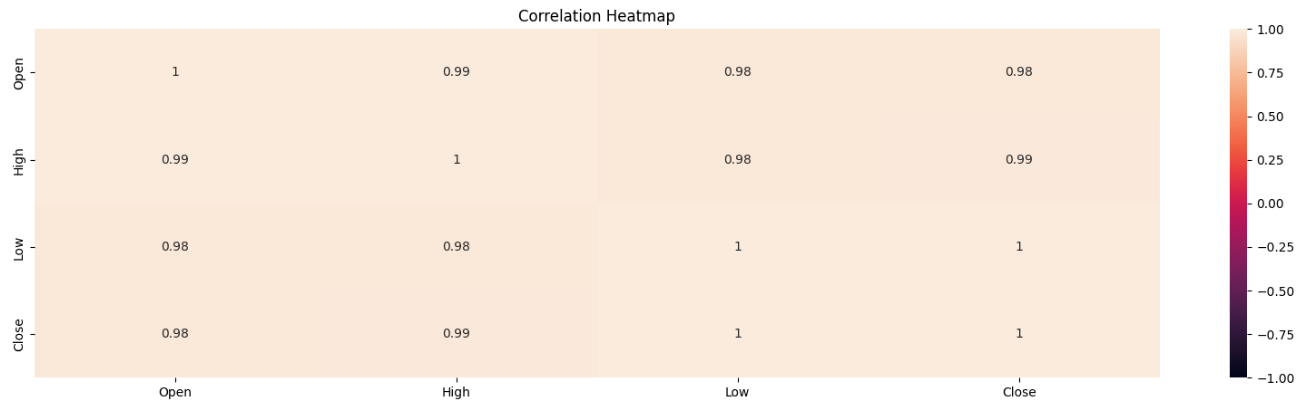

Close vs High- correlation: 0.9851
Close
High


Close vs Low- correlation: 0.9954
Close

We can see that all of our independent variables are highly correlated to the dependent variable.

And the relationship between dependent and independent variables is linear in nature.

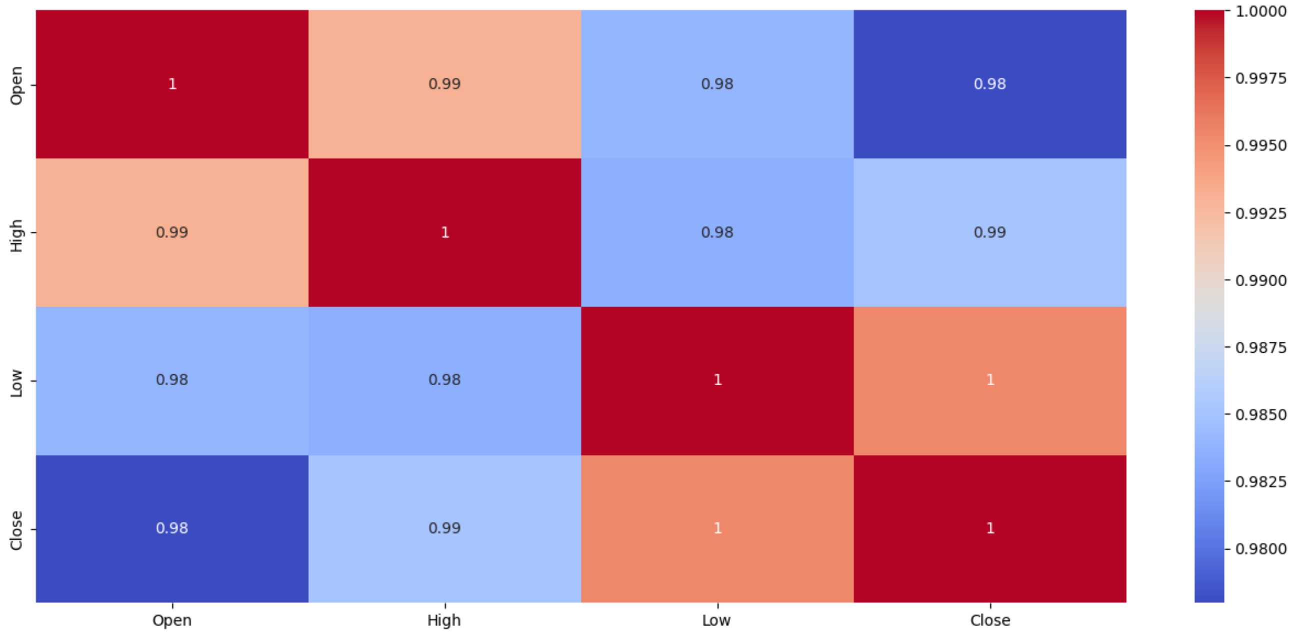
```
# check for existence of corelation
plt.figure(figsize=(20,5))
plt.title('Correlation Heatmap')
cor = sns.heatmap(df.corr(), vmin=-1, vmax=1, cmap=None, annot=True )
```



Every feature is extremely corelated with each other, so taking just one feature or average of these

```
# Now let's visualise for the correlation among all variables.
corr = df.corr()
plt.figure(figsize=(16,7))
sns.heatmap(corr, annot=True, cmap='coolwarm')
```

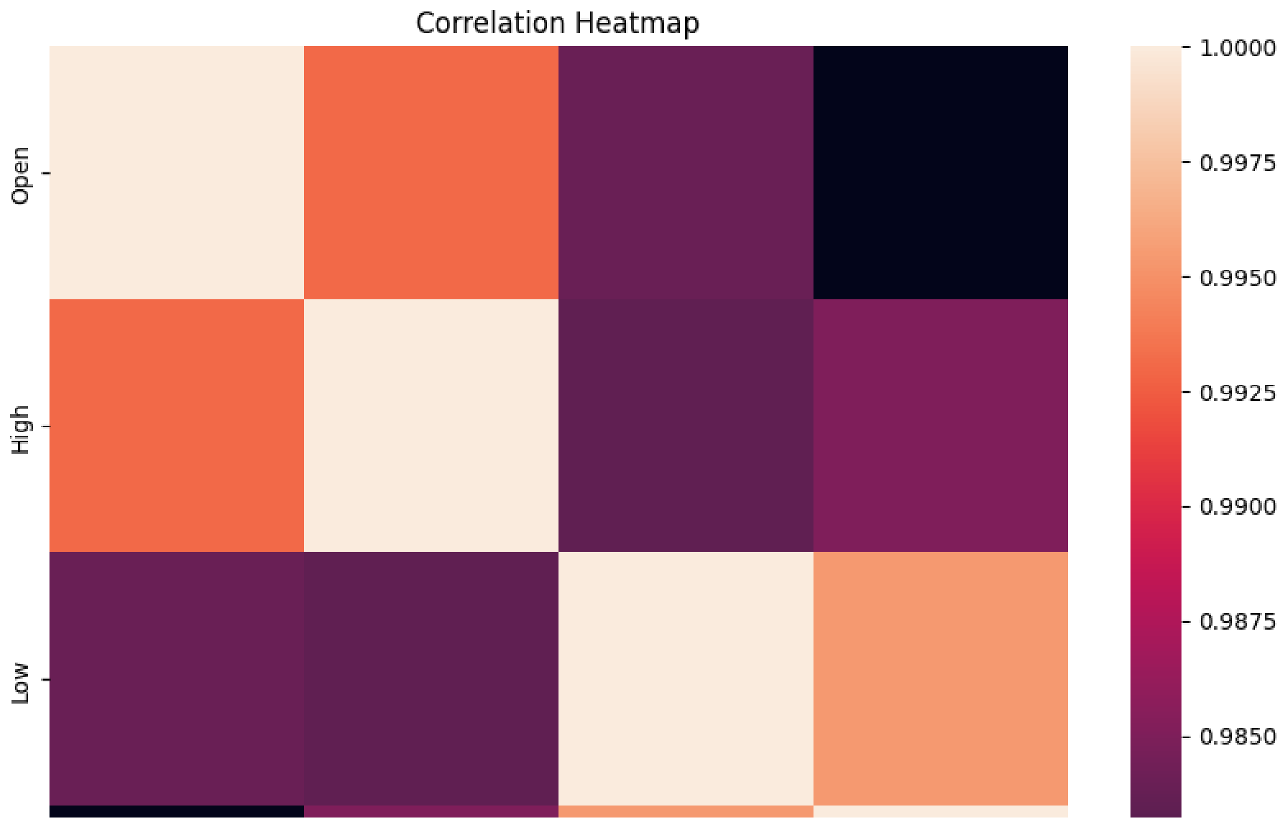
<Axes: >



From the heatmap above, we can clearly see that there is a very high correlation between each pair of features in our dataset. While it is desirable for the dependent variable to be highly correlated with independent variables, the independent varibles should ideally not have high correlation with one another.

This causes a problem for us as high correlation among independent variables (multicollinearity) is a problem for our models.

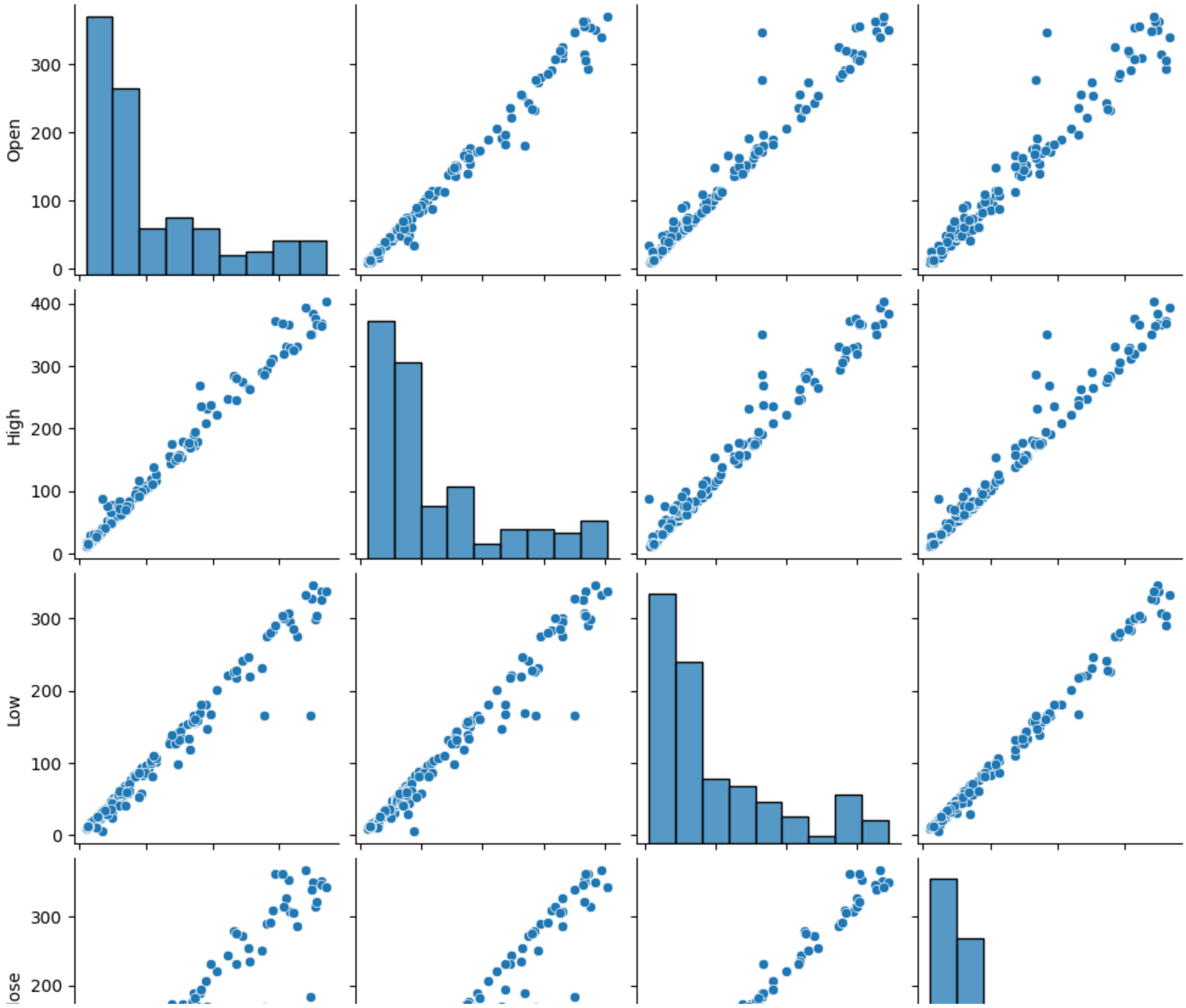
```
# correlation between features
plt.figure(figsize=(10, 8))
sns.heatmap(df.corr())
plt.title('Correlation Heatmap')
plt.show()
```



To reduce multicollinearity we can use regularization that means to keep all the features but reducing the magnitude of the coefficients of the model. This is a good solution when each predictor contributes to predict the dependent variable.

```
# Let's visualise the relationship between each pair of variables using pair plots.
sns.pairplot(df)
```

```
<seaborn.axisgrid.PairGrid at 0x7f20ad7ace50>
```



Data Preprocessing--

```
# Dealing with multicollinearity using VIF analysis.
# Calculating VIF(Variation Inflation Factor) to see the correlation between independent v

from statsmodels.stats.outliers_influence import variance_inflation_factor

def calc_vif(X):

    # Calculating VIF
    vif = pd.DataFrame()
    vif["variables"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]

    return(vif)
```

```
calc_vif(df[[i for i in df.describe().columns if i not in ['Date','Close']]])
```

	variables	VIF
0	Open	175.185704

As we can see the values of VIF factor are very high. However since the dataset is so small and has just 3 independent features, multicollinearity is unavoidable here as any feature engineering will lead to loss of information.

```
# Creating arrays of our input variable and label to feed the data to the model.
# Create the data of independent variables
x = np.log10(df[independent_variables]).values          # applying log transform on our

# Create the dependent variable data
y = np.log10(df[dependent_variable]).values             # applying log transform on our
```

```
# splitting the data into a train and a test set. we do this using train test split.
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state =
```

Scaling the data is very important for us so as to avoid giving more importance to features with large values. This is achieved by normalization or standardization of the data.

```
# Scaling the data.
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
```

```
# checking the values.
x_train[0:10]
```

```
array([[ 0.83311596,  0.8243388 ,  0.88445745],
       [-1.41735108, -1.31675483, -1.23862182],
       [ 0.3871812 ,  0.35973888,  0.04241403],
       [-0.06900104,  0.01215654, -0.30051561],
       [-1.91321118, -1.50865163, -1.71568543],
       [-0.2660071 ,  0.10246554, -0.21069831],
       [-0.29592654, -0.34290717, -0.15641974],
       [-0.59033534, -0.59737272, -0.45688014],
       [-0.24949754, -0.27329508, -0.60357017],
       [-0.94310352, -0.99502356, -1.60535529]])
```

1. Linear Regression

```
# importing LinearRegression model and the metrics that we will use for evaluating differe
from sklearn.linear_model import LinearRegression
```

```
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
```

```
# Initializing the model.
model_lr = LinearRegression()

# Fitting the model on our train data.
model_lr.fit(x_train, y_train)
```

```
▾ LinearRegression
LinearRegression()
```

```
# Predicting on our test data.
y_pred_linear = model_lr.predict(x_test)
```

```
# Checking the model parameters. printing the intercept.
model_lr.intercept_
```

```
array([1.79986471])
```

```
# printing the model coefficients.
model_lr.coef_
```

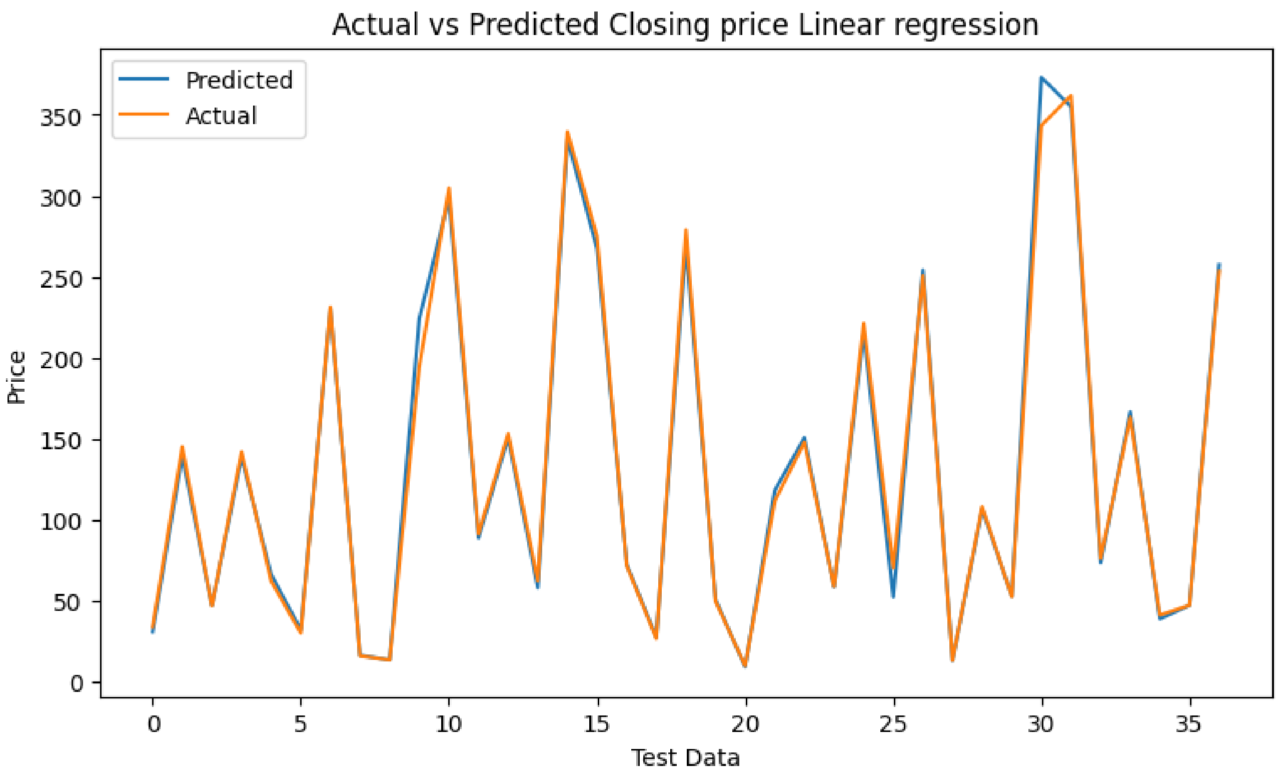
```
array([[-0.22992597,  0.33533242,  0.31585415]])
```

```
# Calculating the performance metrics.
MAE_linear = round(mean_absolute_error(10**(y_test),(10**y_pred_linear)),4)
print(f"Mean Absolute Error : {MAE_linear}")

MSE_linear = round(mean_squared_error((10**y_test),10**(y_pred_linear)),4)
print(f"Mean squared Error : {MSE_linear}")

RMSE_linear = round(np.sqrt(MSE_linear),4)
print(f"Root Mean squared Error : {RMSE_linear}")

R2_linear = round(r2_score(10**(y_test), 10**(y_pred_linear)),4)
print(f"R2 score : {R2_linear}")

Adjusted_R2_linear = round(1-(1-r2_score(10**y_test,10**y_pred_linear))*((x_test.shape[0]-
print(f"Adjusted R2 score : {Adjusted_R2_linear}")
```

```
Mean Absolute Error : 4.8168
Mean squared Error : 70.4204
Root Mean squared Error : 8.3917
R2 score : 0.9937
Adjusted R2 score : 0.993
```

```
# Plotting the actual and predicted test data.
plt.figure(figsize=(9,5))
```

```
plt.plot(10**y_pred_linear)
plt.plot(np.array(10**y_test))
plt.legend(["Predicted","Actual"])
plt.xlabel('Test Data')
plt.ylabel("Price")
plt.title("Actual vs Predicted Closing price Linear regression")
plt.show()
```



Now we need to store our performance data for this model so that we can compare them with other models. Let's store them in a dict for now.

```
linear_regessor_list = {'Mean Absolute Error' : MAE_linear,'Mean squared Error' : MSE_line
                    'Root Mean squared Error' : RMSE_linear,'R2 score' : R2_linear,'Adjuste
```

```
# converting above dict into a dataframe
metric_df = pd.DataFrame.from_dict(linear_regessor_list, orient='index').reset_index()
```

```
# renaming the columns.
metric_df = metric_df.rename(columns={'index':'Metric',0:'Linear Regression'})
metric_df
```

	Metric	Linear Regression
0	Mean Absolute Error	4.8168
1	Mean squared Error	70.4204

We will now use this df to store all metrics of all other models so we can easily compare them.

2. Lasso Regression with cross validated regularization

```
# Importing Lasso model.
from sklearn.linear_model import Lasso
```

```
# Initializing the model with some base values.
lasso  = Lasso(alpha=0.0001 , max_iter= 3000)
# Fitting the model on our training data.
lasso.fit(x_train, y_train)
```

```
Lasso
Lasso(alpha=0.0001, max_iter=3000)
```

```
# Printing the intercept and coefficients.
lasso.intercept_
```

```
array([1.79986471])
```

```
lasso.coef_
```

```
array([-0.2079326 ,  0.319775  ,  0.30927158])
```

```
# Cross validation. optimizing our model by finding the best value of our hyperparameter.
from sklearn.model_selection import GridSearchCV

lasso_param_grid = {'alpha': [1e-15,1e-13,1e-10,1e-8,1e-5,1e-4,1e-3,0.005,0.006,0.007,0.01

lasso_regressor = GridSearchCV(lasso, lasso_param_grid, scoring='neg_mean_squared_error',
lasso_regressor.fit(x_train, y_train)
```

```
GridSearchCV
estimator: Lasso
Lasso
```

```
# getting the best parameter
lasso_regressor.best_params_          # after several iterations and trials, we get this v
```

```
{'alpha': 1e-05}
```

```
# getting the best score
lasso_regressor.best_score_
```

```
-0.0011530156671872803
```

```
# Predicting on the test dataset.
y_pred_lasso = lasso_regressor.predict(x_test)
print(y_pred_lasso)
```

```
[1.49138725 2.14480164 1.67440535 2.14228699 1.82187891 1.50772917
 2.36207529 1.21547491 1.13723019 2.35007689 2.4750589  1.94911733
 2.17805254 1.76496504 2.52500153 2.427082   1.86088626 1.44157089
 2.43007104 1.70654066 0.97170315 2.07286344 2.17847869 1.76889148
 2.33378329 1.71856753 2.40521703 1.1226477  2.02876294 1.72319367
 2.5717837  2.5499049  1.86710909 2.22199908 1.59040105 1.67512911
 2.41082202]
```

```
# checking the performance using evaluation metrics.
MAE_lasso = round(mean_absolute_error(10**(y_test),10**(y_pred_lasso)),4)
print(f"Mean Absolute Error : {MAE_lasso}")

MSE_lasso  = round(mean_squared_error(10**(y_test),10**(y_pred_lasso)),4)
print("Mean squared Error :" , MSE_lasso)

RMSE_lasso = round(np.sqrt(MSE_lasso),4)
print("Root Mean squared Error :" ,RMSE_lasso)

R2_lasso = round(r2_score(10**(y_test), 10**(y_pred_lasso)),4)
print("R2 score :" ,R2_lasso)

Adjusted_R2_lasso = round(1-(1-r2_score(10**y_test, 10**y_pred_lasso))*((x_test.shape[0]-1
print("Adjusted R2 score: ", Adjusted_R2_lasso)
```

```
Mean Absolute Error : 4.8262
Mean squared Error : 70.3311
Root Mean squared Error : 8.3864
R2 score : 0.9938
Adjusted R2 score:  0.9932
```

```
# Now saving these metrics to our metrics dataframe. First we save them in a list and then
metric_df['Lasso'] = [MAE_lasso, MSE_lasso, RMSE_lasso, R2_lasso, Adjusted_R2_lasso]
```

```
# plotting the predicted values vs actual.
plt.figure(figsize=(9,5))
plt.plot(10**y_pred_lasso)
plt.plot(np.array(10**y_test))
plt.legend(["Predicted","Actual"])
plt.ylabel("Price")
plt.title("Actual vs Predicted Closing price Lasso regression")
```

```
Text(0.5, 1.0, 'Actual vs Predicted Closing price Lasso regression')
```

3. Ridge Regression with cross validated regularization

```
# importing ridge regressor model.
from sklearn.linear_model import Ridge
ridge = Ridge()         # iitializing the model

# initiating the parameter grid for alpha (regularization strength).
ridge_param_grid = {'alpha': [1e-15,1e-10,1e-8,1e-5,1e-4,1e-3,1e-2,0.3,0.7,1,1.2,1.33,1.36

# cross validation.
ridge_regressor = GridSearchCV(ridge, ridge_param_grid, scoring='neg_mean_squared_error',
ridge_regressor.fit(x_train,y_train)
```

```
▸ GridSearchCV
▸ estimator: Ridge
    ▸ Ridge
```

```
# finding the best parameter value (for alpha)
ridge_regressor.best_params_
```

```
{'alpha': 0.01}
```

```
# getting the best score for optimal value of alpha.
ridge_regressor.best_score_
```

```
-0.001306921437493189
```

```
# predicting on the test dataset now.
y_pred_ridge = ridge_regressor.predict(x_test)
```

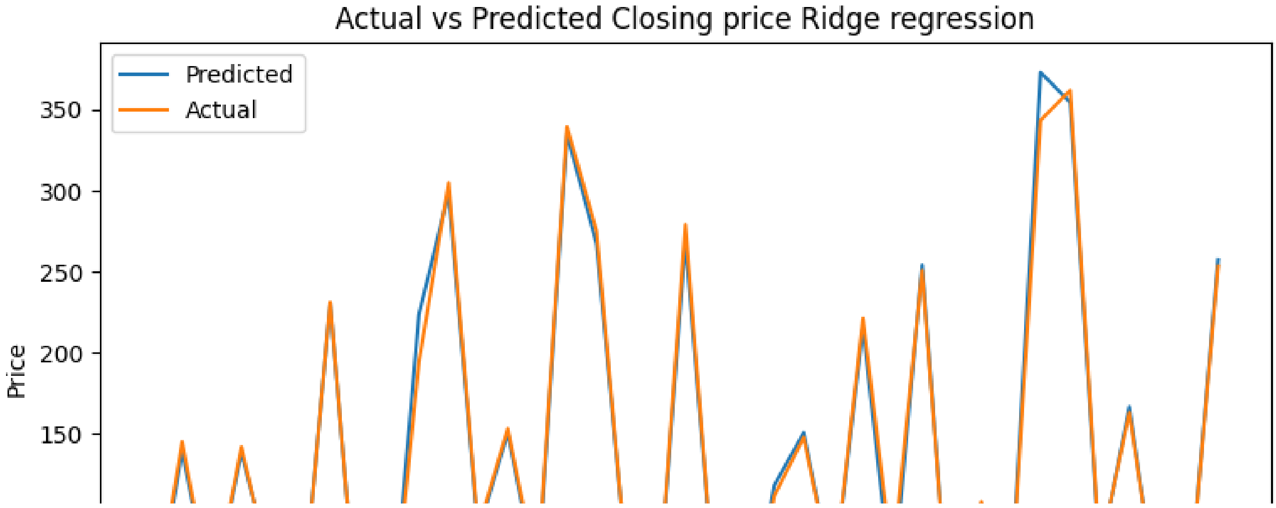
```
# evaluating performance.
MAE_ridge = round(mean_absolute_error(10**(y_test),10**(y_pred_ridge)),4)
print(f"Mean Absolute Error : {MAE_ridge}")

MSE_ridge  = round(mean_squared_error(10**(y_test),10**(y_pred_ridge)),4)
print("Mean squared Error :" , MSE_ridge)

RMSE_ridge = round(np.sqrt(MSE_ridge),4)
print("Root Mean squared Error :" ,RMSE_ridge)

R2_ridge = round(r2_score(10**(y_test), 10**(y_pred_ridge)),4)
print("R2 score :" ,R2_ridge)

Adjusted_R2_ridge = round(1-(1-r2_score(10**y_test, 10**y_pred_ridge))*((x_test.shape[0]-1
print("Adjusted R2 score: ", Adjusted_R2_ridge)
```

```
Mean Absolute Error : 4.8334
Mean squared Error : 70.2641
Root Mean squared Error : 8.3824
R2 score : 0.9938
Adjusted R2 score:  0.9932
```

```
# storing these values in a list and appending to our metric df.
ridge_regressor_list = [MAE_ridge,MSE_ridge,RMSE_ridge,R2_ridge,Adjusted_R2_ridge]
metric_df['Ridge'] = ridge_regressor_list
```

```
# Plotting predicted and actual target variable values.
plt.figure(figsize=(9,5))
plt.plot(10**y_pred_ridge)
plt.plot(np.array(10**y_test))
plt.legend(["Predicted","Actual"])
plt.ylabel("Price")
plt.title("Actual vs Predicted Closing price Ridge regression")
```

```
Text(0.5, 1.0, 'Actual vs Predicted Closing price Ridge regression')
```



4. Elastic-Net Regression with cross validation

```
# importing and initializing Elastic-Net Regression.
from sklearn.linear_model import ElasticNet
elasticnet_model = ElasticNet(alpha=0.1, l1_ratio=0.5)

# initializing parameter grid.
elastic_net_param_grid = {'alpha': [1e-15,1e-13,1e-10,1e-8,1e-5,1e-4,1e-3,0.001,0.01,0.02,
                          'l1_ratio':[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]}

# cross-validation.
elasticnet_regressor = GridSearchCV(elasticnet_model, elastic_net_param_grid, scoring='neg
elasticnet_regressor.fit(x_train, y_train)
```

```
▸ GridSearchCV
▸ estimator: ElasticNet
  ▸ ElasticNet
```

```
# finding the best parameter
elasticnet_regressor.best_params_
```

```
{'alpha': 0.0001, 'l1_ratio': 0.1}
```

```
# finding the best score for the optimal parameter.
elasticnet_regressor.best_score_
```

```
-0.0011528695836730079
```

```
# making the predictions.
y_pred_elastic_net = elasticnet_regressor.predict(x_test)
```

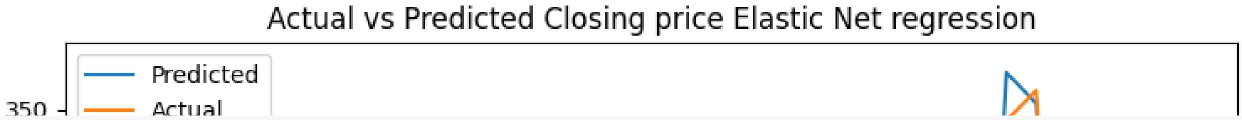
```
MAE_elastic_net = round(mean_absolute_error(10**(y_test),10**(y_pred_elastic_net)),4)
print(f"Mean Absolute Error : {MAE_elastic_net}")

MSE_elastic_net  = round(mean_squared_error(10**(y_test),10**(y_pred_elastic_net)),4)
print("Mean squared Error :" , MSE_elastic_net)

RMSE_elastic_net = round(np.sqrt(MSE_elastic_net),4)
print("Root Mean squared Error :" ,RMSE_elastic_net)

R2_elastic_net = round(r2_score(10**(y_test), (10**y_pred_elastic_net)),4)
print("R2 score :" ,R2_elastic_net)

Adjusted_R2_elastic_net = round(1-(1-r2_score(10**y_test, 10**y_pred_elastic_net))*((x_tes
print("Adjusted R2 score: ", Adjusted_R2_elastic_net)
```

```
Mean Absolute Error : 4.8483
Mean squared Error : 70.1569
Root Mean squared Error : 8.376
R2 score : 0.9938
Adjusted R2 score:  0.9932
```

```
# storing these metrics in our dataframe.
elastic_net_metric_list = [MAE_elastic_net,MSE_elastic_net,RMSE_elastic_net,R2_elastic_net
metric_df['Elastic Net'] = elastic_net_metric_list
```

```
# Now let us plot the actual and predicted target variables values.
plt.figure(figsize=(9,5))
plt.plot(10**y_pred_elastic_net)
plt.plot(np.array(10**y_test))
plt.legend(["Predicted","Actual"])
plt.ylabel("Price")
plt.title("Actual vs Predicted Closing price Elastic Net regression")
```

```
Text(0.5, 1.0, 'Actual vs Predicted Closing price Elastic Net regression')
```

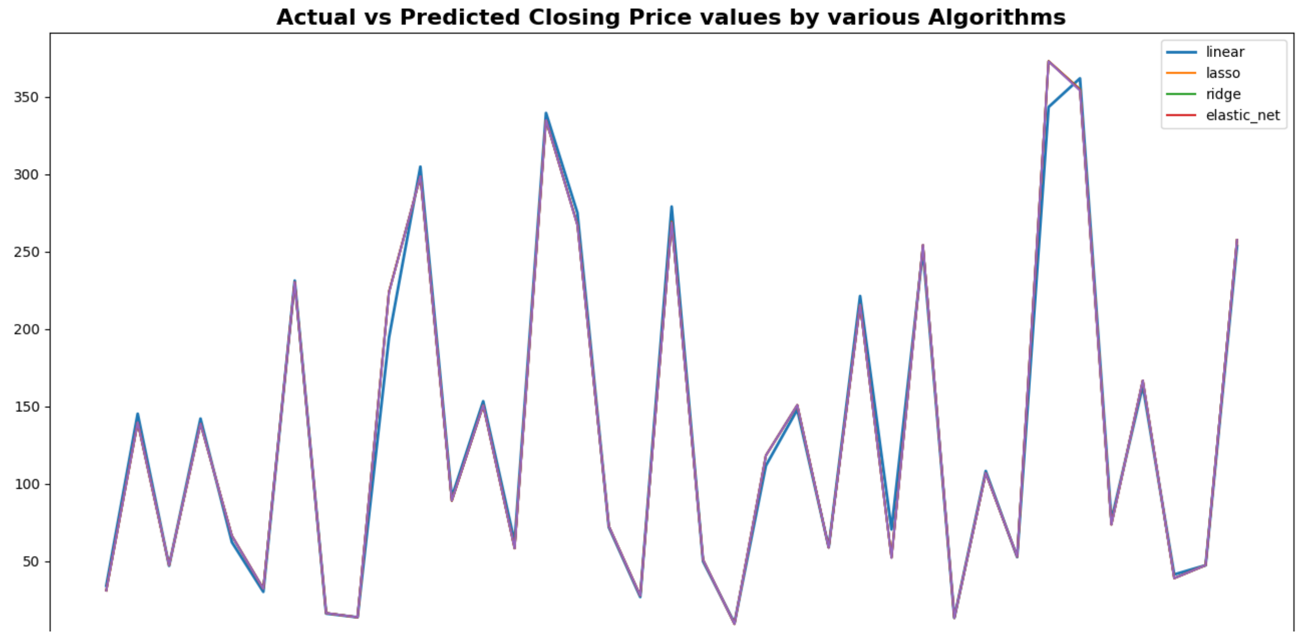



```
# comparing the performance of all models that we have implemented.
metric_df
```

	Metric	Linear Regression	Lasso	Ridge	Elastic Net
0	Mean Absolute Error	4.8168	4.8262	4.8334	4.8483
1	Mean squared Error	70.4204	70.3311	70.2641	70.1569
2	Root Mean squared Error	8.3917	8.3864	8.3824	8.3760
3	R2 score	0.9937	0.9938	0.9938	0.9938
4	Adjusted R2 score	0.9930	0.9932	0.9932	0.9932

From above data, we can clearly see that the best performing model is Elastic Net as it scores the best in every single metric.

```
# Plotting the predicted values of all the models against the true values.
plt.figure(figsize=(16,8))
plt.plot(10**y_test, linewidth=2)
plt.plot(10**y_pred_linear)
plt.plot(10**y_pred_lasso)
plt.plot(10**y_pred_ridge)
plt.plot(10**y_pred_elastic_net)
plt.legend(['linear','lasso','ridge','elastic_net'])
plt.title('Actual vs Predicted Closing Price values by various Algorithms', weight = 'bold
plt.show()
```



As we can see from above graph, all of our models are performing really well and are able to closely approximate the actual values.

```
# Lets check for Heterodasticity. Homoscedasticity is an assumption in linear regression a
# Homoscedasticity means that the model should perform well on all the datapoints.

# Plotting the residuals(errors) against actual test data.
residuals = 10**y_test - 10**y_pred_elastic_net.reshape(37,1)
plt.scatter(10**y_test,residuals,c='red')
plt.title('Actual Test data vs Residuals (Elastic Net)')
```

```
Text(0.5, 1.0, 'Actual Test data vs Residuals (Elastic Net)')
```