TCS Close price Forecasting Report

done by

Harsha R
(2nd year CSE)

and

Muthu Brijesh R
(2nd year CSE)

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

RAMCO INSTITUTE OF TECHNOLOGY,

RAJAPALAYAM

INTRODUCTION

The project is on Stock Market Prediction. Stock Price Prediction using machine learning is the process of predicting the future value of a stock traded on a stock exchange for reaping profits. With multiple factors involved in predicting stock prices, it is challenging to predict stock prices with high accuracy, and this is where machine learning plays a vital role. Using pandas to get stock information, visualize different aspects of it, and finally we will look at a few ways of analysing the risk of a stock, based on its previous performance history. We will also be predicting future stock prices through a Long Short-Term Memory (LSTM) method. The Given dataset contain information about stock market data. The dataset contains 13 columns. This is day data of stock market which give the information about each day trade. The day data range from 2004-2021. This is forecasting problem. The Goal of this problem is to predict the close price for the next 10 days.

PROJECT DESCRIPTION

Let us see the data on which we will be working before we begin implementing the software to anticipate stock market values. In this section, we will examine the stock price of Tata Consultancy Services (TCS). The stock price data will be supplied as a Comma Separated File (.csv), that may be opened and analysed in Excel or a Spreadsheet.

Date	Symbol	Series	Prev Close	Open	High	Low	Last	Close	VWAP	Volume	Tumover	Trades	Deliverable Volume	%Deliverble
2004-08-25	TCS	EQ	850	1198.7	1198.7	979	985	987.95	1008.32	17116372	1.72588E+15		5206360	
2004-08-26	TCS	EQ	987.95	992	997	975.3	976.85	979	985.65	5055400	498286476730000		1294899	0.
2004-08-27	TCS	EQ	979	982.4	982.4	958.55	961.2	962.65	969.94	3830750	371558603080000		976527	0.
2004-08-30	TCS	EQ	962.65	969.9	990	965	986.4	986.75	982.65	3058151	300510633990000		701664	0.
2004-08-31	TCS	EQ	986.75	986.5	990	976	987.8	988.1	982.18	2649332	260213265065000		695234	0.
2004-09-01	TCS	EQ	988.1	990	995	983.6	986	987.9	989.68	2491943	246623557694999		790586	0.
2004-09-02	TCS	EQ	987.9	989.9	1004.6	986	994	993.65	996.96	2669544	266142619640000		501792	0
2004-09-03	TCS	EQ	993.65	1006	1100	990.35	998.7	997.85	996.91	1233732	122991693990000		235508	0.
2004-09-06	TCS	EQ	997.85	1039.9	1039.9	992.9	996.8	994.85	998.87	1129834	112855395175000		430184	0.
2004-09-07	TCS	EQ	994.85	1035	1035	995	995	995.6	997.34	721529	71961091360000		198212	0.
2004-09-08	TCS	EQ	995.6	996	1000.8	991.1	992.25	993.7	995.29	824248	82036168250000		220195	0.
2004-09-09	TCS	EQ	993.7	997	997.9	978.45	979.65	979.95	985.16	993398	97865742605000		364189	0.
2004-09-10	TCS	EQ	979.95	990	990	976	989.95	988.8	985.12	801884	78995066425000		203388	0.
2004-09-13	TCS	EQ	988.8	991	1015.5	991	1002.75	1003.5	1007.31	2613114	263222057075000		735865	0.
2004-09-14	TCS	EQ	1003.5	1005	1018.8	1002.95	1015.95	1015.65	1012.95	1916934	194176341765000		493998	0.
2004-09-15	TCS	EQ	1015.65	1018	1020	1001.2	1005	1006.1	1009.12	1498536	151220835185000		483466	0.

TCS's stocks are listed and their value is updated every working day of the stock market. It should be noted that the market does not allow trading on Saturdays and Sundays, therefore there is a gap between the two dates. The Opening Value of the stock, the Highest and Lowest values of that stock on the same days, as well as the Closing Value at the end of the day, are all indicated for each date.

Exploratory Data Analysis:

The objectives of the EDA are as follows:

i. To get an overview of the distribution of the Data set.

ii. Check for missing numerical values, outliers or other anomalies in the Data set.

iii. Discover patterns and relationships between variables in the Data set.

iv. Check the underlying assumptions in the Data set.

Importing Data Set and Previewing the Data Set:

Importing the required Libraries

As we all know, the first step is to import the libraries required to Preprocess TCS stock data and the other libraries required for constructing and visualizing the ARIMA model and LSTM model outputs.

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dense, Dropout
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.preprocessing import MinMaxScaler,StandardScaler
import seaborn as sns
from datetime import datetime
```

```python
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.metrics import mean_absolute_error, mean_squared_error
import math
from math import sqrt

from statsmodels.tsa.seasonal import seasonal_decompose
from pandas.plotting import lag_plot
from pandas.plotting import autocorrelation_plot
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf

from pmdarima import auto_arima
from statsmodels.tsa.arima_model import ARIMA
```

Next, load the data and let’s take a look, while loading the dataset parse the date column to date data type in python.

Preview the dataset

```python
dateparse = lambda dates: pd.datetime.strptime(dates, '%Y-%m-%d')
df = pd.read_csv('C:/Users/Rshs/Documents/python notebooks/mldatasets/TCS.csv',sep=',', parse_dates=['Date'], date_parser=datepar
df.head()
```

```
C:\Users\Rshs\AppData\Local\Temp/ipykernel_10144/1666060968.py:1: FutureWarning: The pandas.datetime class is deprecated and wi
ll be removed from pandas in a future version. Import from datetime module instead.
  dateparse = lambda dates: pd.datetime.strptime(dates, '%Y-%m-%d')
```

	Date	Symbol	Series	Prev Close	Open	High	Low	Last	Close	VWAP	Volume	Turnover	Trades	Deliverable Volume	%Deliverble
0	2004-08-25	TCS	EQ	850.00	1198.7	1198.7	979.00	985.00	987.95	1008.32	17116372	1.725876e+15	NaN	5206360	0.3042
1	2004-08-26	TCS	EQ	987.95	992.0	997.0	975.30	976.85	979.00	985.65	5055400	4.982865e+14	NaN	1294899	0.2561
2	2004-08-27	TCS	EQ	979.00	982.4	982.4	958.55	961.20	962.65	969.94	3830750	3.715586e+14	NaN	976527	0.2549
3	2004-08-30	TCS	EQ	962.65	969.9	990.0	965.00	986.40	986.75	982.65	3058151	3.005106e+14	NaN	701664	0.2294
4	2004-08-31	TCS	EQ	986.75	986.5	990.0	976.00	987.80	988.10	982.18	2649332	2.602133e+14	NaN	695234	0.2624

Summary Data Set and Description of the Data Set

Summary of the dataset

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4287 entries, 2004-08-25 to 2021-11-30
Data columns (total 14 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Symbol              4287 non-null   object
 1   Series              4287 non-null   object
 2   Prev Close          4287 non-null   float64
 3   Open                4287 non-null   float64
 4   High                4287 non-null   float64
 5   Low                 4287 non-null   float64
 6   Last                4287 non-null   float64
 7   Close               4287 non-null   float64
 8   VWAP                4287 non-null   float64
 9   Volume              4287 non-null   int64
 10  Turnover            4287 non-null   float64
 11  Trades              2603 non-null   float64
 12  Deliverable Volume  4287 non-null   int64
 13  %Deliverble         4287 non-null   float64
dtypes: float64(10), int64(2), object(2)
memory usage: 502.4+ KB
```

Datatypes of each column in the dataset

```
Symbol                  object
Series                  object
Prev Close             float64
Open                   float64
High                   float64
Low                    float64
Last                   float64
Close                  float64
VWAP                   float64
Volume                   int64
Turnover               float64
Trades                 float64
Deliverable Volume       int64
%Deliverble            float64
dtype: object
```

Checking for missing values in the dataset

```
Symbol                  0
Series                  0
Prev Close              0
Open                    0
High                    0
Low                     0
Last                    0
Close                   0
VWAP                    0
Volume                  0
Turnover                0
Trades               1684
Deliverable Volume      0
%Deliverble             0
dtype: int64
```

Dataset Description

The dataset contains several columns some of the important columns are as follows: -

- Date - Denotes each trade dates
- Symbol - Denotes traded company
- Previous Close - Previous day market close price
- Open - Current day open price
- High - Current day high price traded
- Low - Current day high price traded
- Last - Most recent transaction price
- Close - Close price of day market

Statistical properties of dataset:

```
df.describe()
```

	Close
count	4287.000000
mean	1752.734395
std	778.528524
min	366.650000
25%	1117.500000
50%	1717.800000
75%	2397.775000
max	3954.550000

Pre-processing the Dataset

Data pre-processing is a process of preparing the raw data and making it suitable for a machine learning model. It is the first and crucial step while creating a machine learning model.

When creating a machine learning project, it is not always a case that we come across the clean and formatted data. And while doing any operation with data, it is mandatory to clean it and put in a formatted way. So, for this, we use data pre-processing task.

A real-world data generally contains noises, missing values, and maybe in an unusable format which cannot be directly used for machine learning models. Data pre-processing is required tasks for cleaning the data and making it suitable for a machine learning model which also increases the accuracy and efficiency of a machine learning model.

Dropping some Columns:

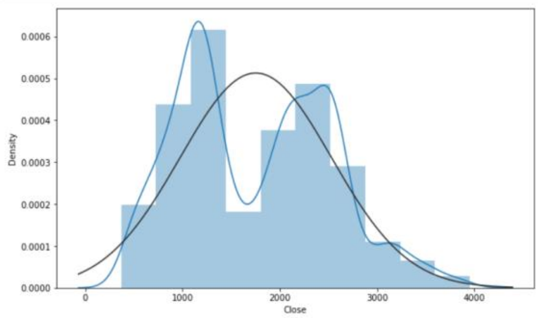
```
df=df.drop(['Symbol','Series','Turnover','Trades','Deliverable Volume','%Deliverble'],axis=1)
df.head()
```

	Prev Close	Open	High	Low	Last	Close	VWAP	Volume
Date								
2004-08-25	850.00	1198.7	1198.7	979.00	985.00	987.95	1008.32	17116372
2004-08-26	987.95	992.0	997.0	975.30	976.85	979.00	985.65	5055400
2004-08-27	979.00	982.4	982.4	958.55	961.20	962.65	969.94	3830750
2004-08-30	962.65	969.9	990.0	965.00	986.40	986.75	982.65	3058151
2004-08-31	986.75	986.5	990.0	976.00	987.80	988.10	982.18	2649332

Visualize the frequency distribution of Close variable using:

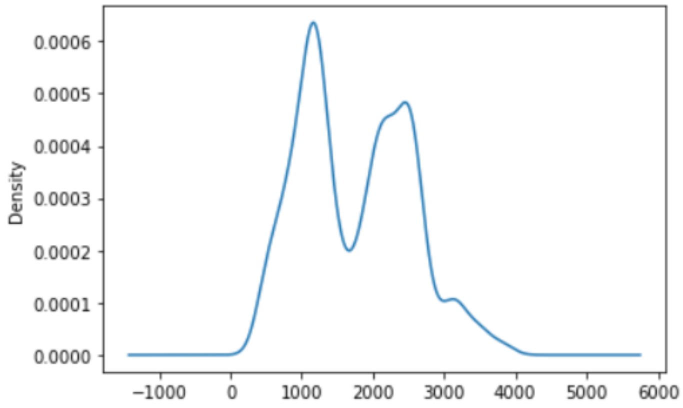
1.Distplot for Close:

```
from scipy import stats
f, ax = plt.subplots(figsize=(10,6))
x = df['Close']
ax = sns.distplot(x, bins=10,fit=stats.norm)
plt.show()
```



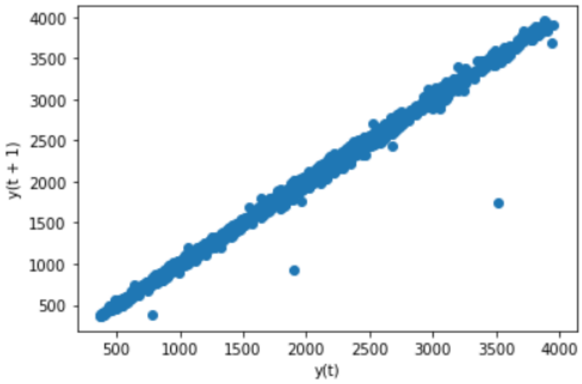
2.Density Plot for Close:

```
df['Close'].plot(kind='kde')
plt.show()
```



3.Plotted lag Graph for Close:

```
# Lag Scatter Plots
lag_plot(df['Close'])
plt.show()
```



Findings:

From the lag plot we can see that the close are not completely random the follow a linear type of relationship this indicates that there is no white noise in the dataset.

1st Order Difference and Second Order Difference and Plotting:

Differencing can help stabilise the mean of a time series by removing changes in the level of a time series, and therefore eliminating (or reducing) trend and seasonality.

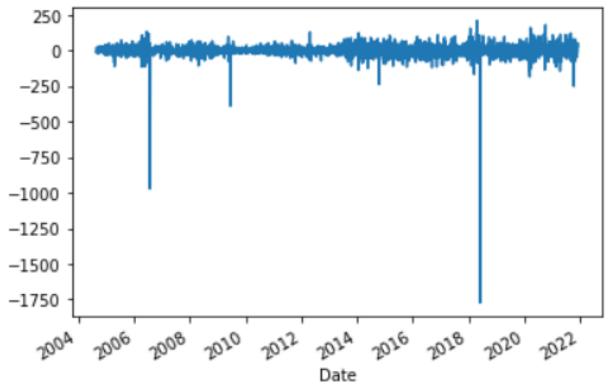
```
df['1st Order Difference'] = df['Close'] - df['Close'].shift(1)
```

```
df['Second Order Difference']=df['Close']-df['Close'].shift(2)
df.head()
```

Date	Prev Close	Open	High	Low	Last	Close	VWAP	Volume	1st Order Difference	Second Order Difference
2004-08-25	850.00	1198.7	1198.7	979.00	985.00	987.95	1008.32	17116372	NaN	NaN
2004-08-26	987.95	992.0	997.0	975.30	976.85	979.00	985.65	5055400	-8.95	NaN
2004-08-27	979.00	982.4	982.4	958.55	961.20	962.65	969.94	3830750	-16.35	-25.30
2004-08-30	962.65	969.9	990.0	965.00	986.40	986.75	982.65	3058151	24.10	7.75
2004-08-31	986.75	986.5	990.0	976.00	987.80	988.10	982.18	2649332	1.35	25.45

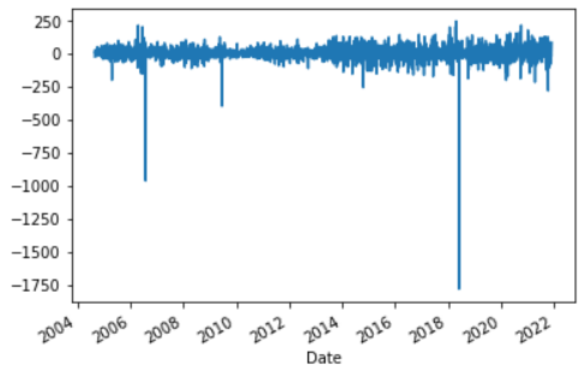
```
df['1st Order Difference'].plot()
```

<AxesSubplot:xlabel='Date'>




```
df['Second Order Difference'].plot()
```

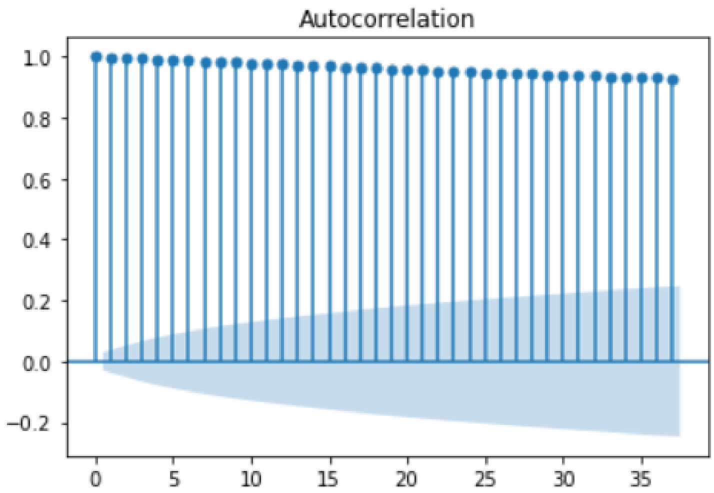
<AxesSubplot:xlabel='Date'>



Autocorrelation Plot:

Autocorrelation represents the degree of similarity between a given time series and a lagged version of itself over successive time intervals. Autocorrelation measures the relationship between a variable's current value and its past values.

```
plot_acf(df['Close'])
plt.show()
```



Findings:

- From the autocorrelation plot of close price, its clear that the close price has strong autocorrelation with itself.
- Since it has very high degree of similarity with its own lagged version of itself.

Partial Autocorrelation Plot:

The partial autocorrelation function is a measure of the correlation between observations of a time series that are separated by k time units (y t and y t–k), after adjusting for the presence of all the other terms of shorter lag (y t–1, y t–2, ..., y t–k–1).

Identification of an AR model is often best done with the PACF.

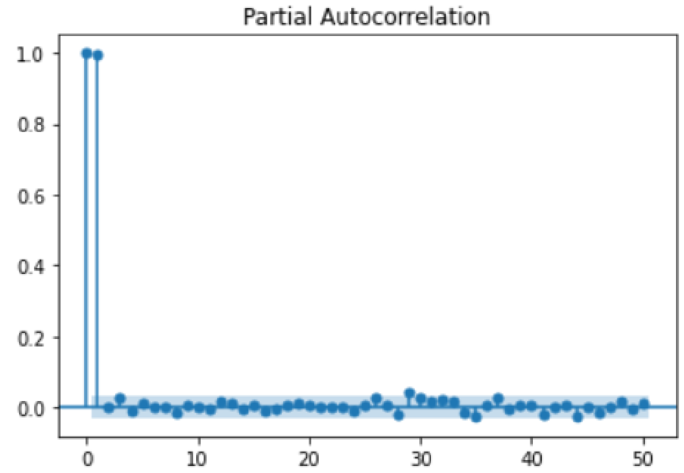
- For an AR model, the theoretical PACF "shuts off" past the order of the model. The phrase "shuts off" means that in theory the partial autocorrelations are equal to 0 beyond that point. Put another way, the number of non-zero partial autocorrelations give the order of the AR model. By the "order of the model" we mean the most extreme lag of x that is used as a predictor.

Identification of an MA model is often best done with the ACF rather than the PACF.

- For an MA model, the theoretical PACF does not shut off, but instead tapers toward 0 in some manner. A clearer pattern for an MA model is in the ACF. The ACF will have non-zero autocorrelations only at lags involved in the model.

p,d,q p AR model lags d differencing q MA lags

```
plot_pacf(df['Close'], lags=50)
plt.show()
```



Detrend the Stochastic trend:

Take the log value for the Close Column and Difference the log value to Detrend the Stochastic trend

```
df['logClose']=np.log(df['Close'])
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4287 entries, 2004-08-25 to 2021-11-30
Data columns (total 11 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   Prev Close              4287 non-null   float64
 1   Open                    4287 non-null   float64
 2   High                    4287 non-null   float64
 3   Low                     4287 non-null   float64
 4   Last                    4287 non-null   float64
 5   Close                   4287 non-null   float64
 6   VWAP                    4287 non-null   float64
 7   Volume                  4287 non-null   int64  
 8   1st Order Difference    4286 non-null   float64
 9   Second Order Difference 4285 non-null   float64
 10  logClose                4287 non-null   float64
dtypes: float64(10), int64(1)
memory usage: 401.9 KB
```

```
df['Detrend'] = df['logClose'] - df['logClose'].shift(1)
df.head()
```

logClose	Detrend
6.895632	NaN
6.886532	-0.009100
6.869690	-0.016842
6.894417	0.024727
6.895784	0.001367

Plotting the Close Price:

Creating a Training Set and a Test Set for Stock Market Prediction:

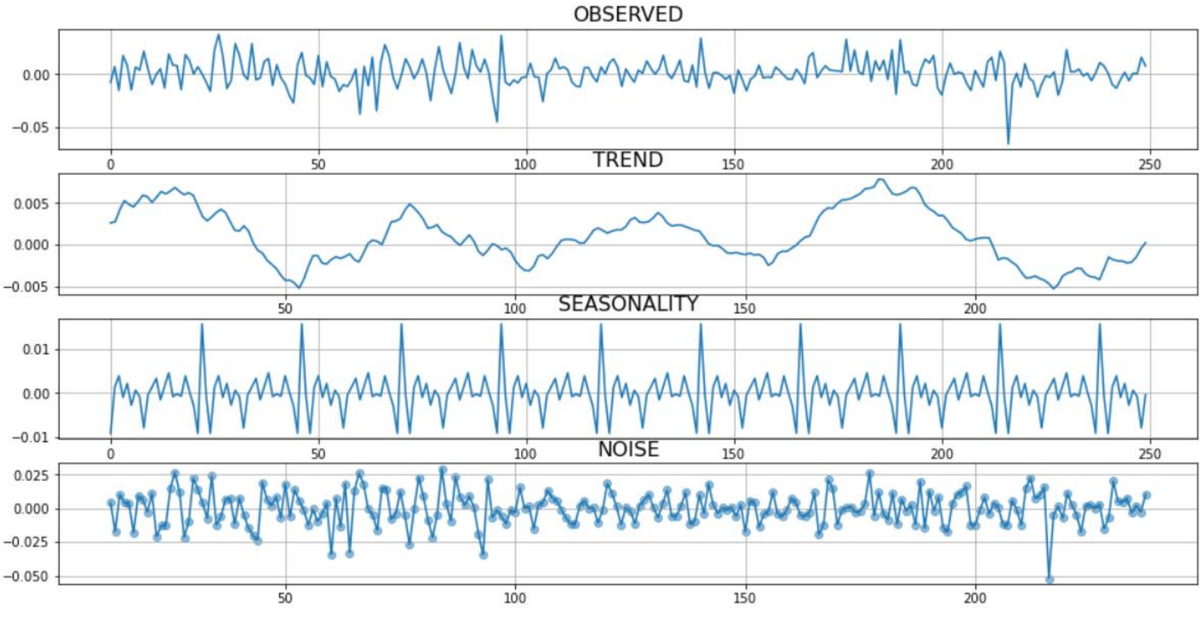
We have to divide the entire dataset into training and test sets before feeding it into the training model. The Machine Learning LSTM model will be trained on the data in the training set and tested for accuracy and backpropagation on the test set.

```
print(df.shape)
train=df.iloc[:-30]
test=df.iloc[-30:]
print(train.shape,test.shape)
```

```
(4287, 12)
(4257, 12) (30, 12)
```

The Seasonal Decomposition of The Close Price:

Time series decomposition involves thinking of a series as a combination of level, trend, seasonality, and noise components. Decomposition provides a useful abstract model for thinking about time series generally and for better understanding problems during time series analysis and forecasting



Standardization the value:

Standardization or Z-Score Normalization is the transformation of features by subtracting from mean and dividing by standard deviation. This is often called as Z-score.

```
ss = StandardScaler()
df.iloc[:,:-1] = ss.fit_transform(df.iloc[:,:-1])
print(df)
```

```
            Prev Close      Open      High       Low      Last     Close  \
Date
2004-08-25   -1.159383 -0.713369 -0.734491 -0.973672 -0.986007 -0.982461
2004-08-26   -0.982090 -0.978949 -0.991756 -0.978459 -0.996475 -0.993958
2004-08-27   -0.993592 -0.991284 -1.010378 -1.000127 -1.016575 -1.014962
2004-08-30   -1.014605 -1.007345 -1.000684 -0.991783 -0.984209 -0.984002
2004-08-31   -0.983632 -0.986016 -1.000684 -0.977553 -0.982411 -0.982268
...                ...       ...       ...       ...       ...       ...
2021-11-24    2.200448  2.207762  2.191915  2.189287  2.156798  2.171742
2021-11-25    2.173523  2.171401  2.153587  2.202289  2.179981  2.175082
2021-11-26    2.176864  2.147117  2.188025  2.173634  2.166110  2.176302
2021-11-29    2.178085  2.175577  2.238216  2.166584  2.248695  2.247149
2021-11-30    2.248964  2.231918  2.282092  2.275962  2.289153  2.282027

                VWAP    Volume  1st Order Difference  Second Order Difference  \
Date
2004-08-25 -0.956482  9.601614                   NaN                      NaN
2004-08-26 -0.985600  2.088231             -0.216808                      NaN
2004-08-27 -1.005778  1.325336             -0.384930                -0.425477
2004-08-30 -0.989453  0.844045              0.534063                 0.105529
2004-08-31 -0.990057  0.589371              0.017201                 0.389911
...              ...       ...                   ...                      ...
2021-11-24  2.205086  0.285630             -0.489439                -0.261597
2021-11-25  2.181633  0.091190              0.045600                -0.313813
2021-11-26  2.185306  0.148272              0.008113                 0.038049
2021-11-29  2.231391  0.796448              1.239496                 0.882357
2021-11-30  2.289447  2.420292              0.603357                 1.303306
```

Normalization the value:

Normalization is the process of reorganizing data in a database so that it meets two basic requirements:

- There is no redundancy of data, all data is stored in only one place.
- Data dependencies are logical,all related data items are stored together.

```
norm = Normalizer()
df.dropna().iloc[:,:-1] = norm.fit_transform(df.dropna().iloc[:,:-1])
df
```

Date	Prev Close	Open	High	Low	Last	Close	VWAP	Volume	1st Order Difference	Second Order Difference	logClose	Detrend
2004-08-25	-1.159383	-0.713369	-0.734491	-0.973672	-0.986007	-0.982461	-0.956482	9.601614	NaN	NaN	-0.926419	NaN
2004-08-26	-0.982090	-0.978949	-0.991756	-0.978459	-0.996475	-0.993958	-0.985600	2.088231	-0.216808	NaN	-0.944721	-0.009100
2004-08-27	-0.993592	-0.991284	-1.010378	-1.000127	-1.016575	-1.014962	-1.005778	1.325336	-0.384930	-0.425477	-0.978591	-0.016842
2004-08-30	-1.014605	-1.007345	-1.000684	-0.991783	-0.984209	-0.984002	-0.989453	0.844045	0.534063	0.105529	-0.928863	0.024727
2004-08-31	-0.983632	-0.986016	-1.000684	-0.977553	-0.982411	-0.982268	-0.990057	0.589371	0.017201	0.389911	-0.926114	0.001367
...	...	...	...	...	...	...	...	...	...	...	...	...
2021-11-24	2.200448	2.207762	2.191915	2.189287	2.156798	2.171742	2.205086	0.285630	-0.489439	-0.261597	1.584501	-0.006066
2021-11-25	2.173523	2.171401	2.153587	2.202289	2.179981	2.175082	2.181633	0.091190	0.045600	-0.313813	1.586019	0.000755
2021-11-26	2.176864	2.147117	2.188025	2.173634	2.166110	2.176302	2.185306	0.148272	0.008113	0.038049	1.586573	0.000276
2021-11-29	2.178085	2.175577	2.238216	2.166584	2.248695	2.247149	2.231391	0.796448	1.239496	0.882357	1.618496	0.015873
2021-11-30	2.248964	2.231918	2.282092	2.275962	2.289153	2.282027	2.289447	2.420292	0.603357	1.303306	1.634027	0.007723

Finding MinMaxScaler:

We will scale the stock values to values between 0 and 1. As a result, all of the data in large numbers is reduced, and therefore memory consumption is decreased. Also, because the data is not spread out in huge values, we can achieve greater precision by scaling down. To perform this we will be using the MinMaxScaler class of the sci-kit-learn library.

```
trans = MinMaxScaler()
df2 = trans.fit_transform(df)
```

```
df2
```

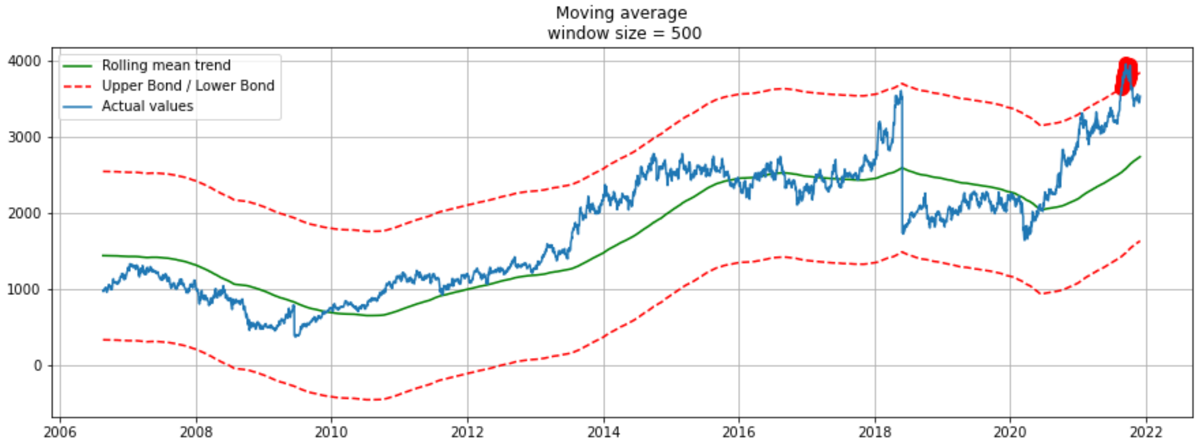
```
array([[0.13471669, 0.23492997, 0.22727461, ...,        nan, 0.41679353,
          nan],
       [0.17316536, 0.17703081, 0.17143529, ...,        nan, 0.41296695,
        0.82190246],
       [0.17067087, 0.17434174, 0.16739338, ..., 0.86738572, 0.40588527,
        0.81290093],
       ...,
       [0.85823183, 0.85854342, 0.86160597, ..., 0.88162701, 0.94222302,
        0.83280493],
       [0.85849661, 0.8647479 , 0.87249976, ..., 0.90756738, 0.94889755,
        0.85094197],
       [0.87386772, 0.87703081, 0.88202317, ..., 0.92050054, 0.95214487,
        0.84146444]])
```

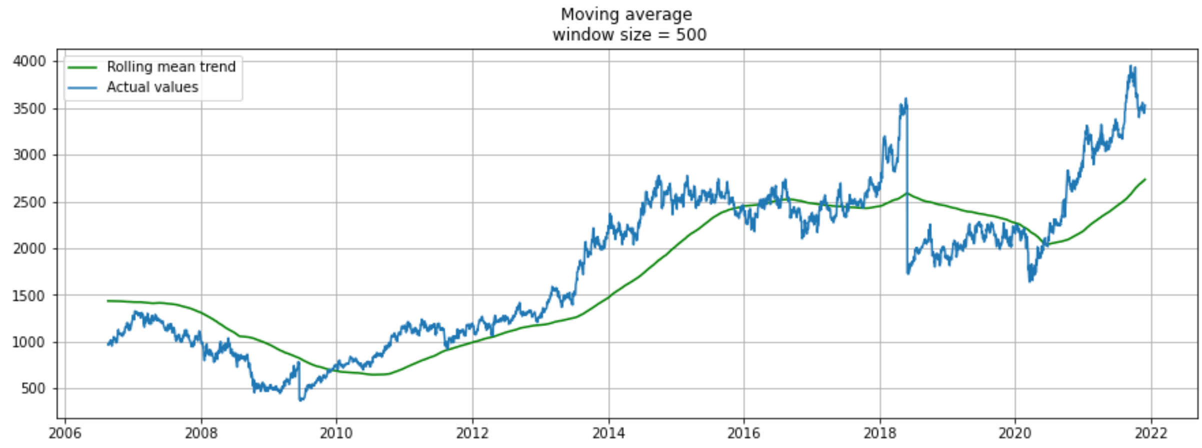
Smoothening of Close Price

Smoothing is a technique applied to time series to remove the fine-grained variation between time steps. The hope of smoothing is to remove noise and better expose the signal of the underlying causal processes.

Plotting the moving average

```
def plotMovingAverage(series, window, plot_intervals=False, scale=1.96, plot_anomalies=False):

    """
        series - dataframe with timeseries
        window - rolling window size
        plot_intervals - show confidence intervals
        plot_anomalies - show anomalies

    """
    rolling_mean = series.rolling(window=window).mean()

    plt.figure(figsize=(15,5))
    plt.title("Moving average\n window size = {}".format(window))
    plt.plot(rolling_mean, "g", label="Rolling mean trend")

    # Plot confidence intervals for smoothed values
    if plot_intervals:
        mae = mean_absolute_error(series[window:], rolling_mean[window:])
        deviation = np.std(series[window:] - rolling_mean[window:])
        lower_bond = rolling_mean - (mae + scale * deviation)
        upper_bond = rolling_mean + (mae + scale * deviation)
        plt.plot(upper_bond, "r--", label="Upper Bond / Lower Bond")
        plt.plot(lower_bond, "r--")

        # Having the intervals, find abnormal values
        if plot_anomalies:
            anomalies = pd.DataFrame(index=series.index, columns=series.columns)
            anomalies[series<lower_bond] = series[series<lower_bond]
            anomalies[series>upper_bond] = series[series>upper_bond]
            plt.plot(anomalies, "ro", markersize=10)

    plt.plot(series[window:], label="Actual values")
    plt.legend(loc="upper left")
    plt.grid(True)
```



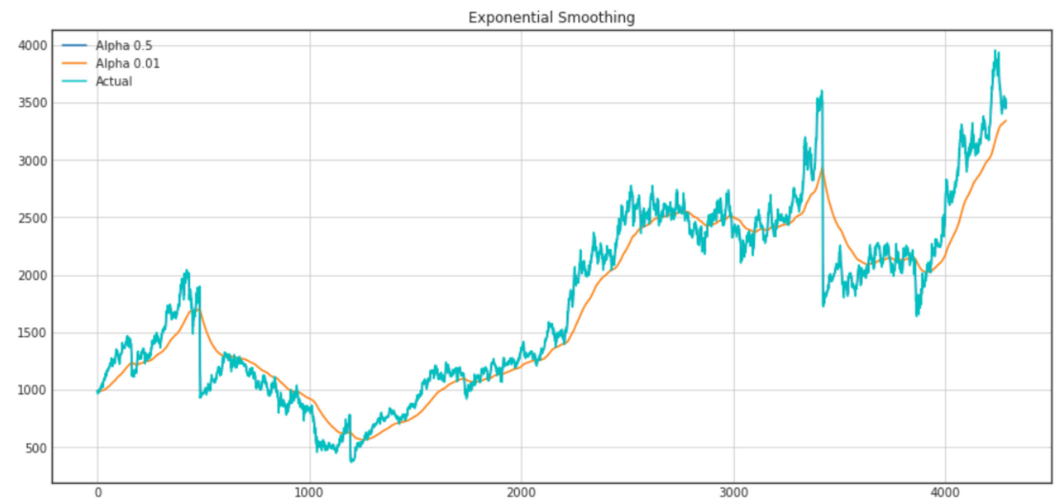


Exponential Smoothing

Exponential smoothing is a time series forecasting method for univariate data that can be extended to support data with a systematic trend or seasonal component. It is a powerful forecasting method that may be used as an alternative to the popular Box-Jenkins ARIMA family of methods.

```
def plotExponentialSmoothing(series, alphas):
    """
        Plots exponential smoothing with different alphas

        series - dataset with timestamps
        alphas - list of floats, smoothing parameters

    """
    with plt.style.context('seaborn-white'):
        plt.figure(figsize=(15, 7))
        for alpha in alphas:
            plt.plot(exponential_smoothing(series, alpha), label="Alpha {}".format(alpha))
        plt.plot(series.values, "c", label = "Actual")
        plt.legend(loc="best")
        plt.axis('tight')
        plt.title("Exponential Smoothing")
        plt.grid(True);
```

```
plotExponentialSmoothing(df.Close, [0.5, 0.010])
```



ARIMA Model

Determine the p, d and q values and Forecasting:

The auto-ARIMA process seeks to identify the most optimal parameters for an ARIMA model, settling on a single fitted ARIMA model.

The standard ARIMA models expect as input parameters 3 arguments p,d,q.

- p is the number of lag observations.
- d is the degree of differencing.
- q is the size/width of the moving average window.

We simply used the *.fit()* command to fit the model without having to select the combination of p, q, d.

```
stepwise_fit = auto_arima(df['Close'], trace=True,
suppress_warnings=True)
stepwise_fit.summary()
```

```
Performing stepwise search to minimize aic
 ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=44611.785, Time=3.96 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=44608.244, Time=0.09 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=44610.244, Time=0.12 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=44610.245, Time=0.31 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=44607.022, Time=0.06 sec
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=44609.312, Time=1.78 sec

Best model:  ARIMA(0,1,0)(0,0,0)[0]
Total fit time: 6.329 seconds
```

ARIMA, abbreviated for 'Auto Regressive Integrated Moving Average', is a class of models that 'demonstrates' a given time series based on its previous values: its lags and the lagged errors in forecasting, so that equation can be utilized in order to forecast future values.

```
model=ARIMA(train['Close'],order=(0,1,0))
model=model.fit()
model.summary()
```

ARIMA Model Results

Dep. Variable:	D.Close	**No. Observations:**	4256
Model:	ARIMA(0, 1, 0)	**Log Likelihood**	-22152.820
Method:	css	**S.D. of innovations**	44.086
Date:	Wed, 27 Apr 2022	**AIC**	44309.639
Time:	21:34:05	**BIC**	44322.352
Sample:	1	**HQIC**	44314.131

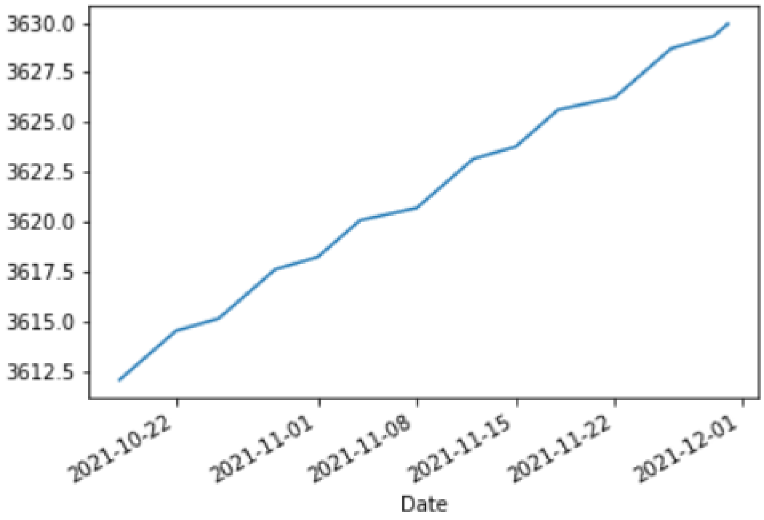
	coef	std err	z	P>|z|	[0.025	0.975]
const	0.6164	0.676	0.912	0.362	-0.708	1.941

Predict the Future Values

```
start=len(train)
end=len(train)+len(test)-1
pred1=model.predict(start=start,end=end,typ='levels')
pred1.index=df['Close'].index[start:end+1]
print(pred1)
```

```
Date
2021-10-18    3612.066424
2021-10-19    3612.682848
2021-10-20    3613.299272
2021-10-21    3613.915695
2021-10-22    3614.532119
2021-10-25    3615.148543
2021-10-26    3615.764967
2021-10-27    3616.381391
2021-10-28    3616.997815
2021-10-29    3617.614239
2021-11-01    3618.230663
2021-11-02    3618.847086
2021-11-03    3619.463510
2021-11-04    3620.079934
2021-11-08    3620.696358
2021-11-09    3621.312782
2021-11-10    3621.929206
2021-11-11    3622.545630
2021-11-12    3623.162054
2021-11-15    3623.778477
2021-11-16    3624.394901
2021-11-17    3625.011325
2021-11-18    3625.627749
2021-11-22    3626.244173
2021-11-23    3626.860597
2021-11-24    3627.477021
2021-11-25    3628.093445
2021-11-26    3628.709868
```

```
pred1.plot()
plt.show()
```



Forecasting Accuracy

The error is measured by fitting points for the time periods with historical data and then comparing the fitted points to the historical data

1 . MAD(Mean Absolute Deviation):

The mean absolute deviation of a dataset is the average distance between each data point and the mean. It gives us an idea about the variability in a dataset.

2 . MSE(Mean Square Error):

MSE is the average of the squared error that is used as the loss functio n for least squares regression: It is the sum, over all the data points, of the square of the difference between the predicted and actual target va riables, divided by the number of data points.

3 . RMSE(Root Mean Square Error):

RMSE is the standard deviation of the errors which occur when a pred iction is made on a dataset.

4 . MAE(Mean Absolute Error):

Mean Absolute error refers to the magnitude of difference between the prediction of an observation and the true value of that observation

5 . MAPE(Mean Absolute Percentage Error):

It is a statistical measure to define the accuracy of a machine learning algorithm on a particular dataset. MAPE can be considered as a loss function to define the error termed by the model evaluation.

6 . Mean:

The mean value is the average value. To calculate the mean, find the sum of all values, and divide the sum by the number of values

```
MAD=test['Close'].mad()
print(MAD)
```

```
39.853999999999984
```

```
mse=mean_squared_error(test['Close'],pred1)
print(mse)
rmse=sqrt(mse)
print(rmse)
```

```
17378.113898973774
131.82607442753414
```

```
mae = mean_absolute_error(test['Close'],pred1)
print(mae)
```

```
122.37961857769251
```

```
def MAPE(Y_actual,Y_Predicted):
    mape = np.mean(np.abs((Y_actual - Y_Predicted)/Y_actual))*100
    return mape
print(MAPE(test['Close'],pred1))
```

```
3.5163117284808574
```

```
test['Close'].mean()
```

```
3502.394999999999
```

Preprocessing for LSTM Model

Variables chosen for training

```
#Variables for training
cols = list(df)[3:9]
#Date and volume columns are not used in training.
print(cols) #['Prev Close', 'Open', 'High', 'Low', 'Last', 'Close']
```

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

```
df_for_training = df[cols].astype(float)
```

Scaling the values

scale all the values using standard scaler.

```
scaler = StandardScaler()
scaler = scaler.fit(df_for_training)
df_for_training_scaled = scaler.transform(df_for_training)
```

Preprocessing the values for passing it to LSTM neural network

```
trainX = []
trainY = []

n_future = 1   # Number of days we want to look into the future based on the past days.
n_past = 14  # Number of past days we want to use to predict the future.
```

```
for i in range(n_past, len(df_for_training_scaled) - n_future +1):
    trainX.append(df_for_training_scaled[i - n_past:i, 0:df_for_training.shape[1]])
    trainY.append(df_for_training_scaled[i + n_future - 1:i + n_future, 0])

trainX, trainY = np.array(trainX), np.array(trainY)

print('trainX shape == {}.'.format(trainX.shape))
print('trainY shape == {}.'.format(trainY.shape))
```

```
trainX shape == (4273, 14, 6).
trainY shape == (4273, 1).
```

LSTM Modeling

LSTM Model:

Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning (DL). Unlike standard feedforward neural networks, LSTM has feedback connections. It can process not only single data points (such as images), but also entire sequences of data (such as speech or video). A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. LSTM networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series. LSTMs were developed to deal with the vanishing gradient problem that can be encountered when training traditional RNNs.

Model 1:

Here, 2 LSTM layers are used with their activation functions as relu, adam as optimizer and loss as mse(mean squared error).

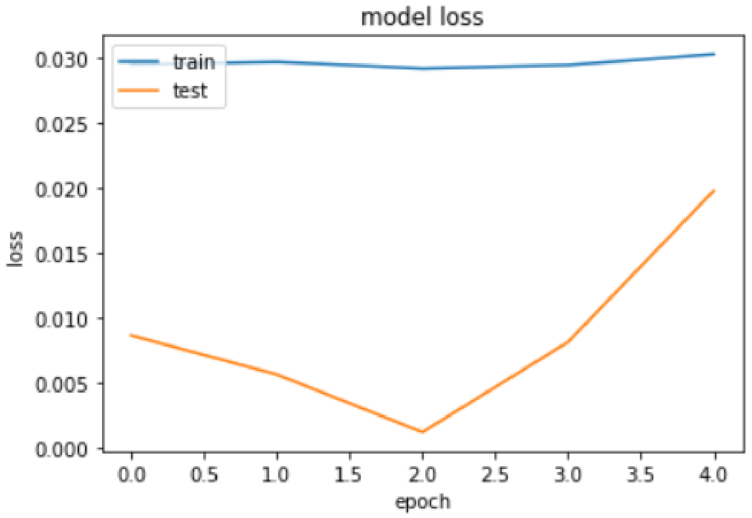
```
model = Sequential()
model.add(LSTM(64, activation='relu', input_shape=(trainX.shape[1], trainX.shape[2]), return_sequences=True))
model.add(LSTM(32, activation='relu', return_sequences=False))
model.add(Dropout(0.3))
model.add(Dense(trainY.shape[1]))

model.compile(optimizer='adam', loss='mse')
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_11 (LSTM)              (None, 14, 64)            18176

 lstm_12 (LSTM)              (None, 32)                12416

 dropout_5 (Dropout)         (None, 32)                0

 dense_5 (Dense)             (None, 1)                 33

=================================================================
Total params: 30,625
Trainable params: 30,625
Non-trainable params: 0
```

```
history = model.fit(trainX, trainY, epochs=5, batch_size=2, validation_split=0.1, verbose=1)
```

```
Epoch 1/5
1923/1923 [==============================] - 12s 6ms/step - loss: 0.0295 - val_loss: 0.0087
Epoch 2/5
1923/1923 [==============================] - 13s 7ms/step - loss: 0.0297 - val_loss: 0.0057
Epoch 3/5
1923/1923 [==============================] - 12s 6ms/step - loss: 0.0292 - val_loss: 0.0012
Epoch 4/5
1923/1923 [==============================] - 13s 7ms/step - loss: 0.0294 - val_loss: 0.0082
Epoch 5/5
1923/1923 [==============================] - 13s 7ms/step - loss: 0.0303 - val_loss: 0.0198
```

```
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



This is the plot of loss in the model.

Forecasting the future values using the model,

```
pre = model.predict(trainX)
```

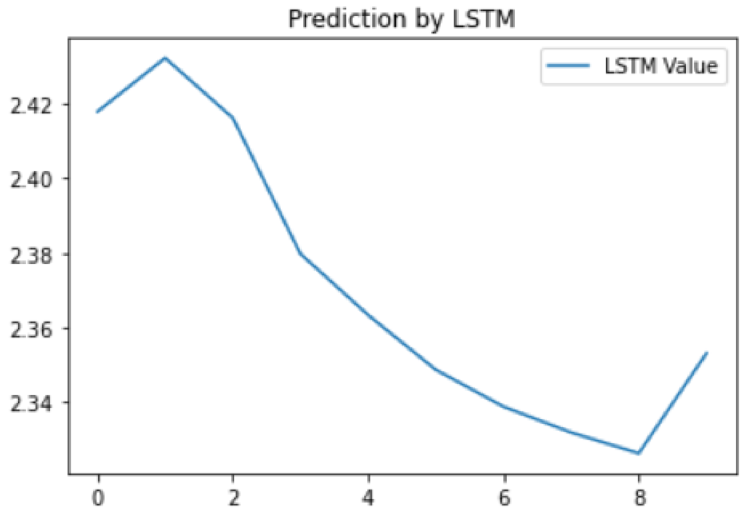
```
n_past = 2
n_days_for_prediction=10
```

```
prediction = model.predict(trainX[-n_days_for_prediction:]) #
#prediction = model.predict(trainX[-n_past:])
```

```
prediction_copies = np.repeat(prediction, df_for_training.shape[1], axis=-1)
y_pred_future = scaler.inverse_transform(prediction_copies)[:,0]
```

The model predictions before inverse transformation of scaling.

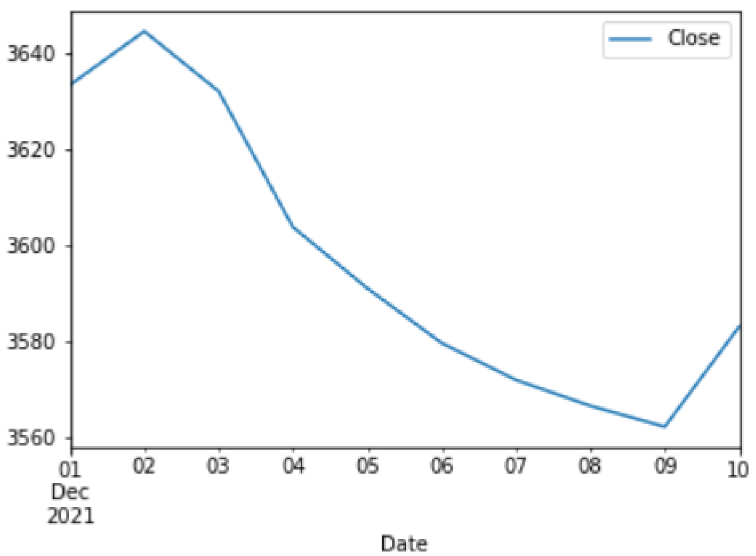
```
plt.plot(prediction, label='LSTM Value')
plt.title("Prediction by LSTM")
plt.legend()
plt.show()
```



The forecasted value of the close price by the model

```
index.plot()
```

<AxesSubplot:xlabel='Date'>



The plot of forecasted value with the previous close price values

```
import matplotlib.pyplot as plt
plt.plot(df1.Close)
plt.plot(index)
```

```
[<matplotlib.lines.Line2D at 0x22e6a77e7c0>]
```

Model -2:

Here, 2 LSTM layers are used with their activation functions as tanh, recurrent activation function as sigmoid, adam as optimizer and loss as mse(mean squared error).

```
model = Sequential()
model.add(LSTM(64, activation="tanh",recurrent_activation="sigmoid", input_shape=(trainX.shape[1], trainX.shape[2]), return_seque
model.add(LSTM(32, activation="tanh",recurrent_activation="sigmoid", return_sequences=False))
model.add(Dropout(0.3))
model.add(Dense(trainY.shape[1]))

model.compile(optimizer='adam', loss='mse')
model.summary()
```

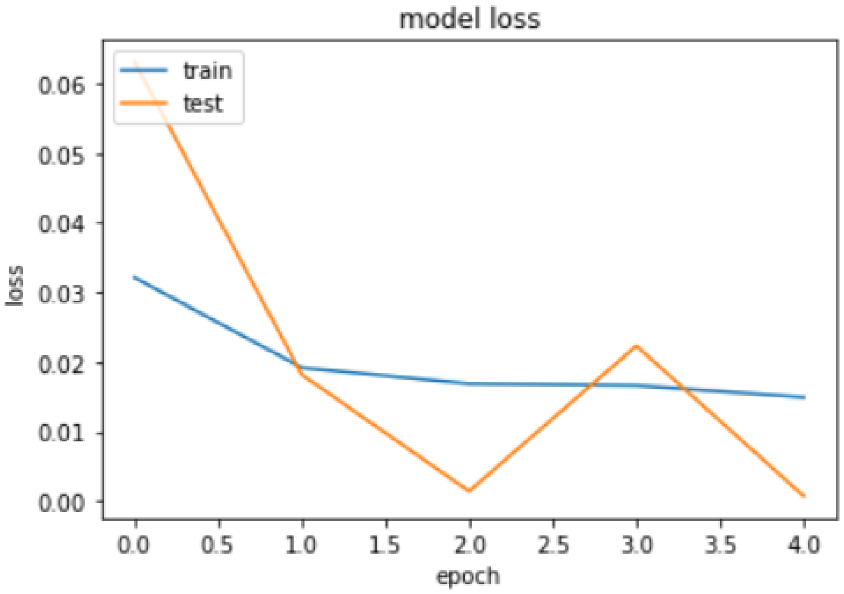
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 lstm (LSTM)                 (None, 14, 64)            18176     
                                                                 
 lstm_1 (LSTM)               (None, 32)                12416     
                                                                 
 dropout (Dropout)           (None, 32)                0         
                                                                 
 dense (Dense)               (None, 1)                 33        
                                                                 
=================================================================
Total params: 30,625
Trainable params: 30,625
Non-trainable params: 0
_________________________________________________________________
```

```
history = model.fit(trainX, trainY, epochs=5, batch_size=2, validation_split=0.1, verbose=1)
```

```
Epoch 1/5
1923/1923 [==============================] - 48s 22ms/step - loss: 0.0321 - val_loss: 0.0631
Epoch 2/5
1923/1923 [==============================] - 40s 21ms/step - loss: 0.0192 - val_loss: 0.0182
Epoch 3/5
1923/1923 [==============================] - 40s 21ms/step - loss: 0.0169 - val_loss: 0.0015
Epoch 4/5
1923/1923 [==============================] - 41s 22ms/step - loss: 0.0166 - val_loss: 0.0223
Epoch 5/5
1923/1923 [==============================] - 40s 21ms/step - loss: 0.0149 - val_loss: 7.7536e-04
```

This is the loss plot for the model -2

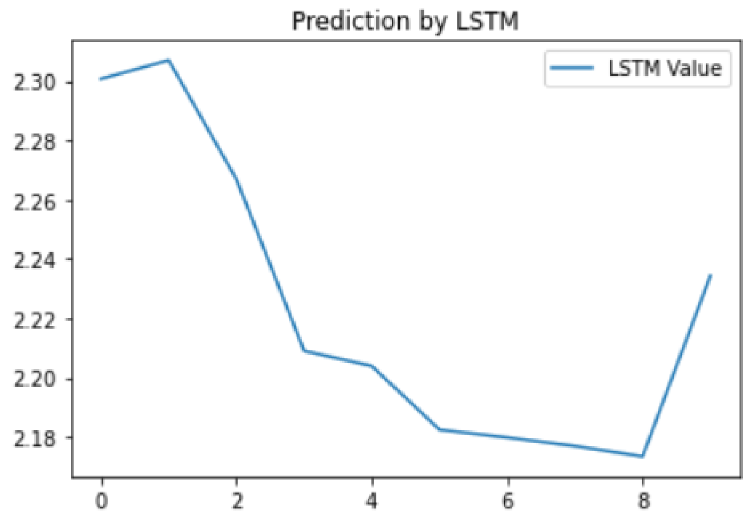
```
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



The Forecasted Close Price by model -2:

This is the forecasted close price values before inverse transform of the scaling.

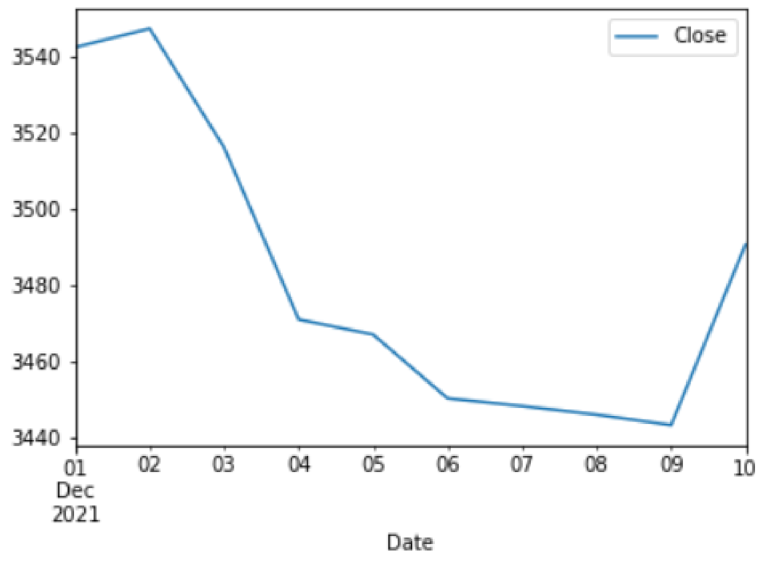
```
plt.plot(prediction, label='LSTM Value')
plt.title("Prediction by LSTM")
plt.legend()
plt.show()
```



The close price values after inverse transform the scaling,

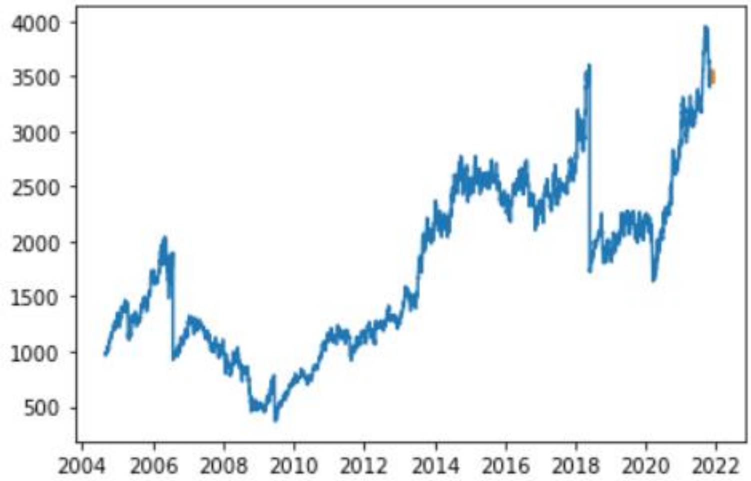
```
index.plot()
```

```
<AxesSubplot:xlabel='Date'>
```



```python
import matplotlib.pyplot as plt
plt.plot(df1.Close)
plt.plot(index)
```

```
[<matplotlib.lines.Line2D at 0x28fea8a9f40>]
```



The model -2 (i.e., the model with tanh as activation function and sigmoid as recurrent activation function) performs well compared to the previous model (i.e., the model with relu as activation function).

Conclusion

The LSTM model provides better results compared to ARIMA model but they are not completely accurate the accuracy can be improved by improving preprocessing of data and the models can improved by tuning them with right parameters. Since it’s a huge time series the data set can be split up to various time ranges and they can be preprocessed to improve the performance of both the ARIMA and LSTM models.