Lab Group: FSP6

Dataset: Alpha Vantage Stock APIs , YahooFinance

Problem Statement:

1. To what extent is the price of Bitcoin dependent on the global financial system that is represented through stock indices?
2. Which is the more accurate model in predicting bitcoin prices, Autoregression or Long Short Term Memory?

Group Members: Ingale Omkar, Lau Chen Yi Wynne, Himari Ang Lixin

Data Extraction and Data Cleaning

```
# Importing essential libraries
import numpy as np
import pandas as pd
import seaborn as sb
import matplotlib.pyplot as plt
%matplotlib inline
import yfinance as yf
import math
import datetime

# Importing libraries needed for auto regression
from pandas.plotting import lag_plot,autocorrelation_plot
from statsmodels.tsa.ar_model import AR
from sklearn.metrics import mean_squared_error

# importing libraries needed for VAR
import statsmodels.api as sm
from statsmodels.tsa.api import VAR
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.stattools import grangercausalitytests

# importing libraries needed for LSTM
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM
```

```
/usr/local/lib/python3.7/dist-packages/statsmodels/tools/_testing.py:19: Futur
  import pandas.util.testing as tm
```

```
from alpha_vantage.cryptocurrencies import CryptoCurrencies
```

```
# This is used to download BTC data from AlphaVantage
crypt = CryptoCurrencies('WQ8W67GO6PP73AD4', output_format = 'pandas')
```

```
# Extract BTC data from AlphaVantage
btc_av = crypt.get_digital_currency_daily('BTC', market='USD')[0]
btc_av = pd.DataFrame(btc_av)
btc_av = btc_av['2021-03-24':]
btc_av
```

	1a. open (USD)	1b. open (USD)	2a. high (USD)	2b. high (USD)	3a. low (USD)	3b. low (USD)	4a. close (USD)	4b. close (USD)	
date									
2021-03-24	54342.80	54342.80	57200.00	57200.00	51700.00	51700.00	52303.65	52303.65	83
2021-03-23	54083.25	54083.25	55830.90	55830.90	53000.00	53000.00	54340.89	54340.89	59
2021-03-22	57351.56	57351.56	58430.73	58430.73	53650.00	53650.00	54083.25	54083.25	62
2021-03-21	58100.02	58100.02	58589.10	58589.10	55450.11	55450.11	57351.56	57351.56	48
2021-03-20	58030.01	58030.01	59880.00	59880.00	57820.17	57820.17	58102.28	58102.28	44
...	...	...	...	...	...	...	...	...	
2018-08-01	7735.67	7735.67	7750.00	7750.00	7430.00	7430.00	7604.58	7604.58	42
2018-07-31	8171.40	8171.40	8180.00	8180.00	7633.00	7633.00	7730.93	7730.93	48
2018-07-30	8210.99	8210.99	8273.00	8273.00	7866.00	7866.00	8173.92	8173.92	39
2018-07-29	8225.04	8225.04	8294.51	8294.51	8115.00	8115.00	8211.00	8211.00	25
2018-07-28	8188.57	8188.57	8246.54	8246.54	8067.00	8067.00	8225.04	8225.04	26

971 rows × 10 columns

```
# Keep the BTC data for "close (USD)"
btc_av = btc_av[['4a. close (USD)']]

# Rename the header names
btc_av.columns = ['Close']
btc_av.index.names = ['Date']

# Make sure data is in ascending order (oldest to latest)
btc_av = btc_av.iloc[::-1]
```

```
btc_av
```

	Close
Date	
2018-07-28	8225.04
2018-07-29	8211.00
2018-07-30	8173.92
2018-07-31	7730.93
2018-08-01	7604.58
...	...
2021-03-20	58102.28
2021-03-21	57351.56
2021-03-22	54083.25
2021-03-23	54340.89
2021-03-24	52303.65

971 rows × 1 columns

```
end_date = btc_av.index[0].date()
days = datetime.timedelta(1)
end_date = end_date - days
print("Extract prices till :", end_date)
```

```
Extract prices till : 2018-07-27
```

```
# Download BTC data from Yahoo Finance
btc_yf = yf.download('BTC-USD', end = end_date)
btc_yf
```

```
[*********************100%***********************]  1 of 1 completed
```

	Open	High	Low	Close	Adj Close	Volume
Date						
2014-09-17	465.864014	468.174011	452.421997	457.334015	457.334015	21056800

```
# Keep the BTC data for "Close" by dropping the other columns
btc_yf = btc_yf.drop(['Open', 'High', 'Low', 'Adj Close', 'Volume'], axis = 1)
btc_yf
```

	Close
Date	
2014-09-17	457.334015
2014-09-18	424.440002
2014-09-19	394.795990
2014-09-20	408.903992
2014-09-21	398.821014
...	...
2018-07-23	7711.109863
2018-07-24	8424.269531
2018-07-25	8181.390137
2018-07-26	7951.580078
2018-07-27	8165.009766

1410 rows × 1 columns

```
# Combine the two data sets (from Alpha Vantage and Yahoo Finance)
BTC = pd.concat([btc_yf, btc_av])
BTC
```

Date	Close
2014-09-17	457.334015
2014-09-18	424.440002
2014-09-19	394.795990

```
main_start = BTC.index[0].date()
main_end = BTC.index[-1].date()
print("Startd date: \t", main_start)
print("End date: \t", main_end)
```

```
Startd date:     2014-09-17
End date:        2021-03-24
```

2021-03-22	54083.250000

The data for all the other factors can now be collected using the yahoo finance api. This has to be done between the dates that were used for the BTC dataframe.

The data will be collected for:

- S&P500 index ('^GSPC')
- Dow Jones Industrial Average ('^DJI')
- NASDAQ Composite ('^IXIC')
- NYSE COMPOSITE ('^NYA')
- Nikkei ('^N225')
- S&P BSE SENSEX ('^BSESN')
- S&P/ASX 200 ('^AXJO')
- NIFTY 50 ('^NSEI')
- S&P/TSX Composite index ('^GSPTSE')
- KOSPI Composite Index (^KS11)

```
# Download other indices that could affect BTC
indices = yf.download(['^GSPC','^DJI','^IXIC','^NYA','^N225','^BSESN','^AXJO',
                        '^NSEI','^GSPTSE','^KS11'],
                       start=main_start, end=main_end)

# Keep the indices data for "Close"
indices = indices['Close']
indices
```

```
[*********************100%***********************]  10 of 10 completed
```

Date	^AXJO	^BSESN	^DJI	^GSPC	^GSPTSE	^IXIC
2014-09-17	5407.299805	26631.289062	17156.849609	2001.569946	15458.900391	4562.189941
2014-09-18	5415.799805	27112.210938	17265.990234	2011.359985	15465.500000	4593.430176
2014-09-19	5433.100098	27090.419922	17279.740234	2010.400024	15265.400391	4579.790039
2014-09-22	5363.000000	27206.740234	17172.679688	1994.290039	15129.000000	4527.689941
2014-	5415.700195	26775.689453	17055.869141	1982.770020	15125.700195	4508.689941

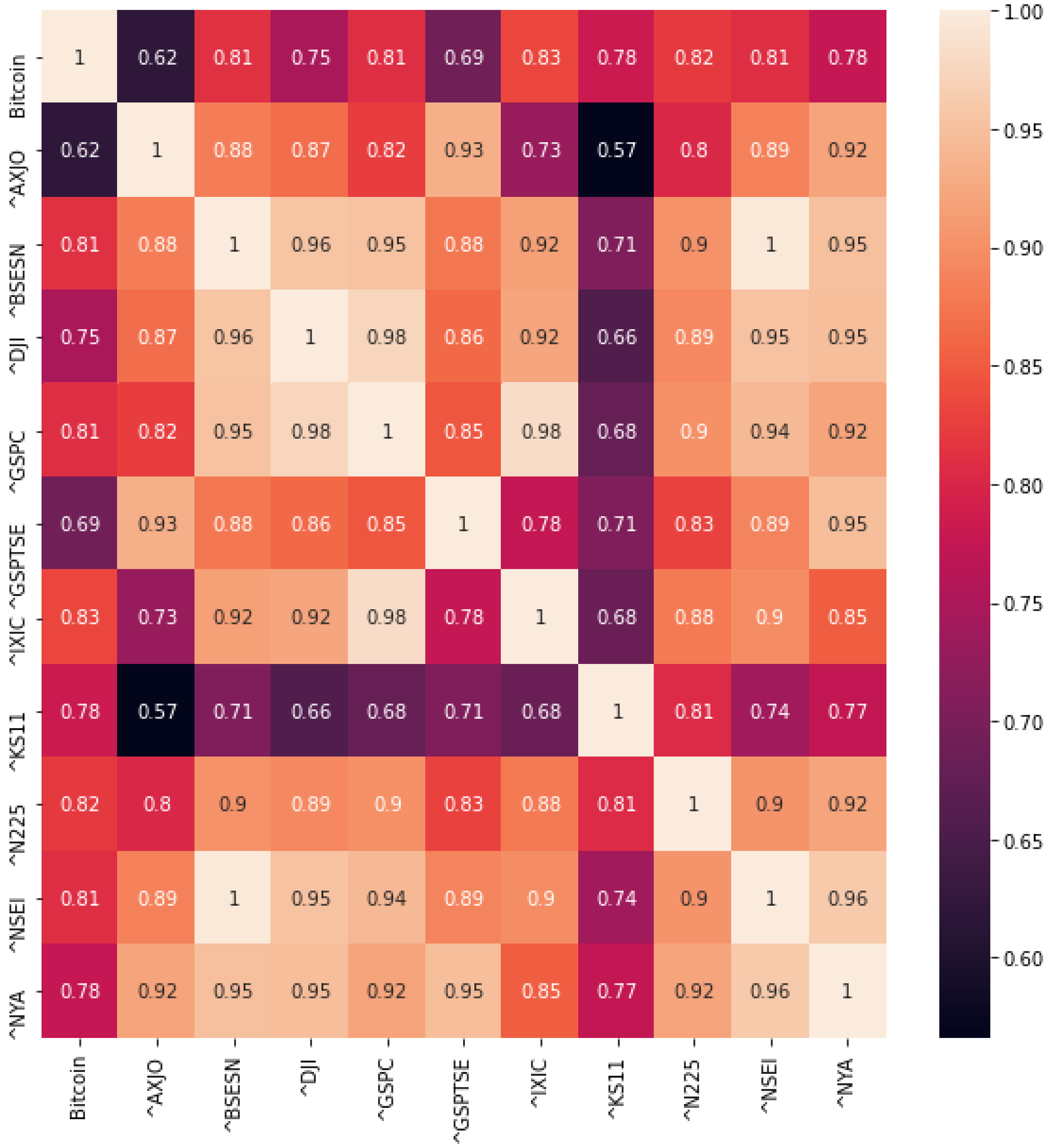
```
# Combine the BTC data with the other indices
combined = pd.concat([BTC,indices], join='inner', axis=1)
combined.columns = ['Bitcoin','^AXJO','^BSESN','^DJI','^GSPC','^GSPTSE','^IXIC',
                    '^KS11','^N225','^NSEI','^NYA']
combined
```

Date	Bitcoin	^AXJO	^BSESN	^DJI	^GSPC	^GSPTSE
2014-09-17	457.334015	5407.299805	26631.289062	17156.849609	2001.569946	15458.900391
2014-09-18	424.440002	5415.799805	27112.210938	17265.990234	2011.359985	15465.500000
2014-09-19	394.795990	5433.100098	27090.419922	17279.740234	2010.400024	15265.400391
2014-09-22	402.152008	5363.000000	27206.740234	17172.679688	1994.290039	15129.000000
2014-09-23	435.790985	5415.700195	26775.689453	17055.869141	1982.770020	15125.700195
...	...	...	...	...	...	...
2021-03-18	57648.160000	6745.899902	49216.519531	32862.300781	3915.459961	18836.500000
2021-03-19	58030.010000	6708.200195	49858.238281	32627.970703	3913.100098	18854.000000

Using correlation (heatmap) to choose stock indices

```
# Plot the heat map to find out which indices have a high correlation with BTC
plt.figure(figsize = (10,10))
matrix = combined.corr()
sb.heatmap(combined.corr(), annot=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f854656ddd0>
```




```
1 matrix
```

	Bitcoin	^AXJO	^BSESN	^DJI	^GSPC	^GSPTSE	^IXIC	^KS11

```
# Print out the indices that has a high correlation of 0.8 and above with BTC
for i in range(10):
    if (matrix.iloc[0][i] >= 0.8) and (matrix.iloc[0][i] != 1):
        print(matrix.columns[i])
```

```
^BSESN
^GSPC
^IXIC
^N225
^NSEI
```

The indices that have a correlation of above 0.8 are

1) BSESN
2) GSPC
3) IXIC
4) N225
5) NSEI

```
# Keep the indices that have a high correlation with BTC
combined = combined.drop(columns = ['^AXJO','^DJI','^GSPTSE','^NYA','^KS11'])
combined
```

Date	Bitcoin	^BSESN	^GSPC	^IXIC	^N225	^NSEI
2014-09-17	457.334015	26631.289062	2001.569946	4562.189941	15888.669922	7975.50000
2014-09-18	424.440002	27112.210938	2011.359985	4593.430176	16067.570312	8114.75000
2014-09-19	394.795990	27090.419922	2010.400024	4579.790039	16321.169922	8121.45019
2014-09-22	402.152008	27206.740234	1994.290039	4527.689941	16205.900391	8146.29980
2014-09-23	435.790985	26775.689453	1982.770020	4508.689941	NaN	8017.54980
...	...	...	...	...	...	...
2021-03-18	57648.160000	49216.519531	3915.459961	13116.169922	30216.750000	14557.84960
2021-03-19	58030.010000	49858.238281	3913.100098	13215.240234	29792.050781	14744.00000

```
# Clean the data frame here by dropping rows with value "NaN"
combined = combined.dropna()
combined
```

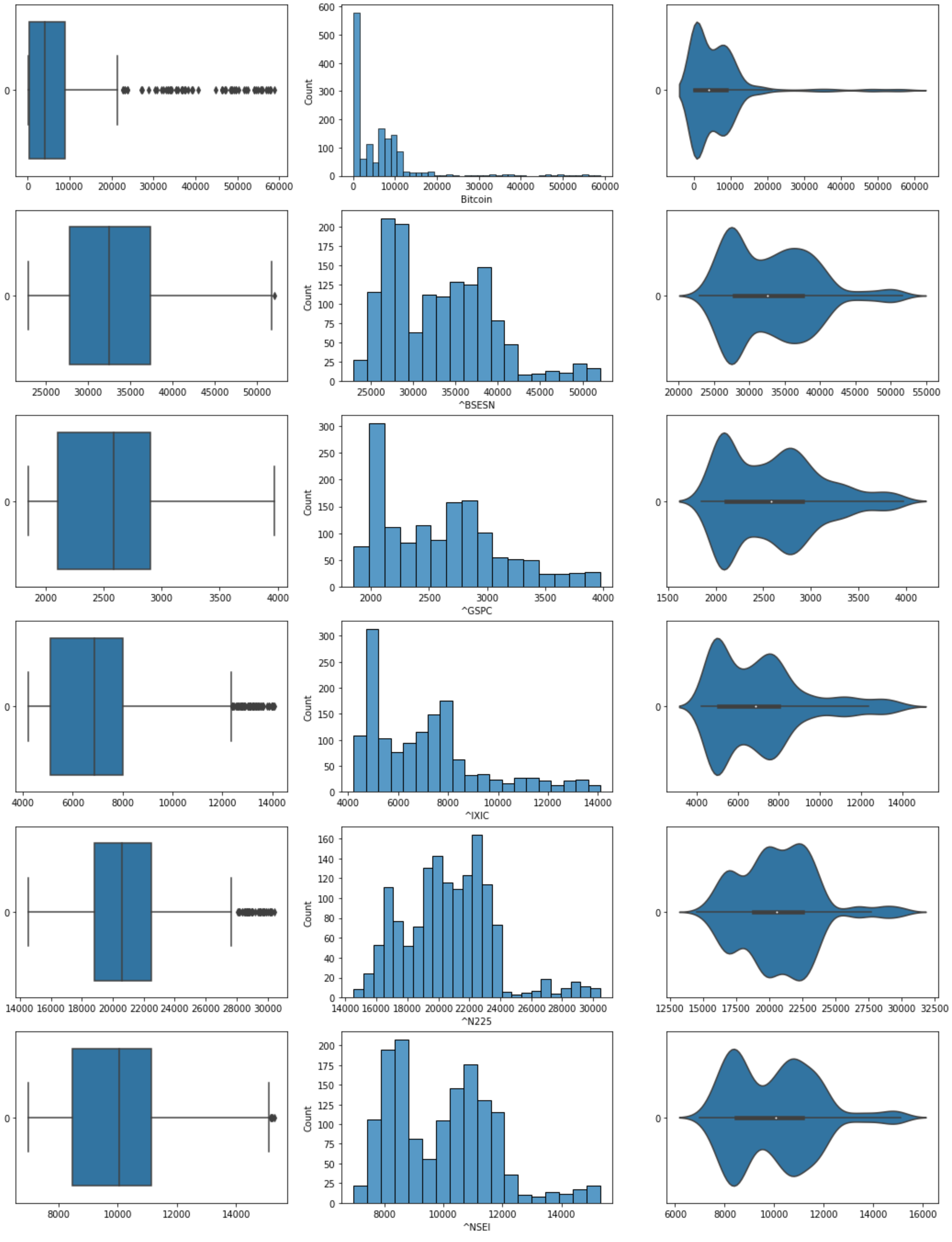
```
combined
```

Date	Bitcoin	^BSESN	^GSPC	^IXIC	^N225	^NSE
2014-09-17	457.334015	26631.289062	2001.569946	4562.189941	15888.669922	7975.50000
2014-09-18	424.440002	27112.210938	2011.359985	4593.430176	16067.570312	8114.75000
2014-09-19	394.795990	27090.419922	2010.400024	4579.790039	16321.169922	8121.45019
2014-09-22	402.152008	27206.740234	1994.290039	4527.689941	16205.900391	8146.29980
2014-09-24	423.204987	26744.689453	1998.300049	4555.220215	16167.450195	8002.39990
...	...	...	...	...	...	.
2021-03-17	58912.970000	49801.621094	3974.120117	13525.200195	29914.330078	14721.29980
2021-03-18	57648.160000	49216.519531	3915.459961	13116.169922	30216.750000	14557.84960

Basic Exploratory Analysis

Box plot, Histogram and Violin plot

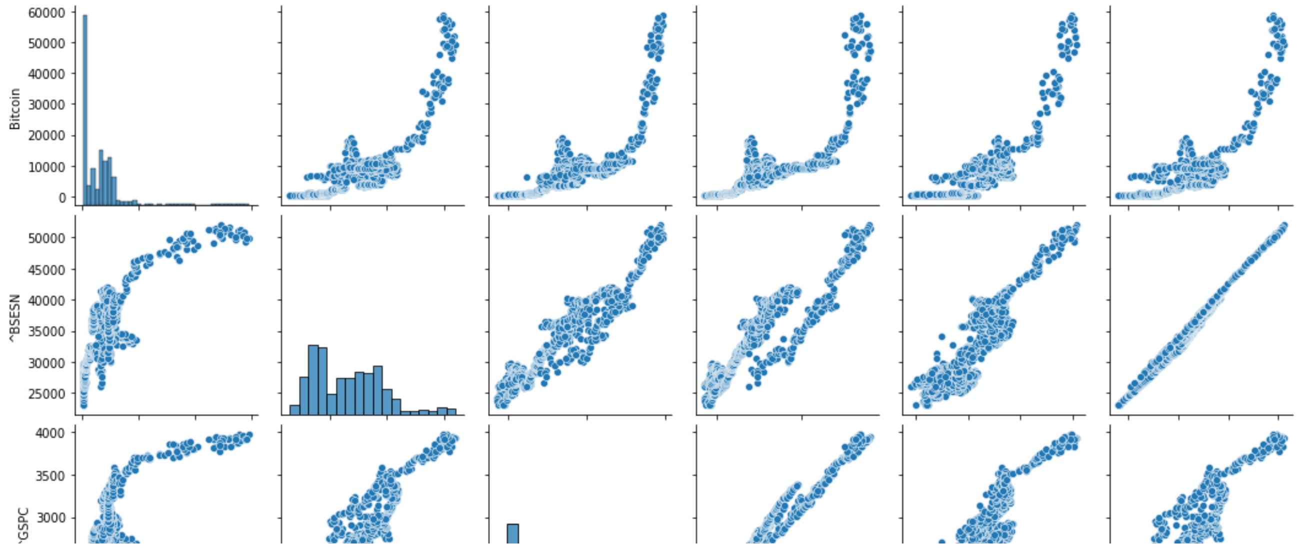
```
# Draw the distributions of all variables
f, axes = plt.subplots(6, 3, figsize=(18, 24))

# Plot the box plot, histogram and violin plot of the variables (including BTC)
count = 0
for var in combined:
    sb.boxplot(data = combined[var], orient = "h", ax = axes[count,0])
    sb.histplot(data = combined[var], ax = axes[count,1])
    sb.violinplot(data = combined[var], orient = "h", ax = axes[count,2])
    count += 1
```


Count
Bitcoin
^BSESN
^GSPC
^IXIC
^N225
^NSEI

Pair plot

```
# Plot the pair plot
sb.pairplot(data=combined)
```

```
<seaborn.axisgrid.PairGrid at 0x7f8545f249d0>
```



Heatmap

```
# Plot the heatmap
plt.figure(figsize = (10,10))
matrix = combined.corr()
sb.heatmap(combined.corr(), annot=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f853739de90>
```

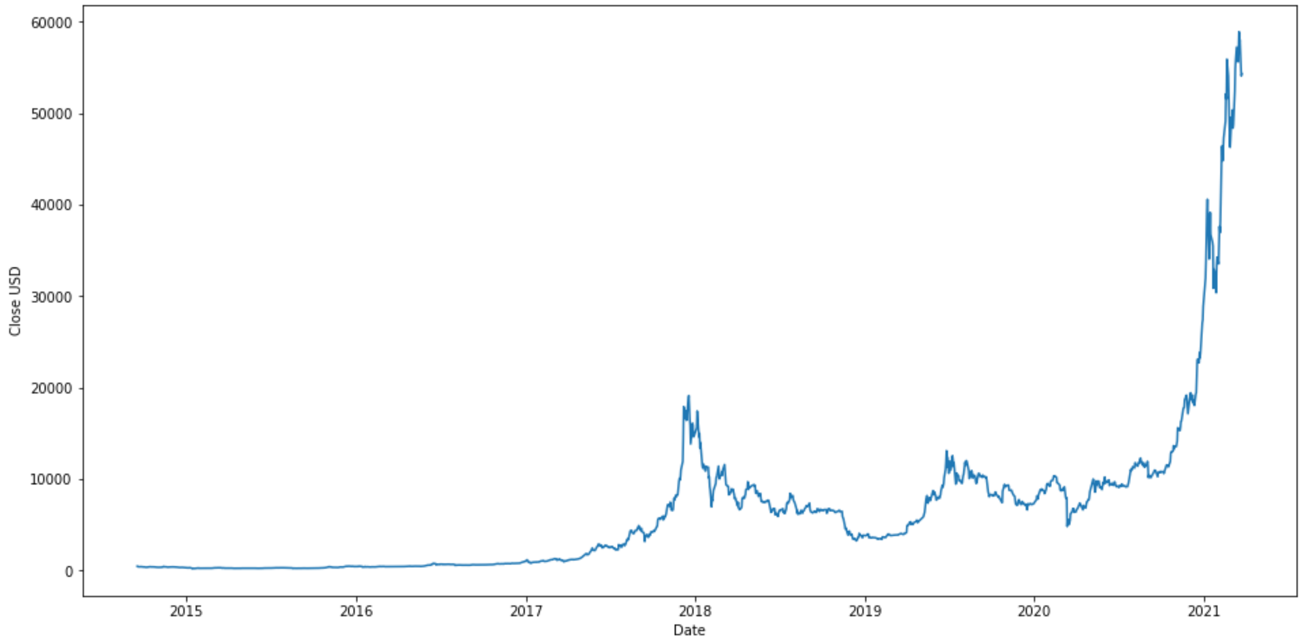
```
# This function is to gauge the accuracy of all predicted values
def accuracy(df):
    total = 0
    actual_values = df.iloc[:,0]
    predicted = df.iloc[:,1]
    for i in range(0, len(df)):
        print('Actual Value: $', actual_values[i], ' Predicted Value: $',
              predicted[i], ' Accuracy: ', (predicted[i]/actual_values[i])*100)
        total += (predicted[i]/actual_values[i])*100
    average = total/len(df)
    print('Average Accuracy over 7 Days: ',average)

# This function is to calculate the residual forecast error and mean forecast
# error
def RFE(df):
    forecast = []
    actual_values = df.iloc[:,0]
    predicted = df.iloc[:,1]
    for i in range(0, len(df)):
        difference = actual_values[i]-predicted[i]
        forecast.append(difference)
    print(forecast)
    bias = sum(forecast) * 1.0/len(df)
    print('Mean Forecast Error: %f' % bias)
```

Autocorrelation and Autoregression on Bitcoin

```
# Extract BTC data
BTC = pd.DataFrame(combined['Bitcoin'])
BTC
```

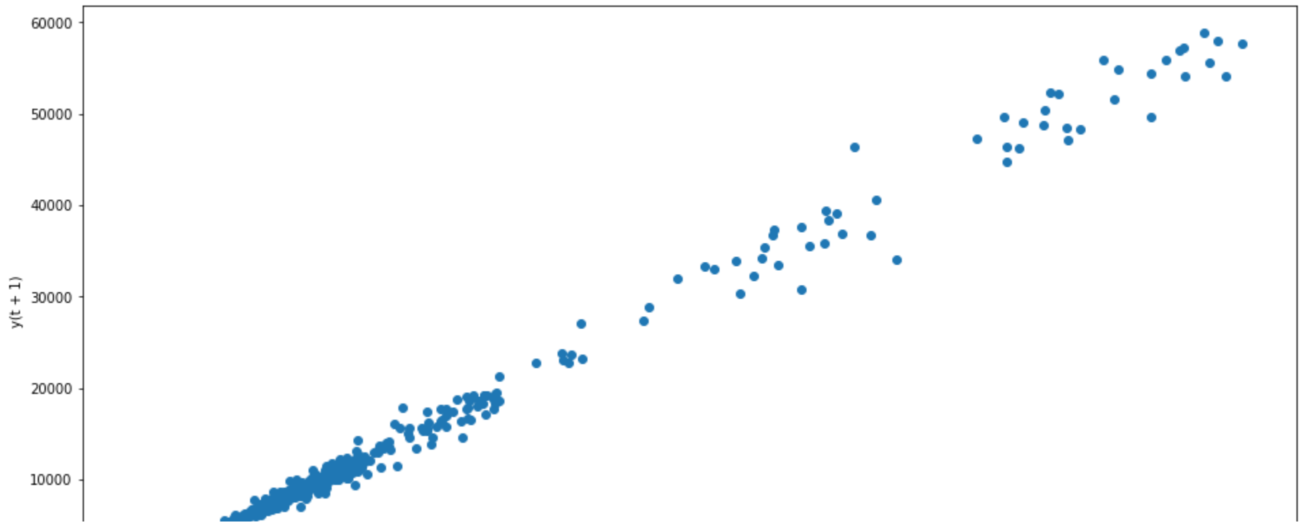
	Bitcoin
Date	
2014-09-17	457.334015

```
# Plot the time series of BTC
plt.figure(figsize = (16,8))
plt.plot(combined['Bitcoin'])
plt.xlabel('Date')
plt.ylabel('Close USD')
plt.show()
```




```
# Plot lag plot of BTC to determine if the values of BTC are random
plt.figure(figsize = (16,8))
lag_plot(BTC)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f853723cd50>
```




```
BTC_corr = pd.concat([BTC.shift(1),BTC], axis = 1)
BTC_corr.columns=['t-1','t+1']
BTC_corr.corr(method='pearson')
```

	t-1	t+1
t-1	1.000000	0.997532
t+1	0.997532	1.000000

```
X = BTC.values
test_size = 7
train, test = X[1:len(X)-test_size],X[len(X)-test_size:]
```

```
print(train.shape)
print(test.shape)
```

```
(1448, 1)
(7, 1)
```

```
# Fit to a UAR model
model = AR(train)
model_fit= model.fit()
```

```
variables = model_fit.k_ar
coefficients = model_fit.params
```

```
historical_data = train[len(train) - variables:]
historical_data
```

```
array([[36936.66],
       [38290.24],
       [46374.87],
       [46420.42],
       [44807.58],
       [47287.6 ],
       [49133.45],
```

```
       [52119.71],
       [51552.6 ],
       [55906.  ],
       [54087.67],
       [49676.2 ],
       [47073.73],
       [46276.87],
       [49587.03],
       [48440.65],
       [50349.37],
       [48374.09],
       [48751.71],
       [52375.17],
       [54884.5 ],
       [55851.59],
       [57221.72]])
```

```
historical_data = [historical_data[i] for i in range(len(historical_data))]
historical_data
```

```
[array([36936.66]),
 array([38290.24]),
 array([46374.87]),
 array([46420.42]),
 array([44807.58]),
 array([47287.6]),
 array([49133.45]),
 array([52119.71]),
 array([51552.6]),
 array([55906.]),
 array([54087.67]),
 array([49676.2]),
 array([47073.73]),
 array([46276.87]),
 array([49587.03]),
 array([48440.65]),
 array([50349.37]),
 array([48374.09]),
 array([48751.71]),
 array([52375.17]),
 array([54884.5]),
 array([55851.59]),
 array([57221.72])]
```

```
# Predict the values of BTC and store them in an array
predicted = []

for t in test:
    length = len(historical_data)
    lag = [historical_data[i] for i in range(length - variables, length)]
    y = coefficients[0]
    for d in range(variables):
        y += coefficients[d + 1] * lag[variables - d - 1]
    predicted.append(y)
    historical_data.append(t)
```

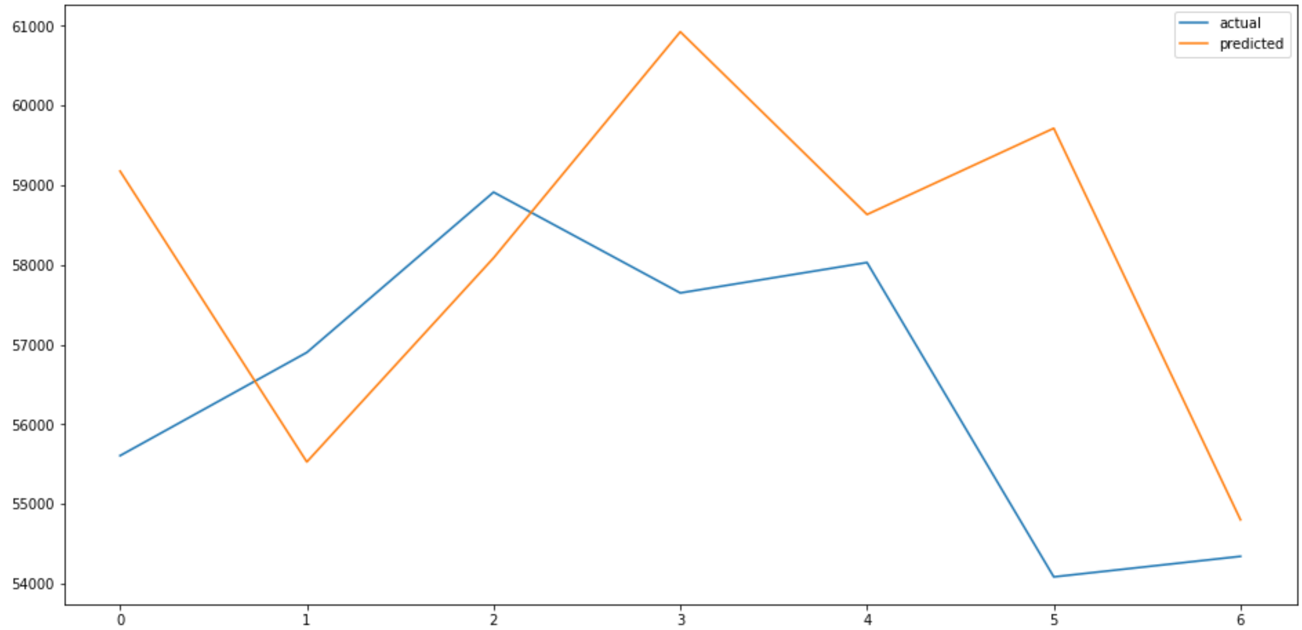
```
# RMSE of the Univariate autoregression
```

```
UAR_rmse = math.sqrt(mean_squared_error(test,predicted))
print("Root mean squared error for Uni-variate autoregression: $", UAR_rmse)
```

```
Root mean squared error for Uni-variate autoregression: $ 2887.6325006673856
```

```
# Plot the Univariate autoregression with the actual values of BTC
f = plt.figure(figsize=(16, 8))
plt.plot(test,label='actual')
plt.plot(predicted,label='predicted')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f85370fc7d0>
```




```
# Printing the accuracy for all the test dates.
test = pd.DataFrame(test)
predicted = pd.DataFrame(predicted)
combined_UAR = pd.concat([test, predicted], axis = 1)
accuracy(combined_UAR)
```

```
Actual Value: $ 55605.2  Predicted Value: $ 59177.147440611436  Accuracy:  106
Actual Value: $ 56900.75  Predicted Value: $ 55526.7975435793  Accuracy:  97.5
Actual Value: $ 58912.97  Predicted Value: $ 58087.574980603575  Accuracy:  98
Actual Value: $ 57648.16  Predicted Value: $ 60926.6208182039  Accuracy:  105.
Actual Value: $ 58030.01  Predicted Value: $ 58633.01445621121  Accuracy:  101
Actual Value: $ 54083.25  Predicted Value: $ 59715.15812162843  Accuracy:  110
Actual Value: $ 54340.89  Predicted Value: $ 54800.03115712002  Accuracy:  100
Average Accuracy over 7 Days:  102.94179314498862
```

```
#printing the forcasted error.
RFE(combined_UAR)
```

```
[-3571.9474406114387, 1373.9524564207022, 825.3950193964265, -3278.46081820389
Mean Forecast Error: -1620.730645
```

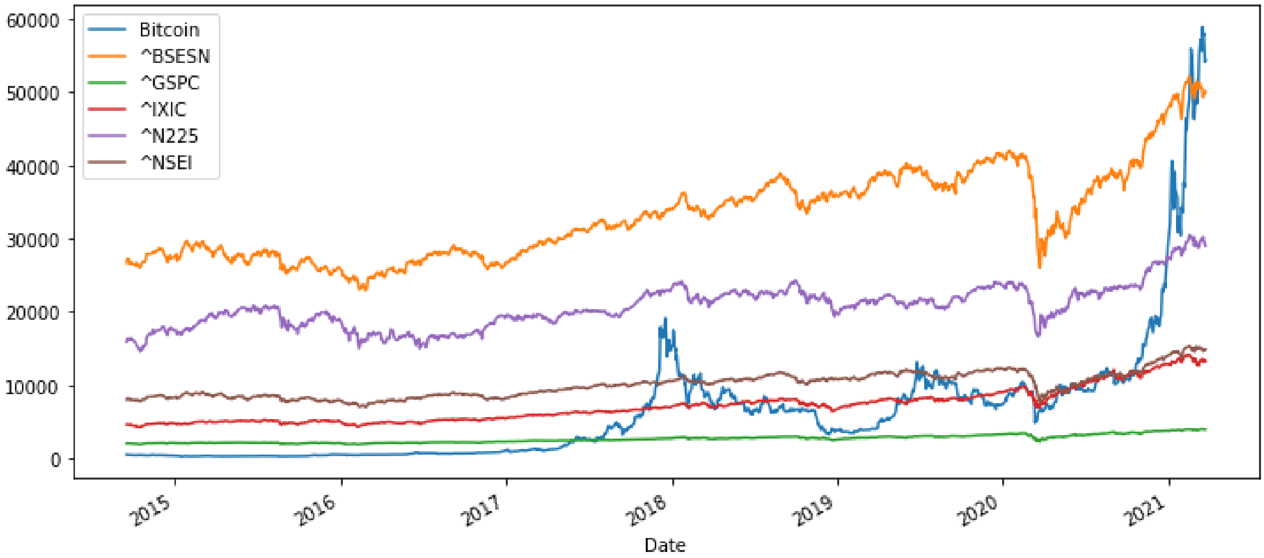
```
# Saving the predicted data frame for future analysis
UAR_pred = pd.DataFrame(index=combined['Bitcoin'].tail(7).index)
predicted.index = combined['Bitcoin'].tail(7).index
UAR_pred['predicted'] = predicted
UAR_pred
```

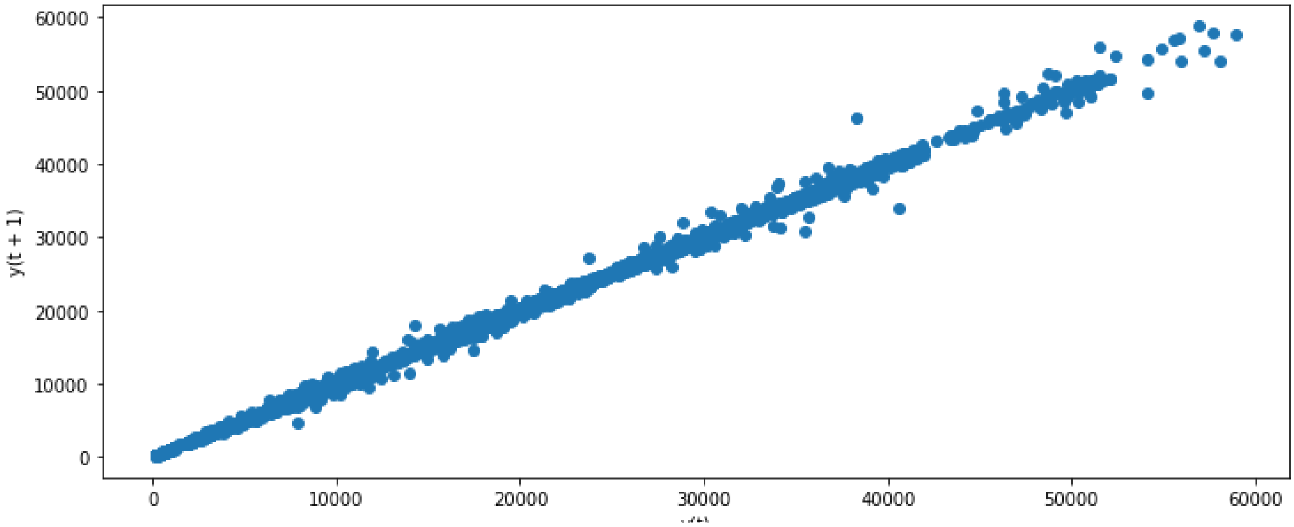
Date	predicted
2021-03-15	59177.147441
2021-03-16	55526.797544
2021-03-17	58087.574981
2021-03-18	60926.620818
2021-03-19	58633.014456
2021-03-22	59715.158122
2021-03-23	54800.031157

Multi-Variate Vector AutoRegression

Response Variable: BTC
Predictor Feature:
1) BSESN
2) GSPC
3) IXIC
4) N225
5) NSEI

```
# Plot the time series and lag plot of all the predictors, along with BTC,
# to determine if their values are random
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols = 1, figsize = (12, 12))
combined.plot(ax = ax1)
pd.plotting.lag_plot(combined)
plt.show()
```






```
def augmented_dickey_fuller_statistics(time_series):
    result = adfuller(time_series.values)
    print('ADF Statistic: %f' % result[0])
    print('p-value: %f' % result[1])
    print('Critical Values:')
    for key, value in result[4].items():
        print('\t%s: %.3f' % (key, value))
    print("\n")
```

```
# Create the train data set
X_train = combined.head(int(len(combined) - 7))
X_train
```

Date	Bitcoin	^BSESN	^GSPC	^IXIC	^N225	^NSE
2014-09-17	457.334015	26631.289062	2001.569946	4562.189941	15888.669922	7975.50000
2014-09-18	424.440002	27112.210938	2011.359985	4593.430176	16067.570312	8114.75000
2014-09-19	394.795990	27090.419922	2010.400024	4579.790039	16321.169922	8121.45019

```python
# Print the Augmented Dickey-Fuller Test for BTC and all predictors involved
print('Augmented Dickey-Fuller Test: Bitcoin Price Time Series')
augmented_dickey_fuller_statistics(X_train['Bitcoin'])

print('Augmented Dickey-Fuller Test: BSESN Price Time Series')
augmented_dickey_fuller_statistics(X_train['^BSESN'])

print('Augmented Dickey-Fuller Test: N225 Price Time Series')
augmented_dickey_fuller_statistics(X_train['^N225'])

print('Augmented Dickey-Fuller Test: GSPC Price Time Series')
augmented_dickey_fuller_statistics(X_train['^GSPC'])

print('Augmented Dickey-Fuller Test: IXIC Price Time Series')
augmented_dickey_fuller_statistics(X_train['^IXIC'])

print('Augmented Dickey-Fuller Test: NSEI Price Time Series')
augmented_dickey_fuller_statistics(X_train['^NSEI'])
```

```
Augmented Dickey-Fuller Test: Bitcoin Price Time Series
ADF Statistic: 4.207013
p-value: 1.000000
Critical Values:
	1%: -3.435
	5%: -2.864
	10%: -2.568


Augmented Dickey-Fuller Test: BSESN Price Time Series
ADF Statistic: 0.018911
p-value: 0.960068
Critical Values:
	1%: -3.435
	5%: -2.864
	10%: -2.568


Augmented Dickey-Fuller Test: N225 Price Time Series
ADF Statistic: -0.817593
p-value: 0.813900
Critical Values:
	1%: -3.435
	5%: -2.864
	10%: -2.568


Augmented Dickey-Fuller Test: GSPC Price Time Series
```

```
ADF Statistic: 0.317357
p-value: 0.978127
Critical Values:
        1%: -3.435
        5%: -2.864
        10%: -2.568


Augmented Dickey-Fuller Test: IXIC Price Time Series
ADF Statistic: 1.089007
p-value: 0.995122
Critical Values:
        1%: -3.435
        5%: -2.864
        10%: -2.568


Augmented Dickey-Fuller Test: NSEI Price Time Series
ADF Statistic: -0.183195
p-value: 0.940495
Critical Values:
        1%: -3.435
        5%: -2.864
        10%: -2.568
```

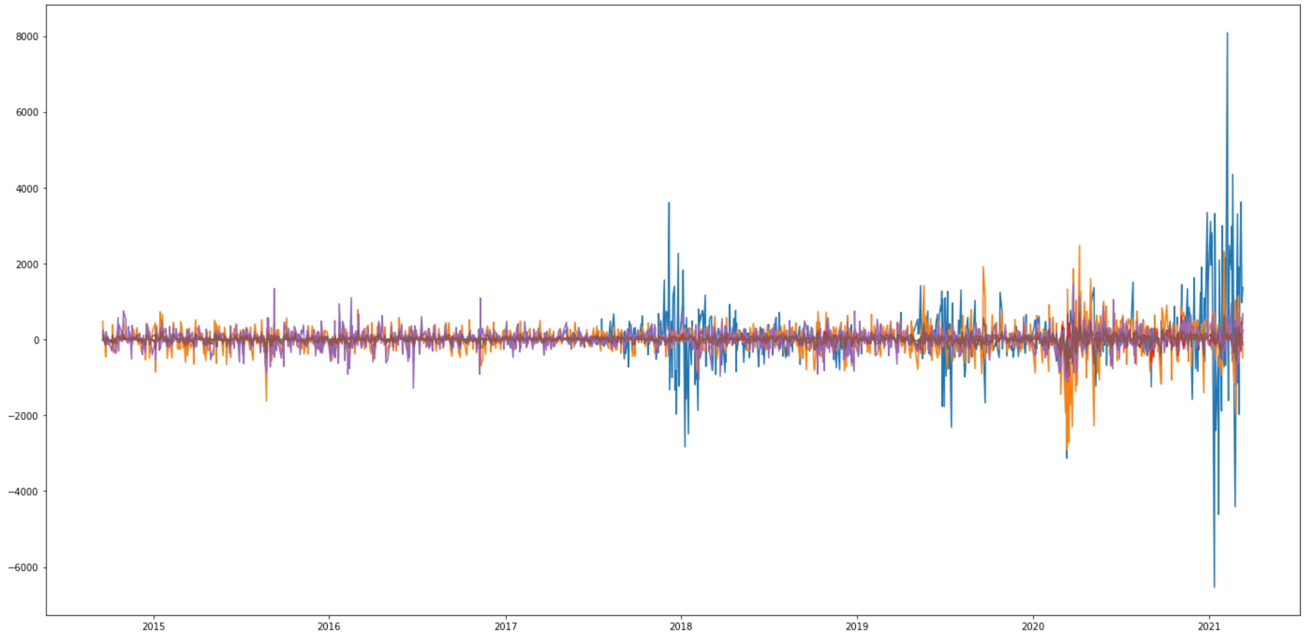
'p' values of less than 0.05 are needed to establish that a series is stationary. As seen above, none of the values are less than 0.05. Hence, to make the data stationary, the first difference was calculated.

```
# Create a copy of the train data set and find the differences between the curre
X_train = X_train.copy()
X_train_diff =(X_train).diff().dropna()
X_train_diff.describe()
```

	Bitcoin	^BSESN	^GSPC	^IXIC	^N225	^NSEI
count	1448.000000	1448.000000	1448.000000	1448.000000	1448.000000	1448.000000
mean	39.201924	16.685628	1.341001	6.048115	9.550525	4.872548
std	608.105180	382.054400	32.370222	107.268741	267.360989	112.788008
min	-6531.570000	-2919.257812	-324.890137	-970.290039	-1286.330078	-868.250000
25%	-30.090996	-152.040039	-8.177612	-26.392822	-115.082031	-44.575684
50%	2.319008	24.154297	1.804932	9.219971	13.674805	6.900391
75%	67.532500	211.768066	14.282532	49.447632	143.080078	63.449463
max	8084.630000	2476.261719	230.380127	673.080078	1454.281250	708.400391

```
plt.figure(figsize = (24,12))
plt.plot(X_train_diff)
```

```
[<matplotlib.lines.Line2D at 0x7f8536af9e10>,
 <matplotlib.lines.Line2D at 0x7f8536a01690>,
 <matplotlib.lines.Line2D at 0x7f8536a01990>,
 <matplotlib.lines.Line2D at 0x7f8536a01b50>,
 <matplotlib.lines.Line2D at 0x7f8536a01d10>,
 <matplotlib.lines.Line2D at 0x7f8536a01ed0>]
```




```
# Print the Augmented Dickey-Fuller Test for BTC and all predictors involved
print('Augmented Dickey-Fuller Test: Bitcoin Price Time Series')
augmented_dickey_fuller_statistics(X_train_diff['Bitcoin'])

print('Augmented Dickey-Fuller Test: BSESN Price Time Series')
augmented_dickey_fuller_statistics(X_train_diff['^BSESN'])

print('Augmented Dickey-Fuller Test: NSEI Price Time Series')
augmented_dickey_fuller_statistics(X_train_diff['^NSEI'])

print('Augmented Dickey-Fuller Test: IXIC Price Time Series')
augmented_dickey_fuller_statistics(X_train_diff['^IXIC'])

```

```
print('Augmented Dickey-Fuller Test: GSPC Price Time Series')
augmented_dickey_fuller_statistics(X_train_diff['^GSPC'])

print('Augmented Dickey-Fuller Test: N225 Price Time Series')
augmented_dickey_fuller_statistics(X_train_diff['^N225'])
```

```
Augmented Dickey-Fuller Test: Bitcoin Price Time Series
ADF Statistic: -3.877933
p-value: 0.002203
Critical Values:
	1%: -3.435
	5%: -2.864
	10%: -2.568


Augmented Dickey-Fuller Test: BSESN Price Time Series
ADF Statistic: -9.537425
p-value: 0.000000
Critical Values:
	1%: -3.435
	5%: -2.864
	10%: -2.568


Augmented Dickey-Fuller Test: NSEI Price Time Series
ADF Statistic: -9.495450
p-value: 0.000000
Critical Values:
	1%: -3.435
	5%: -2.864
	10%: -2.568


Augmented Dickey-Fuller Test: IXIC Price Time Series
ADF Statistic: -11.000730
p-value: 0.000000
Critical Values:
	1%: -3.435
	5%: -2.864
	10%: -2.568


Augmented Dickey-Fuller Test: GSPC Price Time Series
ADF Statistic: -8.697005
p-value: 0.000000
Critical Values:
	1%: -3.435
	5%: -2.864
	10%: -2.568


Augmented Dickey-Fuller Test: N225 Price Time Series
ADF Statistic: -25.064980
p-value: 0.000000
Critical Values:
	1%: -3.435
	5%: -2.864
	10%: -2.568
```

All the p-values now are less than 0.05. This implies that all the series now have been converted into stationary series

```
# Print the Granger Causality test for BTC and the predictors involved
print(grangercausalitytests(X_train_diff[['Bitcoin','^N225']], maxlag=15,
                            addconst=True, verbose=True))
print(grangercausalitytests(X_train_diff[['Bitcoin','^NSEI']], maxlag=15,
                            addconst=True, verbose=True))
print(grangercausalitytests(X_train_diff[['Bitcoin','^GSPC']], maxlag=15,
                            addconst=True, verbose=True))
print(grangercausalitytests(X_train_diff[['Bitcoin','^IXIC']], maxlag=15,
                            addconst=True, verbose=True))
print(grangercausalitytests(X_train_diff[['Bitcoin','^BSESN']], maxlag=15,
                            addconst=True, verbose=True))
```

```
likelihood ratio test: chi2=11.0916 , p=0.1966  , df=8
parameter F test:         F=1.3754  , p=0.2026  , df_denom=1423, df_num=8

Granger Causality
number of lags (no zero) 9
ssr based F test:         F=2.0117  , p=0.0348  , df_denom=1420, df_num=9
ssr based chi2 test:   chi2=18.3477 , p=0.0313  , df=9
likelihood ratio test: chi2=18.2317 , p=0.0326  , df=9
parameter F test:         F=2.0117  , p=0.0348  , df_denom=1420, df_num=9

Granger Causality
number of lags (no zero) 10
ssr based F test:         F=1.9623  , p=0.0339  , df_denom=1417, df_num=10
ssr based chi2 test:   chi2=19.9135 , p=0.0301  , df=10
likelihood ratio test: chi2=19.7769 , p=0.0314  , df=10
parameter F test:         F=1.9623  , p=0.0339  , df_denom=1417, df_num=10

Granger Causality
number of lags (no zero) 11
ssr based F test:         F=2.4695  , p=0.0046  , df_denom=1414, df_num=11
ssr based chi2 test:   chi2=27.6066 , p=0.0037  , df=11
likelihood ratio test: chi2=27.3448 , p=0.0041  , df=11
parameter F test:         F=2.4695  , p=0.0046  , df_denom=1414, df_num=11

Granger Causality
number of lags (no zero) 12
ssr based F test:         F=2.3747  , p=0.0050  , df_denom=1411, df_num=12
ssr based chi2 test:   chi2=29.0017 , p=0.0039  , df=12
likelihood ratio test: chi2=28.7128 , p=0.0043  , df=12
parameter F test:         F=2.3747  , p=0.0050  , df_denom=1411, df_num=12

Granger Causality
number of lags (no zero) 13
ssr based F test:         F=2.4508  , p=0.0027  , df_denom=1408, df_num=13
ssr based chi2 test:   chi2=32.4718 , p=0.0020  , df=13
likelihood ratio test: chi2=32.1098 , p=0.0023  , df=13
parameter F test:         F=2.4508  , p=0.0027  , df_denom=1408, df_num=13

Granger Causality
number of lags (no zero) 14
ssr based F test:         F=2.2604  , p=0.0048  , df_denom=1405, df_num=14
ssr based chi2 test:   chi2=32.2990 , p=0.0036  , df=14
likelihood ratio test: chi2=31.9407 , p=0.0041  , df=14
parameter F test:         F=2.2604  , p=0.0048  , df_denom=1405, df_num=14
```

```
parameter F test:         F=2.2604  , p=0.0048  , df_denom=1405, df_num=14

Granger Causality
number of lags (no zero) 15
ssr based F test:         F=2.2654  , p=0.0037  , df_denom=1402, df_num=15
ssr based chi2 test:   chi2=34.7317 , p=0.0027  , df=15
likelihood ratio test: chi2=34.3175 , p=0.0031  , df=15
parameter F test:         F=2.2654  , p=0.0037  , df_denom=1402, df_num=15
{1: ({'ssr_ftest': (2.8478192526565797e-06, 0.9986537649692334, 1444.0, 1), 's
       [0., 0., 0., 1., 0.]])]), 3: ({'ssr_ftest': (0.7034094517366991, 0.550C
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.]])]), 4: ({'ssr_ftest': (0.7089208605023866
       [0., 0., 0., 0., 0., 1., 0., 0., 0.],

       [0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0.]])]), 5: ({'ssr_ftest': (0.8816268C
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
```

```
# Create the test data set
X_test = combined.tail(7)
X_test
```

Date	Bitcoin	^BSESN	^GSPC	^IXIC	^N225	^NSEI
2021-03-15	55605.20	50395.078125	3968.939941	13459.709961	29766.970703	14929.500000
2021-03-16	56900.75	50363.960938	3962.709961	13471.570312	29921.089844	14910.450195
2021-03-17	58912.97	49801.621094	3974.120117	13525.200195	29914.330078	14721.299805
2021-03-18	57648.16	49216.519531	3915.459961	13116.169922	30216.750000	14557.849609
2021						

```
# Initiate the VAR model
model = VAR(endog=X_train_diff)
res = model.select_order(15)
res.summary()
```

```
/usr/local/lib/python3.7/dist-packages/statsmodels/tsa/base/tsa_model.py:219:
  ' ignored when e.g. forecasting.', ValueWarning)
```

VAR Order Selection (* highlights the minimums)

	AIC	BIC	FPE	HQIC
0	54.63	54.65	5.328e+23	54.64

```
# Fit to a VAR model
model_fit = model.fit(maxlags=3)
# Print a summary of the model results
model_fit.summary()
```

```
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Thu, 22, Apr, 2021
Time:                     13:16:30
--------------------------------------------------------------------
No. of Equations:         6.00000    BIC:                    54.6116
Nobs:                     1445.00    HQIC:                   54.3507
Log likelihood:          -51344.4    FPE:                3.44163e+23
AIC:                      54.1954    Det(Omega_mle):     3.18218e+23
--------------------------------------------------------------------
Results for equation Bitcoin
==============================================================================
                coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const             33.027319        15.994189            2.065           0.039
L1.Bitcoin         0.023892         0.026562            0.899           0.368
L1.^BSESN          0.398054         0.423188            0.941           0.347
L1.^GSPC          -1.304942         1.422924           -0.917           0.359
L1.^IXIC           0.406743         0.414498            0.981           0.326
L1.^N225           0.026202         0.072446            0.362           0.718
L1.^NSEI          -1.473847         1.431245           -1.030           0.303
L2.Bitcoin        -0.017897         0.026508           -0.675           0.500
L2.^BSESN          0.402963         0.423245            0.952           0.341
L2.^GSPC           2.722927         1.463920            1.860           0.063
L2.^IXIC          -0.841588         0.415645           -2.025           0.043
L2.^N225          -0.093042         0.073718           -1.262           0.207
L2.^NSEI          -0.944638         1.431508           -0.660           0.509
L3.Bitcoin         0.158954         0.026748            5.943           0.000
L3.^BSESN          0.319486         0.423689            0.754           0.451
L3.^GSPC          -1.325698         1.468123           -0.903           0.367
L3.^IXIC           0.094346         0.424937            0.222           0.824
L3.^N225          -0.011425         0.067387           -0.170           0.865
L3.^NSEI          -0.808132         1.431774           -0.564           0.572
==============================================================================

Results for equation ^BSESN
==============================================================================
                coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const              8.309752         9.777595            0.850           0.395
L1.Bitcoin        -0.005312         0.016238           -0.327           0.744
L1.^BSESN         -0.320956         0.258704           -1.241           0.215
L1.^GSPC           3.105867         0.869864            3.571           0.000
L1.^IXIC           0.004517         0.253391            0.018           0.986
L1.^N225          -0.030482         0.044288           -0.688           0.491
L1.^NSEI           0.756374         0.874951            0.864           0.387
```

```
L2.Bitcoin          0.050967         0.016205            3.145           0.002
L2.^BSESN           0.013570         0.258739            0.052           0.958
L2.^GSPC            0.597743         0.894926            0.668           0.504
L2.^IXIC            0.280094         0.254093            1.102           0.270
L2.^N225            0.113799         0.045065            2.525           0.012
L2.^NSEI            0.063507         0.875112            0.073           0.942
L3.Bitcoin          0.005406         0.016351            0.331           0.741
L3.^BSESN           0.071073         0.259010            0.274           0.784
L3.^GSPC           -3.703432         0.897495           -4.126           0.000
L3.^IXIC            0.757459         0.259773            2.916           0.004
L3.^N225            0.026089         0.041195            0.633           0.527
```

```
# Get the lag order
lag_order = model_fit.k_ar

# Input data for forecasting
input_data = X_train_diff.values[- lag_order:]

# Forecasting
predicted = model_fit.forecast(y=input_data, steps=7)
predicted = (pd.DataFrame(predicted, index=X_test.index, columns=X_test.columns

```

```
# Inverting transformation
def invert_transformation(X_train, pred_df):
    forecast = predicted.copy()
    columns = X_train.columns
    for col in columns:
        forecast[str(col)+'_pred'] = X_train[col].iloc[-1] + forecast[str(col)
        +'_pred'].cumsum()
    return forecast

output = invert_transformation(X_train, predicted)
print(output)
```

```
            Bitcoin_pred   ^BSESN_pred  ...    ^N225_pred    ^NSEI_pred
Date                                    ...
2021-03-15  57886.778587  51216.480194  ...  29982.507855  15154.302743
2021-03-16  57821.449031  51410.563381  ...  30108.377418  15210.792169
2021-03-17  58025.773888  51539.430236  ...  30144.444826  15247.732361
2021-03-18  58180.316752  51521.838544  ...  30146.639862  15244.323238
2021-03-19  58246.325608  51553.996860  ...  30145.945337  15253.394092
2021-03-22  58303.686446  51565.615457  ...  30166.175852  15257.489602
2021-03-23  58366.471748  51591.732699  ...  30171.410175  15264.775751

[7 rows x 6 columns]
```

```
# Check the predicted BTC values against the actual BTC values
merged = pd.concat([output['Bitcoin_pred'],combined['Bitcoin'].tail(7)], axis =
merged
```

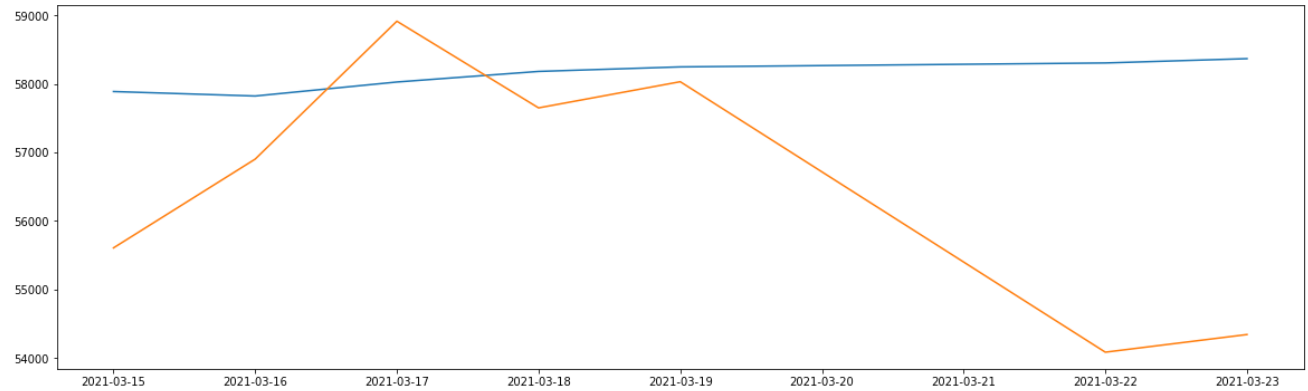
	Bitcoin_pred	Bitcoin
Date		
2021-03-15	57886.778587	55605.20
2021-03-16	57821.449031	56900.75
2021-03-17	58025.773888	58912.97
2021-03-18	58180.316752	57648.16

```
# RMSE of the Multi-Variate Vector AutoRegression
MSE = mean_squared_error(merged['Bitcoin'], merged['Bitcoin_pred'])
VAR_rmse = math.sqrt(MSE)
VAR_rmse
```

```
2425.6892969648397
```

```
# Plot the Multi-Variate Vector AutoRegression with the actual values of BTC
plt.figure(figsize=(20,6))
plt.plot(merged)
```

```
[<matplotlib.lines.Line2D at 0x7f8536903c10>,
 <matplotlib.lines.Line2D at 0x7f85368a7590>]
```




```
# Printing the accuracy for all the test dates.
accuracy(merged)
```

```
Actual Value: $ 57886.77858725814  Predicted Value: $ 55605.2  Accuracy:  96.0
Actual Value: $ 57821.44903124853  Predicted Value: $ 56900.75  Accuracy:  98.
Actual Value: $ 58025.77388752371  Predicted Value: $ 58912.97  Accuracy:  101
Actual Value: $ 58180.31675233638  Predicted Value: $ 57648.16  Accuracy:  99.
Actual Value: $ 58246.32560848445  Predicted Value: $ 58030.01  Accuracy:  99.
Actual Value: $ 58303.686446009946  Predicted Value: $ 54083.25  Accuracy:  92
Actual Value: $ 58366.47174846299  Predicted Value: $ 54340.89  Accuracy:  93.
Average Accuracy over 7 Days:  97.22476629259121
```

```
#printing the forcasted error.
RFE(merged)
```

```
[2281.57858725814, 920.6990312485286, -887.1961124762893, 532.1567523363774, 2
Mean Forecast Error: 1615.653152
```

```
VAR_pred = merged['Bitcoin_pred'].copy()
```

```
VAR_pred
```

```
Date
2021-03-15    57886.778587
2021-03-16    57821.449031
2021-03-17    58025.773888
2021-03-18    58180.316752
2021-03-19    58246.325608
2021-03-22    58303.686446
2021-03-23    58366.471748
Name: Bitcoin_pred, dtype: float64
```

VAR uses past data to predict the next value but that prediction is for the immediate value for the next entry. This was seen when the team tried out different combinations of testing data entries to find that the first value was always predicted very accurately but the next values were not as close. The team decided to use Long Short Term Memory (LSTM), a form of Artificial Neural Netwrok to predict values in a similar way using stock indices. This decision was taken because the overlap of the time series with one another was clearly seen in the tests conducted before developing the Vector Autoregression Model. Also, LSTM uses feedback loops and the data will be provided in a way such that the actual price for bitcoin/ other predictors a day before the test date will always be provided.

```

```

Long short-term memory ANN

Uni-variate LSTM

```
# Find the length of the training data for the Univariate LSTM
training_data_len = len(combined) - 7
training_data_len
```

```
1449
```

```
# Scale the data for bitcoin
scaler = MinMaxScaler(feature_range=(0,1))
BTC_arr = np.array(combined[['Bitcoin']])
scaled_data = scaler.fit_transform(BTC_arr)
```

```
scaled_data = scaler.fit_transform(BTC_arr)
scaled_data
```

```
array([[0.00475409],
       [0.00419405],
       [0.00368934],
       ...,
       [0.98496702],
       [0.91777082],
       [0.92215731]])
```

```
# Creating the training dataset
training_data = scaled_data[:training_data_len, :]

# Split the data into x_train and y_train
x_train = []
y_train = []

for i in range(30, len(training_data)):
    x_train.append(training_data[i-30:i,0])
    y_train.append(training_data[i, 0])
```

```
# Convert x_train and y_train into numpy arrays
x_train, y_train = np.array(x_train), np.array(y_train)
```

```
# Reshape the data
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
x_train.shape
```

```
(1419, 30, 1)
```

```
# Build the LSTM model
model = Sequential()
model.add(LSTM(50, return_sequences=True, input_shape = (x_train.shape[1], 1)))
model.add(LSTM(50, return_sequences=False))
model.add(Dense(25))
model.add(Dense(1))
```

```
# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')
```

```
# Train the model
model.fit(x_train, y_train, batch_size=64, epochs=100)
```

```
23/23 [==============================] - 1s 36ms/step - loss: 1.2164e-04
Epoch 54/100
23/23 [==============================] - 1s 37ms/step - loss: 1.0769e-04
Epoch 55/100
23/23 [==============================] - 1s 37ms/step - loss: 1.2146e-04
Epoch 56/100
23/23 [==============================] - 1s 35ms/step - loss: 1.4485e-04
Epoch 57/100
23/23 [==============================] - 1s 36ms/step - loss: 1.4620e-04
Epoch 58/100
```

```
23/23 [==============================] - 1s 36ms/step - loss: 1.5532e-04
Epoch 59/100
23/23 [==============================] - 1s 37ms/step - loss: 1.2020e-04
Epoch 60/100
23/23 [==============================] - 1s 36ms/step - loss: 1.0823e-04
Epoch 61/100
23/23 [==============================] - 1s 36ms/step - loss: 1.0615e-04
Epoch 62/100
23/23 [==============================] - 1s 35ms/step - loss: 1.1478e-04
Epoch 63/100
23/23 [==============================] - 1s 36ms/step - loss: 1.0897e-04
Epoch 64/100
23/23 [==============================] - 1s 36ms/step - loss: 1.2288e-04
Epoch 65/100
23/23 [==============================] - 1s 37ms/step - loss: 1.4660e-04
Epoch 66/100

23/23 [==============================] - 1s 37ms/step - loss: 1.2737e-04
Epoch 67/100
23/23 [==============================] - 1s 37ms/step - loss: 1.0186e-04
Epoch 68/100
23/23 [==============================] - 1s 38ms/step - loss: 1.5354e-04
Epoch 69/100
23/23 [==============================] - 1s 35ms/step - loss: 1.7294e-04
Epoch 70/100
23/23 [==============================] - 1s 36ms/step - loss: 1.7084e-04
Epoch 71/100
23/23 [==============================] - 1s 36ms/step - loss: 1.2131e-04
Epoch 72/100
23/23 [==============================] - 1s 37ms/step - loss: 1.1328e-04
Epoch 73/100
23/23 [==============================] - 1s 38ms/step - loss: 1.0102e-04
Epoch 74/100
23/23 [==============================] - 1s 36ms/step - loss: 1.3591e-04
Epoch 75/100
23/23 [==============================] - 1s 36ms/step - loss: 1.2003e-04
Epoch 76/100
23/23 [==============================] - 1s 36ms/step - loss: 1.1044e-04
Epoch 77/100
23/23 [==============================] - 1s 36ms/step - loss: 1.1407e-04
Epoch 78/100
23/23 [==============================] - 1s 36ms/step - loss: 1.0592e-04
Epoch 79/100
23/23 [==============================] - 1s 36ms/step - loss: 1.0657e-04
Epoch 80/100
23/23 [==============================] - 1s 37ms/step - loss: 1.0521e-04
Epoch 81/100
23/23 [==============================] - 1s 36ms/step - loss: 1.0620e-04
Epoch 82/100
23/23 [==============================] - 1s 35ms/step - loss: 1.1132e-04
```

```python
# Create the testing dataset
test_data = scaled_data[training_data_len - 30:, :]

# Create testing datasets: x_test, y_test
x_test = []
y_test = BTC_arr[training_data_len:,:]

for i in range(30, len(test_data)):
    x_test.append(test_data[i-30:i,0])
```

```
# Convert data to numpy array
x_test = np.array(x_test)
```

```
# Reshape the data
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))
```

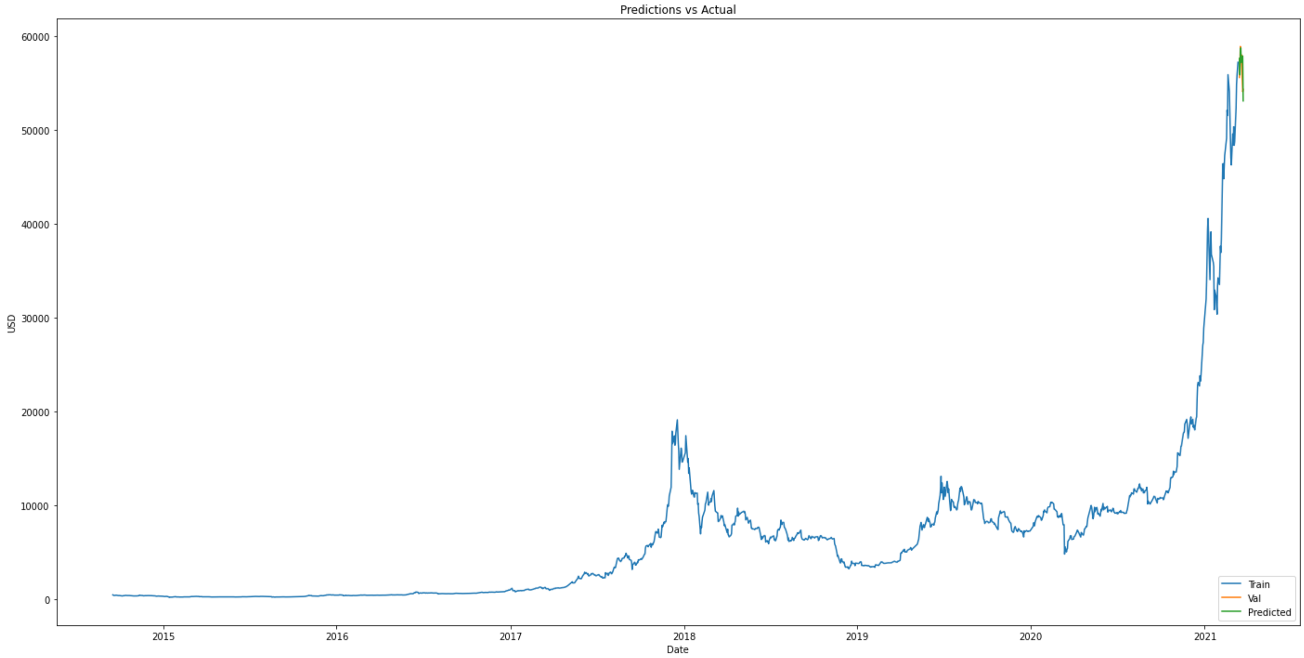
```
# Get the models predicted price value
predicted = model.predict(x_test)
# Unscaling values to obtain actual values instead of values between [0,1]
predicted = scaler.inverse_transform(predicted)
```

```
# RMSE of the Univariate LSTM
ULSTM_rmse = np.sqrt(np.mean(predicted - y_test)**2)
ULSTM_rmse
```

```
417.0938169642853
```

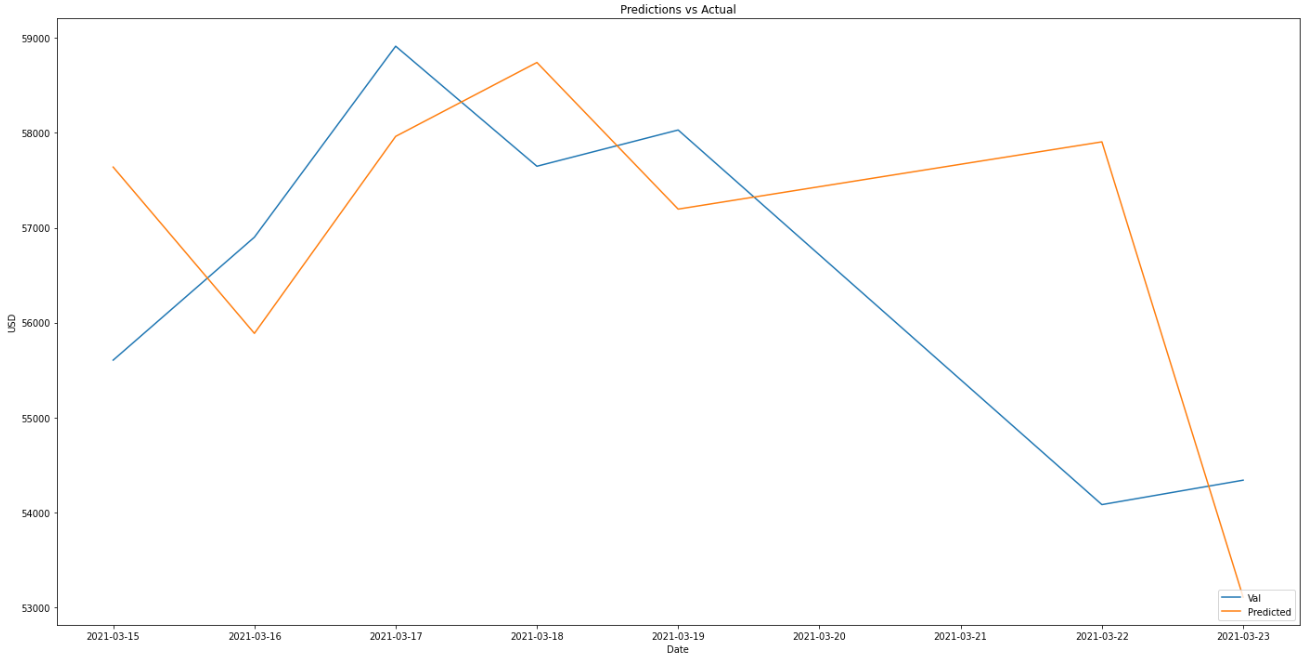
```
data = combined.filter(['Bitcoin'])
# Plot the data
train = data[:training_data_len]
valid = data[training_data_len:]
valid['Predicted'] = predicted
# Visualization
plt.figure(figsize = (24,12))
plt.title('Predictions vs Actual')
plt.xlabel('Date')
plt.ylabel('USD')
plt.plot(train['Bitcoin'])
plt.plot(valid[['Bitcoin','Predicted']])
plt.legend(['Train','Val', 'Predicted'], loc = 'lower right')
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5: SettingWithCop
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
  """
<matplotlib.legend.Legend at 0x7f852f09bd90>
```




```
data = combined.filter(['Bitcoin'])
# Plot the data
valid = data[training_data_len:]
valid['Predicted'] = predicted
# Visualize the data
plt.figure(figsize = (24,12))
plt.title('Predictions vs Actual')
plt.xlabel('Date')
plt.ylabel('USD')
plt.plot(valid[['Bitcoin','Predicted']])
plt.legend(['Val', 'Predicted'], loc = 'lower right')
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4: SettingWithCop
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
  after removing the cwd from sys.path.
<matplotlib.legend.Legend at 0x7f852f101590>
```




```
# Printing the accuracy for all the test dates.
accuracy(valid[['Bitcoin', 'Predicted']])
```

```
Actual Value: $ 55605.2  Predicted Value: $ 57639.688  Accuracy:  103.6588079€
Actual Value: $ 56900.75  Predicted Value: $ 55887.336  Accuracy:  98.21897942
Actual Value: $ 58912.97  Predicted Value: $ 57962.75  Accuracy:  98.387078431
Actual Value: $ 57648.16  Predicted Value: $ 58741.3  Accuracy:  101.896228398
Actual Value: $ 58030.01  Predicted Value: $ 57196.664  Accuracy:  98.5639396
Actual Value: $ 54083.25  Predicted Value: $ 57905.242  Accuracy:  107.0668685
Actual Value: $ 54340.89  Predicted Value: $ 53107.906  Accuracy:  97.73102032
Average Accuracy over 7 Days:  100.78898897474575
```

```
#printing the forcasted error.
RFE(valid[['Bitcoin', 'Predicted']])
```

```
[-2034.487500000003, 1013.4140625, 950.2200000000012, -1093.1407812499965, 833
Mean Forecast Error: -417.093817
```

```
ULSTM_pred = valid['Predicted'].copy()
```

Multi-variate LSTM

```
# Using Multi-Variate LSTM
training_data_multi_len = len(combined) - 7
training_data_multi_len
```

```
1449
```

```
combined_without_date = combined[['Bitcoin','^BSESN','^GSPC','^IXIC','^N225','^N

scaler = MinMaxScaler(feature_range=(0,1))
df_without_date = np.array(combined_without_date)
scaled_data_multi = scaler.fit_transform(df_without_date)
scaled_data_multi
```

```
array([[0.00475409, 0.12548982, 0.0705427 , 0.03490555, 0.08510447,
        0.12045043],
       [0.00419405, 0.14200036, 0.07515573, 0.03806813, 0.09633118,
        0.13714137],
       [0.00368934, 0.14125225, 0.0747034 , 0.03668728, 0.11224557,
        0.13794448],
       ...,
       [0.98496702, 0.9228948 , 0.97124762, 0.91089063, 0.95759717,
        0.93174393],
       [0.91777082, 0.91990974, 0.98420079, 0.92732092, 0.91882146,
        0.93083301],
       [0.92215731, 0.92952766, 0.9700319 , 0.912152  , 0.90763678,
        0.94022424]])
```

```
# Creating the training dataset
training_data_multi = scaled_data_multi[:training_data_multi_len, :]

# Split the data into x_train and y_train
x_train = []
y_train = []

for i in range(30, len(training_data)):
    x_train.append(training_data_multi[i-30:i,:])
    y_train.append(training_data_multi[i, 0])
```

```
# Convert x_train and y_train into numpy arrays
x_train, y_train = np.array(x_train), np.array(y_train)
```

```
# Reshape the data
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 6))
x_train.shape
```

```
(1419, 30, 6)
```

```
# Build the LSTM model
model_multi = Sequential()
model_multi.add(LSTM(50, return_sequences=True, input_shape = (x_train.shape[1],
model_multi.add(LSTM(50, return_sequences=False))
model_multi.add(Dense(25))
model_multi.add(Dense(1))
```

```
# Compile the model
model_multi.compile(optimizer='adam', loss='mean_squared_error')
```

```
# Train the model
model_multi.fit(x_train, y_train, batch_size=512, epochs=50)
```

```
Epoch 1/50
3/3 [==============================] - 0s 160ms/step - loss: 3.2418e-04
Epoch 2/50
3/3 [==============================] - 1s 163ms/step - loss: 3.2392e-04
Epoch 3/50
3/3 [==============================] - 0s 159ms/step - loss: 3.2068e-04
Epoch 4/50
3/3 [==============================] - 0s 155ms/step - loss: 3.1866e-04
Epoch 5/50
3/3 [==============================] - 0s 159ms/step - loss: 3.1862e-04
Epoch 6/50
3/3 [==============================] - 1s 158ms/step - loss: 3.1900e-04
Epoch 7/50
3/3 [==============================] - 1s 161ms/step - loss: 3.1846e-04
Epoch 8/50
3/3 [==============================] - 1s 166ms/step - loss: 3.1929e-04
Epoch 9/50
3/3 [==============================] - 0s 159ms/step - loss: 3.1864e-04
Epoch 10/50
3/3 [==============================] - 1s 166ms/step - loss: 3.1680e-04
Epoch 11/50
3/3 [==============================] - 1s 166ms/step - loss: 3.1656e-04
Epoch 12/50
3/3 [==============================] - 1s 163ms/step - loss: 3.0950e-04
Epoch 13/50
3/3 [==============================] - 0s 160ms/step - loss: 3.0784e-04
Epoch 14/50
3/3 [==============================] - 1s 157ms/step - loss: 3.0533e-04
Epoch 15/50
3/3 [==============================] - 1s 162ms/step - loss: 3.0767e-04
Epoch 16/50
3/3 [==============================] - 1s 166ms/step - loss: 3.2263e-04
Epoch 17/50
3/3 [==============================] - 1s 168ms/step - loss: 3.2055e-04
Epoch 18/50
3/3 [==============================] - 1s 163ms/step - loss: 2.9655e-04
Epoch 19/50
3/3 [==============================] - 1s 167ms/step - loss: 3.0517e-04
Epoch 20/50
3/3 [==============================] - 0s 157ms/step - loss: 3.1768e-04
Epoch 21/50
3/3 [==============================] - 1s 165ms/step - loss: 3.3193e-04
Epoch 22/50
3/3 [==============================] - 0s 162ms/step - loss: 3.6942e-04
Epoch 23/50
3/3 [==============================] - 1s 167ms/step - loss: 3.1663e-04
Epoch 24/50
3/3 [==============================] - 1s 158ms/step - loss: 2.9118e-04
Epoch 25/50
3/3 [==============================] - 1s 169ms/step - loss: 2.8556e-04
Epoch 26/50
3/3 [==============================] - 0s 154ms/step - loss: 2.8131e-04
Epoch 27/50
3/3 [==============================] - 1s 164ms/step - loss: 2.8912e-04
Epoch 28/50
3/3 [==============================] - 1s 165ms/step - loss: 2.7585e-04
Epoch 29/50
```

```
3/3 [==============================] - 1s 166ms/step - loss: 2.8094e-04
Epoch 30/50
```

```
# Create the testing dataset
test_data_multi = scaled_data_multi[training_data_multi_len - 30:, :]

# Create testing datasets: x_test, y_test
x_test = []
y_test = BTC_arr[training_data_multi_len:,:]

for i in range(30, len(test_data_multi)):
    x_test.append(test_data_multi[i-30:i,:])
```

```
# Convert data to numpy array
x_test = np.array(x_test)
```

```
# Reshape the data
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 6))
```

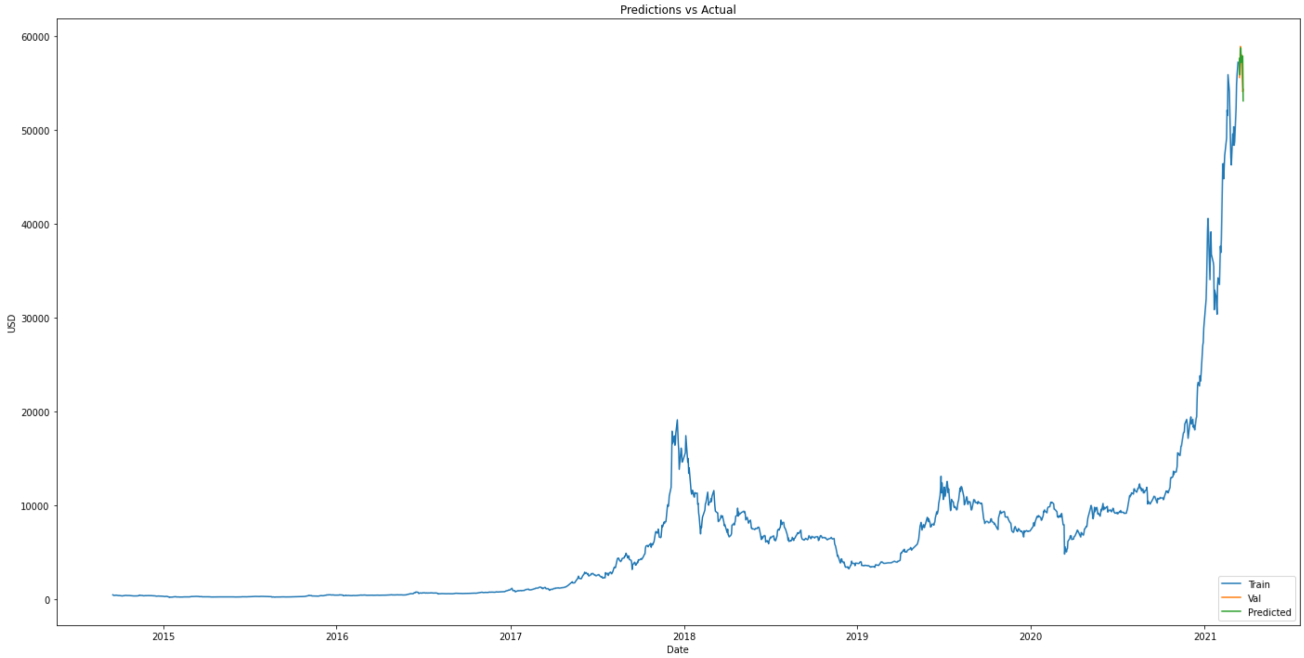
```
# Get the models predicted price value
predicted_multi = model_multi.predict(x_test)
forecast_multi = np.repeat(predicted_multi, df_without_date.shape[1], axis = -1)
# Unscaling values to obtain actual values instead of values between [0,1]
predicted_multi = scaler.inverse_transform(forecast_multi)[:,0]
```

```
# Getting root mean squared error (RMSE)
MLSTM_rmse = np.sqrt(np.mean(predicted_multi - y_test)**2)
print(MLSTM_rmse)
```

```
148.57303571428616
```

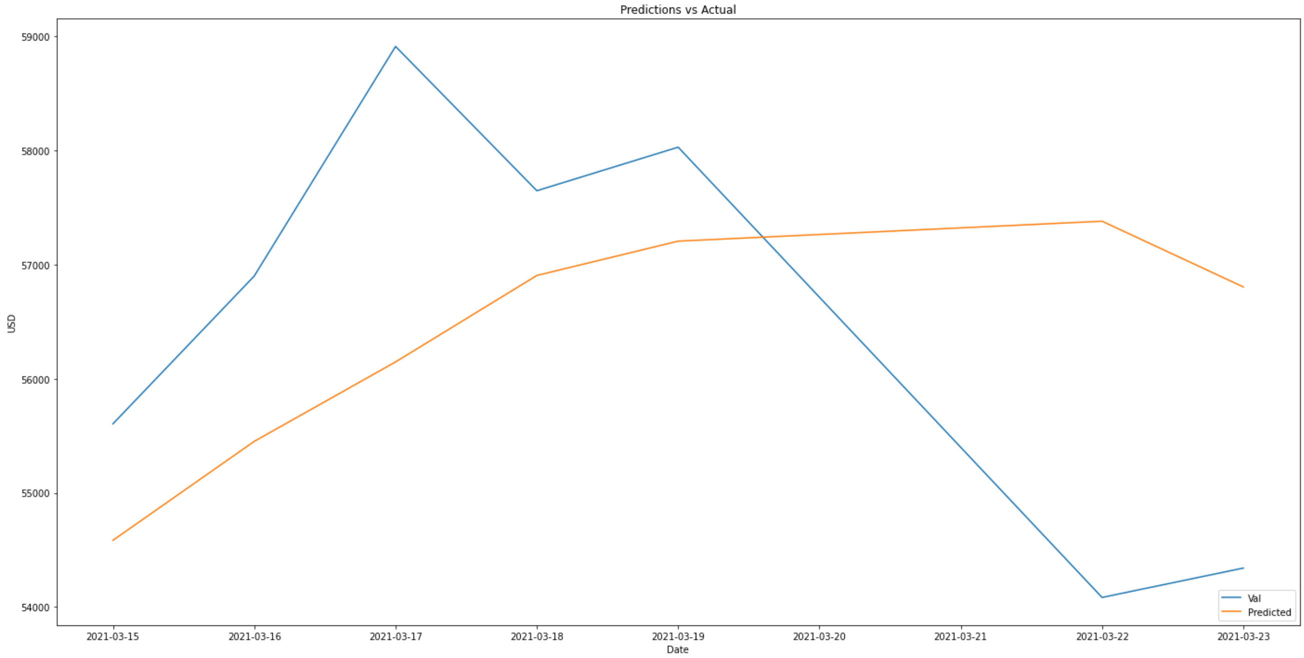
```
data = combined.filter(['Bitcoin'])
# Plot the data
train = data[:training_data_len]
valid = data[training_data_len:]
valid['Predicted'] = predicted
# Visualize the data
plt.figure(figsize = (24,12))
plt.title('Predictions vs Actual')
plt.xlabel('Date')
plt.ylabel('USD')
plt.plot(train['Bitcoin'])
plt.plot(valid[['Bitcoin','Predicted']])
plt.legend(['Train','Val', 'Predicted'], loc = 'lower right')
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5: SettingWithCop
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
  """
<matplotlib.legend.Legend at 0x7f852f0f9b50>
```




```
data = combined.filter(['Bitcoin'])
# Plot the data
valid2 = data[training_data_len:]
valid2['Predicted'] = predicted_multi
# Visualize the data
plt.figure(figsize = (24,12))
plt.title('Predictions vs Actual')
plt.xlabel('Date')
plt.ylabel('USD')
plt.plot(valid2[['Bitcoin','Predicted']])
plt.legend(['Val', 'Predicted'], loc = 'lower right')
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4: SettingWithCop
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/st
  after removing the cwd from sys.path.
<matplotlib.legend.Legend at 0x7f852d451d50>
```




```
1 # Printing the accuracy for all the test dates.
2 accuracy(valid2[['Bitcoin', 'Predicted']])
```

```
Actual Value: $ 55605.2  Predicted Value: $ 54584.406  Accuracy:  98.16421171C
Actual Value: $ 56900.75  Predicted Value: $ 55451.484  Accuracy:  97.4529938C
Actual Value: $ 58912.97  Predicted Value: $ 56147.855  Accuracy:  95.3064418C
Actual Value: $ 57648.16  Predicted Value: $ 56905.62  Accuracy:  98.71194690S
Actual Value: $ 58030.01  Predicted Value: $ 57206.344  Accuracy:  98.5806201E
Actual Value: $ 54083.25  Predicted Value: $ 57380.777  Accuracy:  106.0971323
Actual Value: $ 54340.89  Predicted Value: $ 56804.73  Accuracy:  104.5340451I
Average Accuracy over 7 Days:  99.83534170075141
```

```
1 RFE(valid2[['Bitcoin', 'Predicted']])
```

```
[1020.7937499999971, 1449.265625, 2765.114531250001, 742.5389062500035, 823.66
```

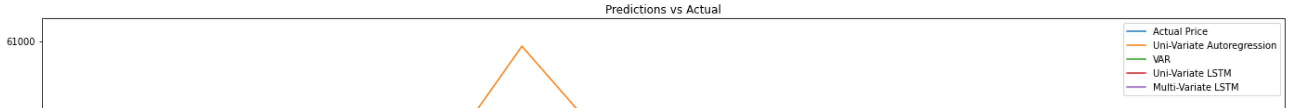
```
Mean Forecast Error: 148.573036
```

```
MLSTM_pred = valid2['Predicted'].copy()
```

Comparisons of all the models

```
plt.figure(figsize = (24,12))
plt.title('Predictions vs Actual')
plt.xlabel('Date')
plt.ylabel('USD')
plt.plot(data[training_data_len:])
plt.plot(UAR_pred)
plt.plot(VAR_pred)
plt.plot(ULSTM_pred)
plt.plot(MLSTM_pred)
plt.legend(['Actual Price','Uni-Variate Autoregression', 'VAR',
            'Uni-Variate LSTM','Multi-Variate LSTM'], loc = 'upper right')
```

```
<matplotlib.legend.Legend at 0x7f852dce7910>
```




```
print("All the RMSE: ")
print("Univariate Auto Regression: \t$", UAR_rmse)
print("Multivariate Auto Regression: \t$", VAR_rmse)
print("Uni-Variate LSTM: \t\t$", ULSTM_rmse)
print("Multi-Variate LSTM: \t\t$", MLSTM_rmse)
```

```
All the RMSE:
Univariate Auto Regression:     $ 2887.6325006673856
Multivariate Auto Regression:   $ 2425.6892969648397
Uni-Variate LSTM:               $ 417.0938169642853
Multi-Variate LSTM:             $ 148.57303571428616
```

```
print("Accuracy of all the models: ")
print()
print("\t\t\t\tUnivariate Auto Regression")
accuracy(combined_UAR)
print()
print("\t\t\t\tMultivariate Auto Regression")
accuracy(merged)
print()
print("\t\t\t\tUni-Variate LSTM")
accuracy(valid[['Bitcoin', 'Predicted']])
print()
print("\t\t\t\tMulti-Variate LSTM")
accuracy(valid2[['Bitcoin', 'Predicted']])
```

```
Accuracy of all the models:

                                Univariate Auto Regression
Actual Value: $ 55605.2  Predicted Value: $ 59177.147440611436  Accuracy:  106
Actual Value: $ 56900.75  Predicted Value: $ 55526.7975435793  Accuracy:  97.5
Actual Value: $ 58912.97  Predicted Value: $ 58087.574980603575  Accuracy:  98
Actual Value: $ 57648.16  Predicted Value: $ 60926.6208182039  Accuracy:  105.
Actual Value: $ 58030.01  Predicted Value: $ 58633.01445621121  Accuracy:  101
Actual Value: $ 54083.25  Predicted Value: $ 59715.15812162843  Accuracy:  110
Actual Value: $ 54340.89  Predicted Value: $ 54800.03115712002  Accuracy:  100
Average Accuracy over 7 Days:  102.94179314498862

                                Multivariate Auto Regression
Actual Value: $ 57886.77858725814  Predicted Value: $ 55605.2  Accuracy:  96.0
Actual Value: $ 57821.44903124853  Predicted Value: $ 56900.75  Accuracy:  98.
Actual Value: $ 58025.77388752371  Predicted Value: $ 58912.97  Accuracy:  101
Actual Value: $ 58180.31675233638  Predicted Value: $ 57648.16  Accuracy:  99.
Actual Value: $ 58246.32560848445  Predicted Value: $ 58030.01  Accuracy:  99.
Actual Value: $ 58303.686446009946  Predicted Value: $ 54083.25  Accuracy:  92
Actual Value: $ 58366.47174846299  Predicted Value: $ 54340.89  Accuracy:  93.
Average Accuracy over 7 Days:  97.22476629259121

                                Uni-Variate LSTM
Actual Value: $ 55605.2  Predicted Value: $ 57639.688  Accuracy:  103.6588079
Actual Value: $ 56900.75  Predicted Value: $ 55887.336  Accuracy:  98.21897942
Actual Value: $ 58912.97  Predicted Value: $ 57962.75  Accuracy:  98.387078431
Actual Value: $ 57648.16  Predicted Value: $ 58741.3  Accuracy:  101.89622839
```

```
Actual Value: $ 58030.01  Predicted Value: $ 57196.664  Accuracy:  98.56393969
Actual Value: $ 54083.25  Predicted Value: $ 57905.242  Accuracy:  107.0668685
Actual Value: $ 54340.89  Predicted Value: $ 53107.906  Accuracy:  97.73102032
Average Accuracy over 7 Days:  100.78898897474575

                                Multi-Variate LSTM
Actual Value: $ 55605.2  Predicted Value: $ 54584.406  Accuracy:  98.164211710
Actual Value: $ 56900.75  Predicted Value: $ 55451.484  Accuracy:  97.4529938C
Actual Value: $ 58912.97  Predicted Value: $ 56147.855  Accuracy:  95.3064418C
Actual Value: $ 57648.16  Predicted Value: $ 56905.62  Accuracy:  98.711946909
Actual Value: $ 58030.01  Predicted Value: $ 57206.344  Accuracy:  98.58062018
Actual Value: $ 54083.25  Predicted Value: $ 57380.777  Accuracy:  106.0971323
Actual Value: $ 54340.89  Predicted Value: $ 56804.73  Accuracy:  104.53404511
Average Accuracy over 7 Days:  99.83534170075141
```

```
print("Accuracy of all the models: ")
print()
print("Univariate Auto Regression:")
RFE(combined_UAR)
print()
print("Multivariate Auto Regression:")
RFE(merged)
print()
print("Uni-Variate LSTM:")
RFE(valid[['Bitcoin', 'Predicted']])
print()
print("Multi-Variate LSTM:")
RFE(valid2[['Bitcoin', 'Predicted']])
```

```
Accuracy of all the models:

Univariate Auto Regression:
[-3571.9474406114387, 1373.9524564207022, 825.3950193964265, -3278.460818203899
Mean Forecast Error: -1620.730645

Multivariate Auto Regression:
[2281.57858725814, 920.6990312485286, -887.1961124762893, 532.1567523363774, 2
Mean Forecast Error: 1615.653152

Uni-Variate LSTM:
[-2034.487500000003, 1013.4140625, 950.2200000000012, -1093.1407812499965, 833
Mean Forecast Error: -417.093817

Multi-Variate LSTM:
[1020.7937499999971, 1449.265625, 2765.114531250001, 742.5389062500035, 823.66
Mean Forecast Error: 148.573036
```