TIME SERIES MODELLING PROJECT
The Bitcoin Price Prediction

The Bitcoins are the means of exchange which are independent of any central authority. They could be transferred electronically in a secure way, which is verifiable as well as immutable.

DATASET USED:

I used the Bitcoin data downloaded from the link below:
https://coinmarketcap.com/currencies/bitcoin/historical-data/
The data has the following parameters:
Date: Index of the dataset
Open: The opening value of the Bitcoin for the corresponding Date
Close: The closing value of the Bitcoin for the corresponding Date
High: The Highest value of the Bitcoin for the corresponding Date
Low: The lowest value of the Bitcoin for the corresponding Date
Volume: The number of Bitcoin circulating in the market
Market Capital: The amount of Capital invested in the market for the Bitcoins.
This data had 730 rows of data from 2019-08-31 to 2017-09-01 with the above features.

The variance and mean of the data is as follows:

Variance:

Open	9.719702e+06
High	1.075295e+07
Low	8.450390e+06
Close	9.715011e+06
Volume	4.957465e+19
Market Cap	2.803524e+21

Mean:

Open	7.428405e+03
High	7.647549e+03
Low	7.186859e+03
Close	7.435033e+03
Volume	9.038557e+09
Market Cap	1.279184e+1

Following is the description of the data:

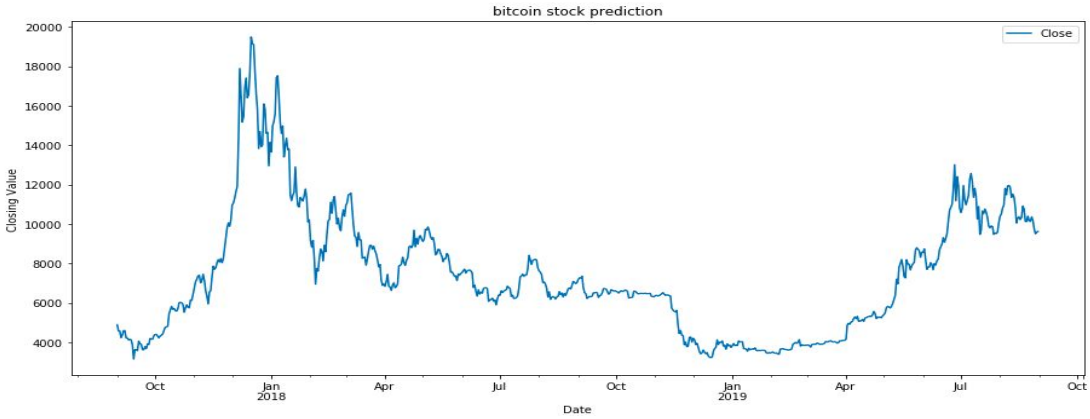
	Open	High	Low	Close	Volume	Market Cap
count	730.000000	730.000000	730.000000	730.000000	7.300000e+02	7.300000e+02
mean	7428.404795	7647.549342	7186.858726	7435.033384	9.038557e+09	1.279184e+11
std	3117.643599	3279.169306	2906.955497	3116.891294	7.040927e+09	5.294831e+10
min	3166.300000	3275.380000	2946.620000	3154.950000	7.680150e+08	5.226545e+10
25%	5072.147500	5236.130000	5018.992500	5091.302500	4.274632e+09	8.982587e+10
50%	6750.745000	6872.780000	6618.775000	6765.250000	6.111765e+09	1.159323e+11
75%	9132.860000	9390.897500	8824.992500	9173.817500	1.261413e+10	1.560330e+11
max	19475.800000	20089.000000	18974.100000	19497.400000	4.510573e+10	3.265025e+11

I plotted all the features in the dataset to observe some trends :

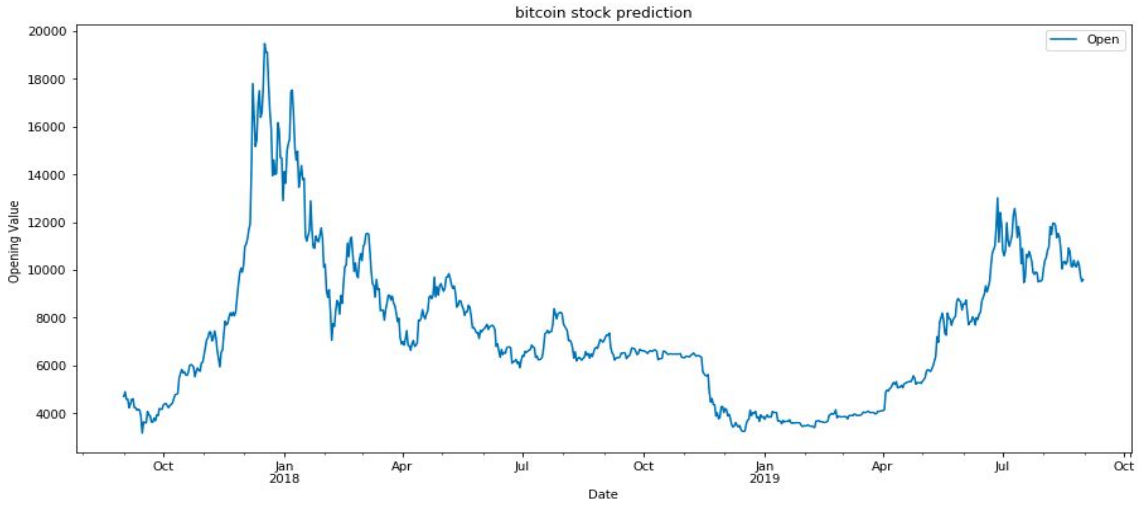
On plotting the values I found that the values like the Opening value, closing value, the market Capital, the highest and the lowest values, all of them follow the same trend. Also, the most important feature amongst all the values is the Closing value of the Bitcoin for a particular day. The closing value for the Bitcoin is the final value for the day and is of the most importance for the investors. Thus, on the basis of these plots, we learn that all the other features do not add any effect to the plot of the closing value and also follow the same trend, therefore, it is not essential to take them into consideration, and using the time series model just on the Closing Value will give us the desired results.

I did the forecast using two approaches: **1. Using the Time Series Model**
2. Using the Machine Learning Model

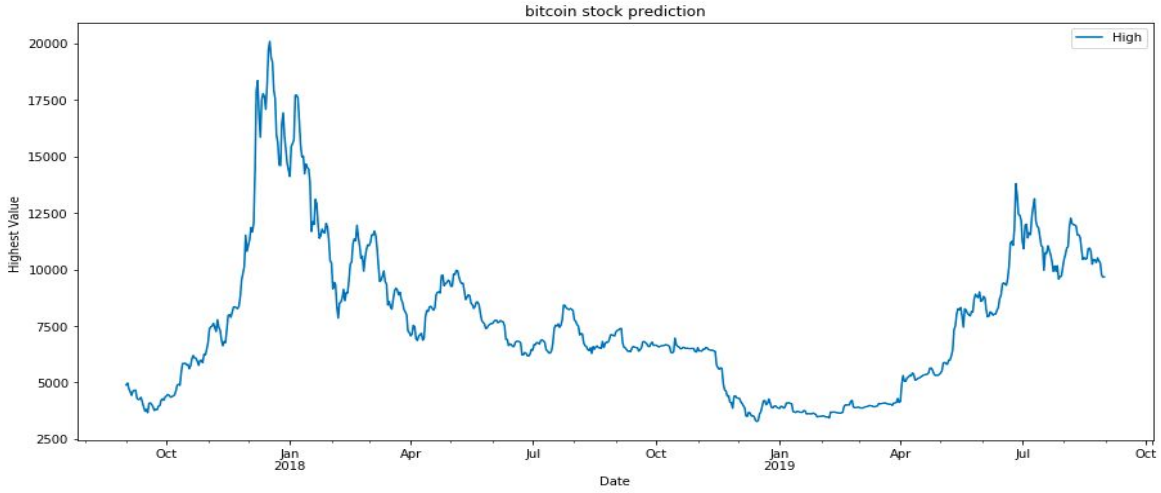
Plot for Opening Value each day:



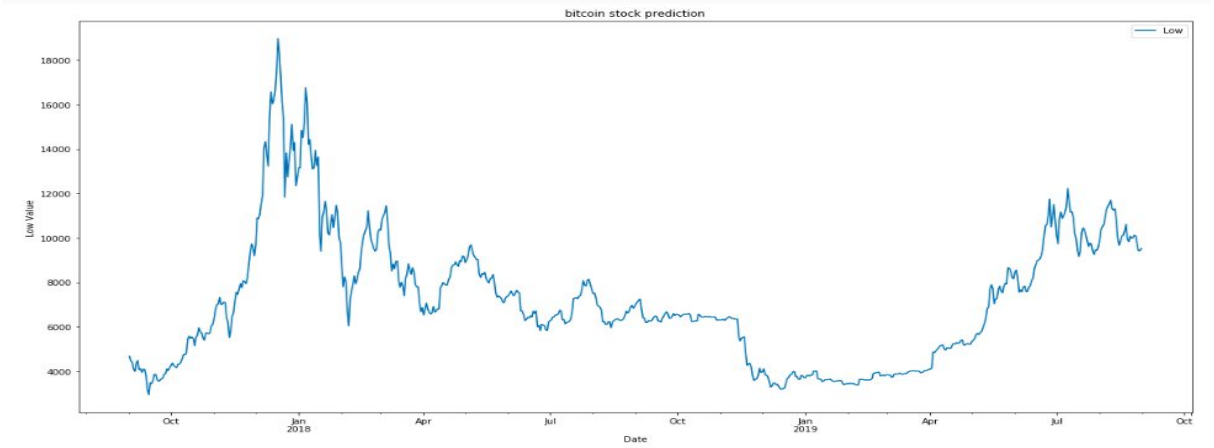
Plot for losing Value each day:



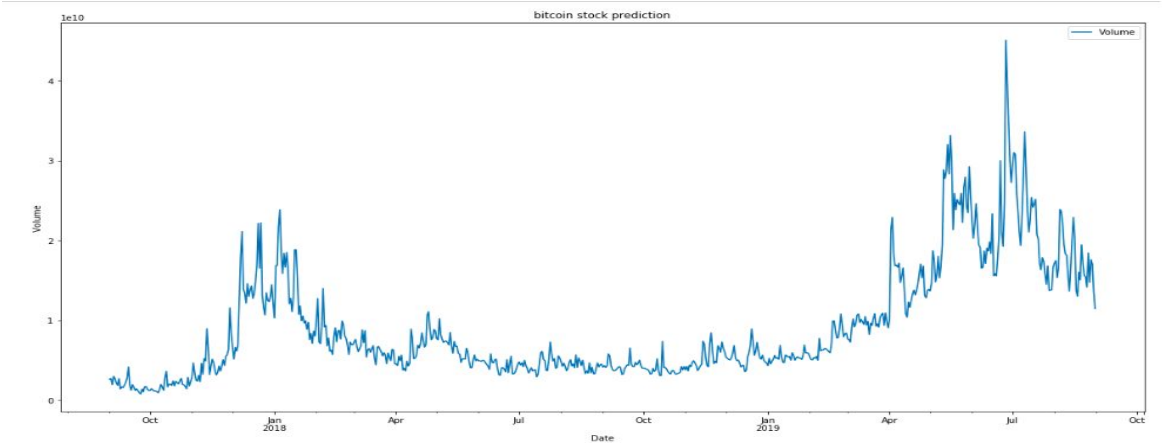
Plot for the highest value each day:



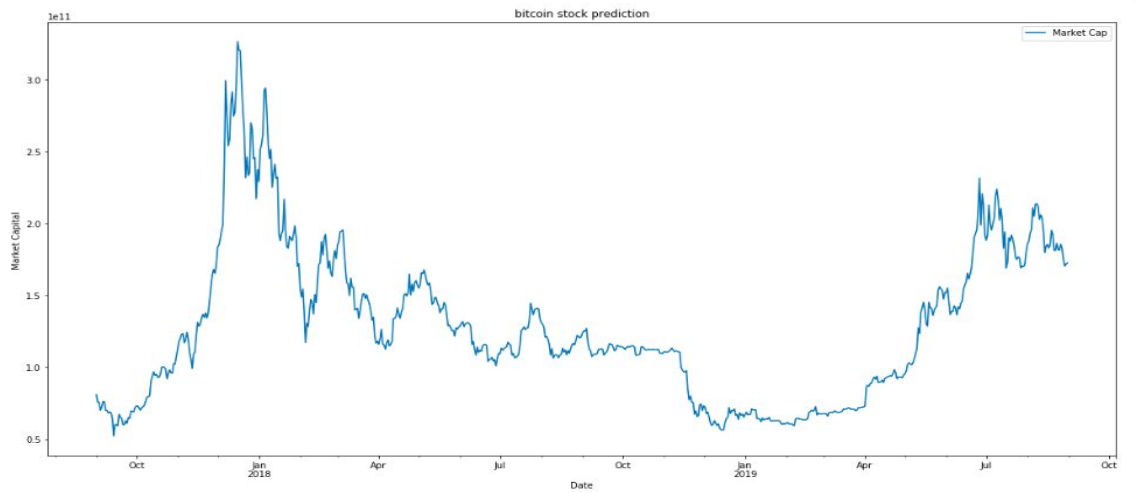
Plot for the lowest value each day:



Plot for the volume in the market day wise:



Plot for the Capital in the market:



Following is the value of correlations between the features given in the data:

We observed in the above graphs that most of the features follow the same trend except the Volume feature, which means that they are highly correlated, on finding out the correlation among them using df.corr(), I got the following

correlation values which proves that indeed the features are highly correlated, since the values of correlation is nearly equal to 1.

	Open	High	Low	Close	Volume	Market Cap
Open	1	0.995681	0.992635	0.990071	0.485406	0.988051
High	0.995681	1	0.992519	0.996415	0.494478	0.99354
Low	0.992635	0.992519	1	0.995167	0.47843	0.994538
Close	0.990071	0.996415	0.995167	1	0.488662	0.997965
Volume	0.485406	0.494478	0.47843	0.488662	1	0.527862
Market Cap	0.988051	0.99354	0.994538	0.997965	0.527862	1

After concluding that the data is highly correlated we drop all the other values, and just keep the closing value in our data frame so the multivariate data is now reduced to a univariate data with the date as index and closing price as the time series data, on which we apply the time series model.

Checking the stationarity and seasonality:

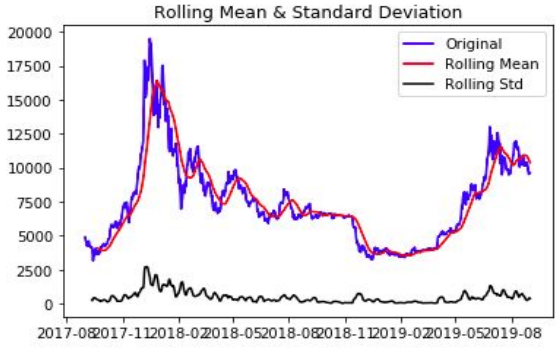
Although we can make out from the above plots that the data is not stationary yet to confirm we performed the **Augmented Dickey Fuller test.**

The Augmented Dickey Fuller Test is a statistical test also called the unit root test.The intuition behind a unit root test is that it determines how strongly a time series is defined by a trend.

1. Null Hypothesis (H0): Null hypothesis of the test is that the time series can be represented by a unit root that is not stationary.
2. Alternative Hypothesis (H1): Alternative Hypothesis of the test is that the time series is stationary.

Interpretation of p value

1. p value > 0.05: Accepts the Null Hypothesis (H0), the data has a unit root and is non-stationary.
2. p value < = 0.05: Rejects the Null Hypothesis (H0), the data is stationary.

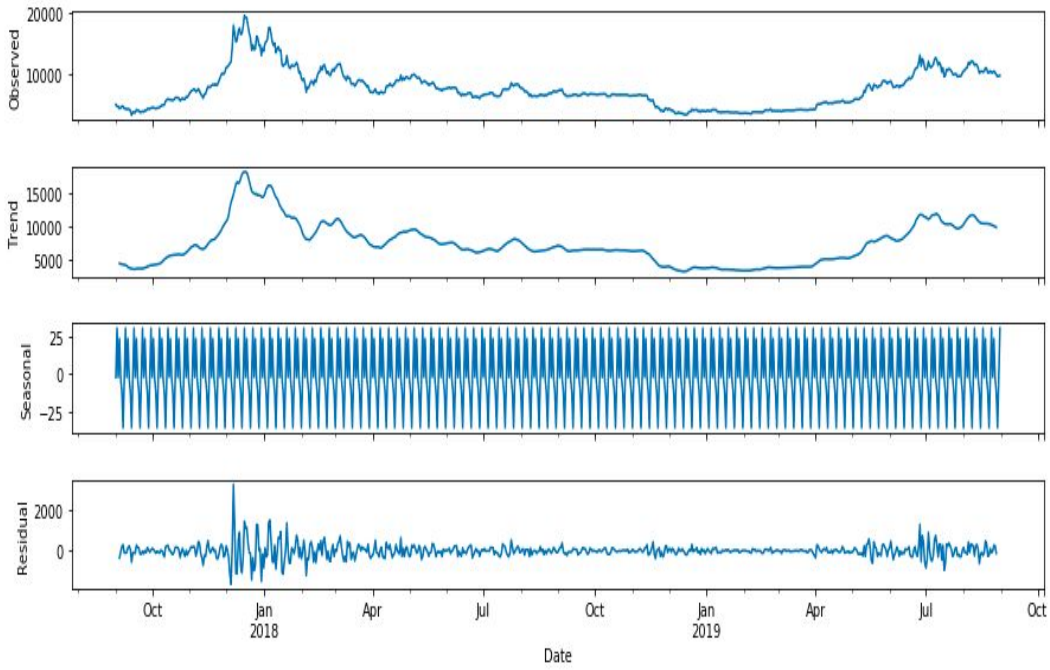



```
ADF Stastistic: -2.512509
p-value: 0.112460
The graph is non stationary
Critical values:
        1%: -3.440
        5%: -2.866
        10%: -2.569
```

In the results we can see that, the p value of the data comes out to be **0.112460,** which is greater than 0.05. Therefore, we accept the Null Hypothesis (H0), the data has a unit root and is non-stationary. our data is not stationary.

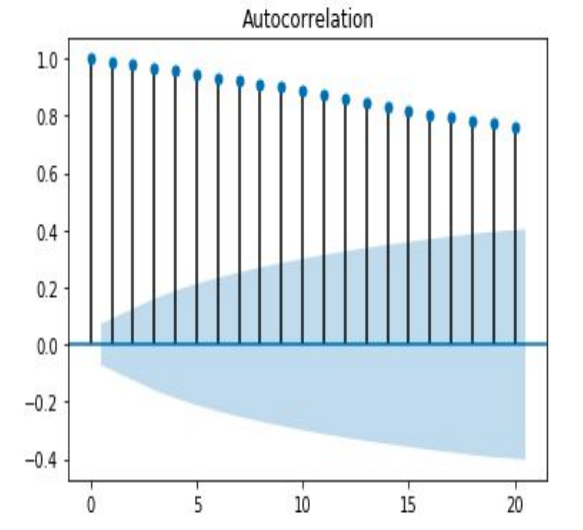
Decomposition Plot:

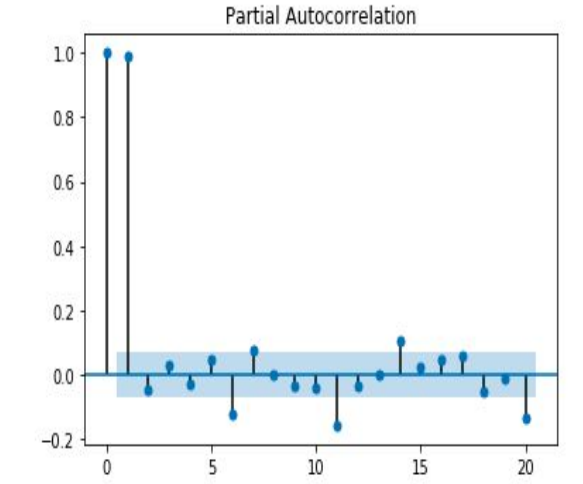
This plots the seasonality, trend and residuals.



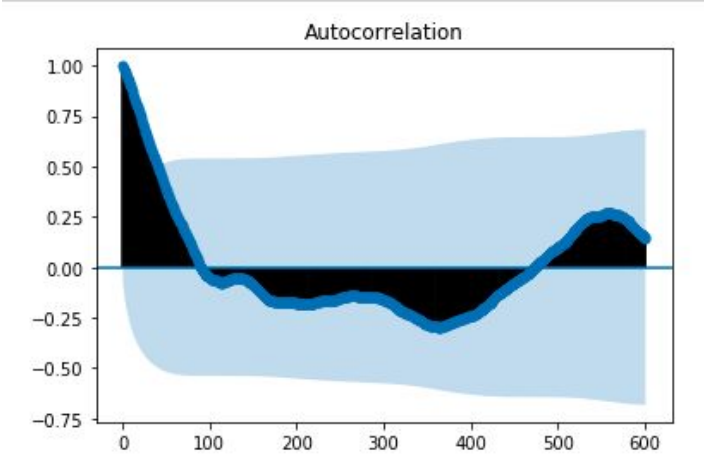
The above plot clearly shows that our data has **seasonality** and is following **certain trends.** The data although is **not periodic** since we do not observe any cycle in the data. Thus, we conclude that our data is not stationary and also it is seasonal in nature.

Following are the ACF and PACF plot for our data:





the ACF and PACF plots we can see that they are not attenuating and not cutting off as well, thus we cannot clearly determine which model we can apply, we further perform differencing to determine the new ACF 0and PACF plots.



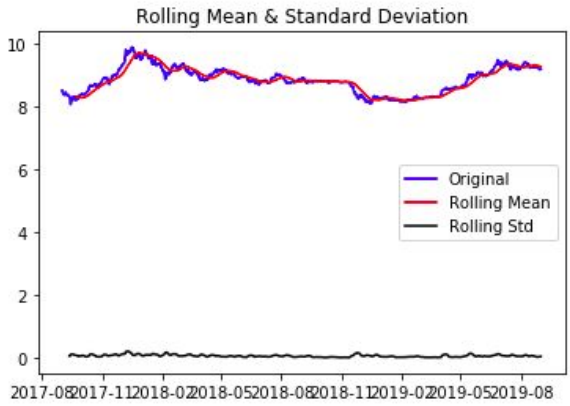
The above ACF plot over 600 lags depicts some kind of seasonality in the data, which we will further study.

Performing Differencing to make the data stationary:

This can be done in two ways, the log differencing and the normal differencing:

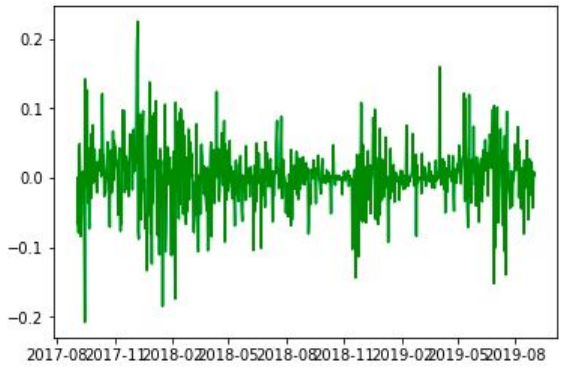
The Log Differencing:

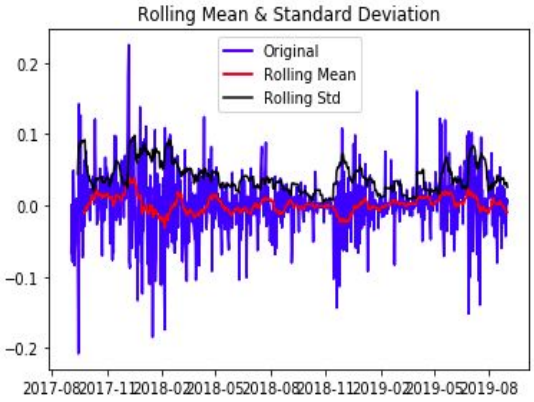
Following is the graph obtained after performing log transformation on the data, and it's corresponding rollin mean and standard deviation and the result of ADF Test for stationarity:




```
ADF Stastistic: -1.487079
p-value: 0.539893
The graph is non stationary
Critical values:
        1%: -3.439
        5%: -2.866
        10%: -2.569
```

After the above transformation we can make out from the graph of rolling mean and standard deviation and also from the p value which is more than 0.05 that the data is not stationary. We then perform the log differencing and obtain the following values:



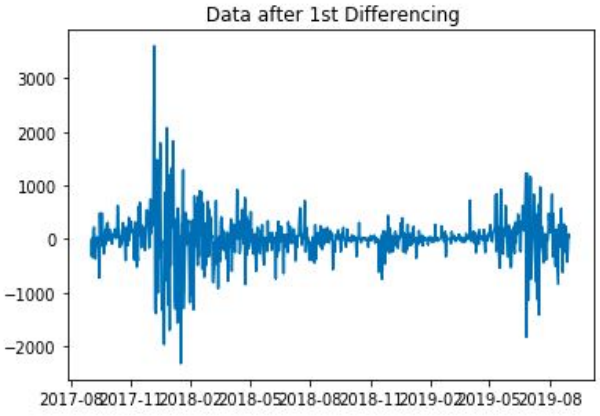


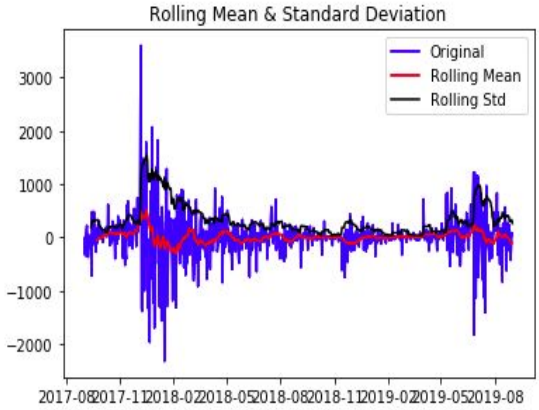

```
ADF Stastistic: -27.279931
p-value: 0.000000
The graph is stationary
Critical values:
        1%: -3.439
        5%: -2.866
        10%: -2.569
```

From the above plots of rolling mean and standard deviation and the p-value, which is equal to zero, we can make out that the data is now Stationary.

Normal Differencing:

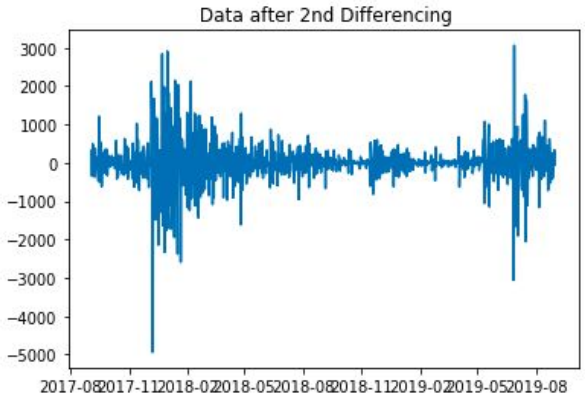
Following is the plot of the data after the first differencing:

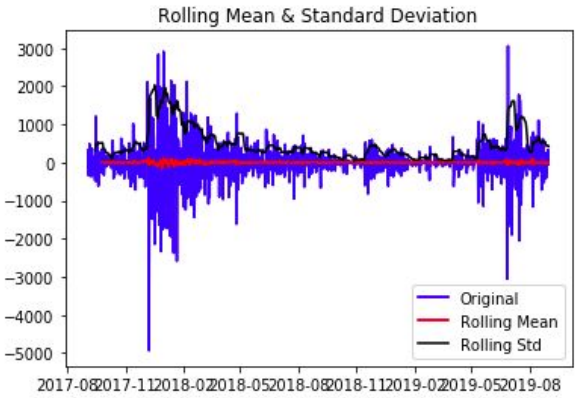





```
ADF Stastistic: -5.112709
p-value: 0.000013
The graph is stationary
Critical values:
        1%: -3.440
        5%: -2.866
        10%: -2.569
```

The data after first differencing has become almost stationary but to further improve the results we perform the second differencing. Following are the results after second differencing.

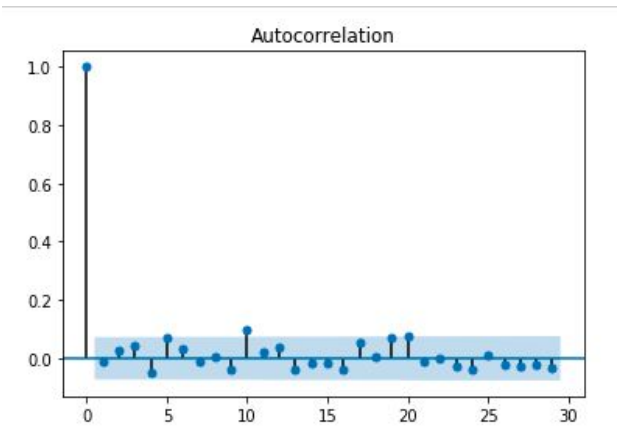


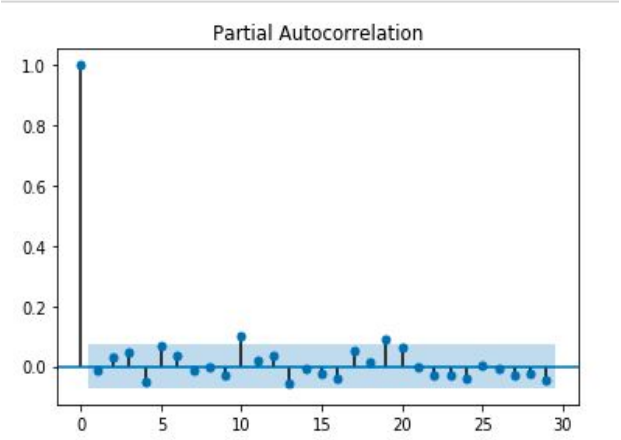



```
ADF Stastistic: -11.733236
p-value: 0.000000
The graph is stationary
Critical values:
        1%: -3.440
        5%: -2.866
        10%: -2.569
```

After performing the 2nd differencing we observe that the data is now nicely stationary and does not require any further differencing, since the p value is also nearly equal to zero for this data. Now the data is completely stationary. We will consider the log differencing since we obtained better results after the log differencing.

Following are the ACF and PACf plots after the log differencing:





We find that again the ACF and PACF plots do not convey any essential information on what values of p,q,d should be used. Thus, we decide the value of p,q,d (non-seasonal parameters) and the value of P,Q,D and m (seasonal parameters) on the basis of the AIC and BIC value. The model with the least value of AIC and BIC is used.

In Python, we can directly use the auto_arima, which calculates all the possible AIC and BIC values and returns the model with the least value of AIC and BIC.

The results of auto_arima are as follows:

I used auto_arima function imported from pmdarima which on it self tells what values of parameters (p,q,d)(P,Q,D)m will give the best result, i.e. it helps us choose the best time series model. Where,

(p,q,d) → non-seasonal part

(P,Q,D)m → seasonal part

Thus, the model with the least value of AIC and BIC is chosen and is the output of the auto_arima function.

```
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 1, 1, 12); AIC=-2335.392, BIC=-2312.523, Fit time=17.872 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 1, 0, 12); AIC=-1434.853, BIC=-1425.706, Fit time=0.178 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(1, 1, 0, 12); AIC=-1823.940, BIC=-1805.645, Fit time=5.861 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 1, 1, 12); AIC=-2336.255, BIC=-2317.961, Fit time=13.690 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(1, 1, 1, 12); AIC=-2335.096, BIC=-2312.228, Fit time=12.750 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 1, 0, 12); AIC=-1922.508, BIC=-1908.787, Fit time=1.443 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 1, 2, 12); AIC=-2275.354, BIC=-2252.486, Fit time=32.864 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(1, 1, 2, 12); AIC=-2331.736, BIC=-2304.294, Fit time=28.855 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 1, 1, 12); AIC=-1849.292, BIC=-1835.571, Fit time=5.478 seconds
Fit ARIMA: order=(0, 1, 2) seasonal_order=(0, 1, 1, 12); AIC=-2335.608, BIC=-2312.740, Fit time=15.045 seconds
Fit ARIMA: order=(1, 1, 2) seasonal_order=(0, 1, 1, 12); AIC=-2302.034, BIC=-2274.592, Fit time=10.247 seconds
Total fit time: 144.291 seconds
```

```
: ARIMA(callback=None, disp=0, maxiter=None, method=None, order=(0, 1, 1),
       out_of_sample_size=0, scoring='mse', scoring_args=None,
       seasonal_order=(0, 1, 1, 12), solver='lbfgs', start_params=None,
       suppress_warnings=True, transparams=True, trend=None,
       with_intercept=True)
```

We found that the model with the least value of AIC and BIC is

Non Seasonal Parameters: **p=0, q=1 and d=**1

Seasonal Parameters: P=0, Q=1, D=1 and m=12

With AIC and BIC values as AIC=-2336.255, BIC=-2317.961

. Where,

p: Trend autoregression order.

d: Trend difference order.

q: Trend moving average order.

P: Seasonal autoregressive order.

D: Seasonal difference order.With AIC and BIC values as AIC=10815.210, BIC=10828.935

Q: Seasonal moving average order.

m: The number of time steps for a single seasonal period.

After splitting the data into the test and training data set with the first 638 values in the training set and the rest 92 values in the test data set. We train the sarima model **'SARIMA(0,1,1)(0,1,1,12)'** with the following details:

The equation for the **SARIMA(0,1,1)(0,1,1,12)** model is $(1-B)(1-B^{12})y_t = (1+B)(1+B^{12})e_t$

Where y_t is the output variable and e_t is the white noise.

```
In [75]: sarima = sm.tsa.statespace.SARIMAX(ts_log_diff,order=(0,1,1),seasonal_order=(0,1,1,12),
                                           enforce_stationarity=False, enforce_invertibility=False).fit()
         sarima.summary()
```

```
/home/mohita/anaconda3/lib/python3.7/site-packages/statsmodels/tsa/base/tsa_model.py:165: ValueWarning: No freque
ncy information was provided, so inferred frequency D will be used.
  % freq, ValueWarning)
```

Residuals

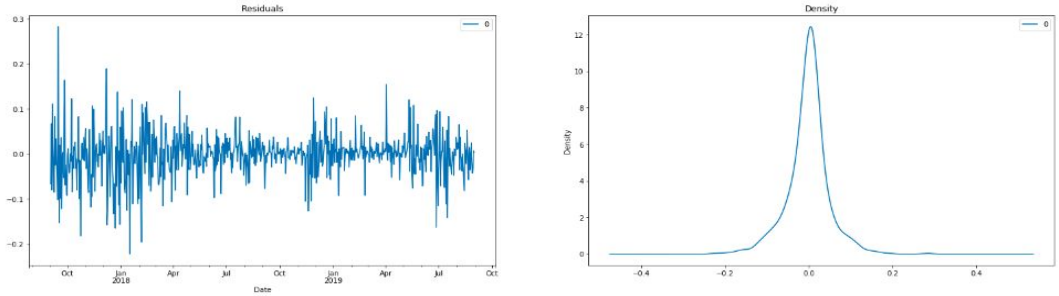
The residuals in a time series model are what is left over after fitting a model.

The residuals have the following properties:

- The residuals are uncorrelated. If there are correlations between residuals, then there is information left in the residuals which should be used in computing forecasts.
- They should have a mean 0.
- They should have constant variance.
- The residuals should be normally distributed.

We find that the residuals which we obtained from the SARIMA(**0,1,1)(0,1,1,12), do not have a constant variance**

```
In [77]: # Plot residual errors
         residuals = pd.DataFrame(sarima.resid)
         fig, ax = plt.subplots(1,2)
         residuals.plot(title="Residuals", ax=ax[0])
         ## kde=kernel density estimation
         residuals.plot(kind='kde', title='Density', ax=ax[1],figsize=(25,7))
         plt.show()
```



They have a **mean of -0.001448** which is nearly 0.

```
In [78]: residuals.mean()
Out[78]: 0   -0.001448
         dtype: float64
```

The Box-Ljung p-value Test for Independence (THE WHITENESS TEST) for SARIMA(0,1,1)(0,1,1,12)

I performed the Box-Ljung test on the residuals. This test is to check whether the discrete white noise being a good fit to the residuals or not.The piece of code below returns two values, the first one is The Ljung-Box test statistic and the second one is the p value. The p-value should be larger than 0.05 for the r discrete white noise being a good fit to the residuals.

The p value comes out to be **0.03129502** and since, the p-value is smaller than 0.05 which is strong evidence for discrete white noise being not a good fit to the residuals. Hence, the SARIMA(0,1,1)(0,1,1,12) model is not a good fit.

```
In [142]: sm.stats.acorr_ljungbox(residuals, lags=[10])
Out[142]: (array([19.79076132]), array([0.03129502]))
```

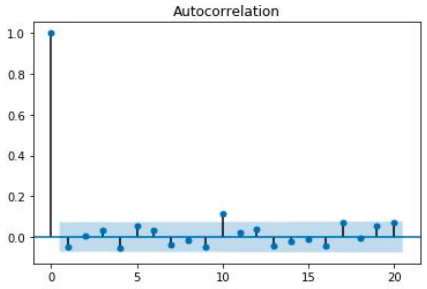
The ACF and PACF plots on residuals:

The residuals are correlated.

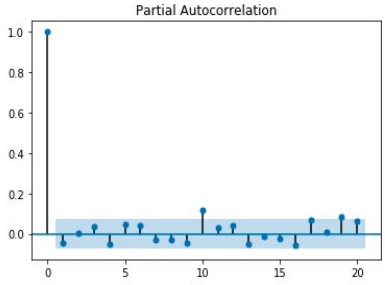
Also, from the ACF graph we deduce that they are correlated as the values do not fall within the confidence boundaries. From the residual plot we deduce that they do not have a constant variance.

Therefore, we need transformation to achieve these properties and the model which have chosen seems incorrect.

```
In [124]: plot_acf(residuals.dropna(), lags=20)
          plt.show()
```




```
In [82]: plot_pacf(residuals.dropna(), lags=20)
         plt.show()
```



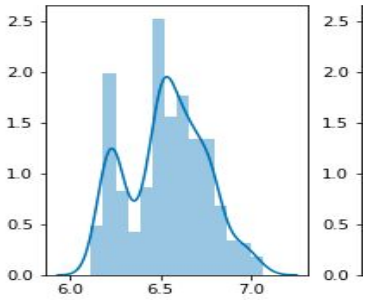
Box Cox Transformation

We performed the Whiteness Test on the residuals and found them not to be white, also we found them to be with no constant variance also, the ACF and PACF plot tell that they are correlated thus we do not need to perform BOX COX Transformation on our data.

```
In [97]: import scipy
         from scipy import stats
         ts_box=scipy.stats.boxcox(ts, lmbda=None, alpha=None)
         ts_bc=pd.DataFrame(ts_box[0])
```

We can see the transformation in our data in the plot below:
The data in the plot below now looks normalized.

```
In [151]: import seaborn as sns
          #sns.distplot(ts_bc, ax=ax[0
          fig, ax=plt.subplots(1,2)
          sns.distplot(ts_bc, ax=ax[0]
          sns.distplot(ts_bc, ax=ax[1]
Out[151]: <matplotlib.axes._subplots.A
```



We apply the various models again on the Box-Cox transformed data and analyse the results:
The auto_arima function gave us the following model on the basis of least AIC and BIC values.

Thus, we select the model **'SARIMA(0,1,0)(0,1,1,12)' with the AIC=-3282.306, BIC=-3268.581**.
This on the basis of the selection of the model with the least value of AIC and BIC.
Below we can see what Auto_arima function returns and on the basis of that we picked the model and fitted it to our data. The equation of the **SARIMA(0,1,0)(0,1,1,12) is** $(1-B)(1-B^{12})y_t = (1+B^{12})e_t$ where, y_t is the output variable and e_t is the white noise

```
In [99]: from pmdarima import auto_arima
         auto_arima(ts_bc, start_p=1, start_q=1,
                                  max_p=3, max_q=3, m=12,
                                  start_P=0, seasonal=True,
                                  d=1, D=1, trace=True,
                                  error_action='ignore',
                                  suppress_warnings=True,
                                  stepwise=True)

         Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 1, 1, 12); AIC=-3278.600, BIC=-3255.725, Fit time=14.689 seconds
         Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 1, 0, 12); AIC=-2863.251, BIC=-2854.101, Fit time=0.315 seconds
         Fit ARIMA: order=(1, 1, 0) seasonal_order=(1, 1, 0, 12); AIC=-3034.786, BIC=-3016.486, Fit time=6.425 seconds
         Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 1, 1, 12); AIC=-3278.720, BIC=-3260.420, Fit time=14.071 seconds
         Fit ARIMA: order=(0, 1, 1) seasonal_order=(1, 1, 1, 12); AIC=-3278.910, BIC=-3256.035, Fit time=14.622 seconds
         Fit ARIMA: order=(0, 1, 1) seasonal_order=(1, 1, 0, 12); AIC=-3034.783, BIC=-3016.483, Fit time=4.154 seconds
         Fit ARIMA: order=(0, 1, 1) seasonal_order=(1, 1, 2, 12); AIC=-3254.854, BIC=-3227.404, Fit time=10.005 seconds
         Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 1, 0, 12); AIC=-2861.331, BIC=-2847.606, Fit time=1.374 seconds
         Fit ARIMA: order=(0, 1, 1) seasonal_order=(2, 1, 2, 12); AIC=-3275.149, BIC=-3243.123, Fit time=40.899 seconds
         Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 1, 1, 12); AIC=-3276.935, BIC=-3249.484, Fit time=17.033 seconds
         Fit ARIMA: order=(0, 1, 0) seasonal_order=(1, 1, 1, 12); AIC=-3281.354, BIC=-3263.053, Fit time=9.239 seconds
         Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 1, 1, 12); AIC=-3282.306, BIC=-3268.581, Fit time=6.391 seconds
         Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 1, 2, 12); AIC=-3264.997, BIC=-3246.696, Fit time=8.790 seconds
         Fit ARIMA: order=(0, 1, 0) seasonal_order=(1, 1, 2, 12); AIC=-3260.178, BIC=-3237.302, Fit time=10.495 seconds
         Fit ARIMA: order=(1, 1, 0) seasonal_order=(0, 1, 1, 12); AIC=-3280.495, BIC=-3262.195, Fit time=11.873 seconds
         Total fit time: 170.382 seconds

Out[99]: ARIMA(callback=None, disp=0, maxiter=None, method=None, order=(0, 1, 0),
               out_of_sample_size=0, scoring='mse', scoring_args=None,
               seasonal_order=(0, 1, 1, 12), solver='lbfgs', start_params=None,
               suppress_warnings=True, transparams=True, trend=None,
               with_intercept=True)
```

Fitting the model on the transformed data with the following summary:

```
]: sarima = sm.tsa.statespace.SARIMAX(ts_bc,order=(0,1,0),seasonal_order=(0,1,1,12),
                                      enforce_stationarity=False, enforce_invertibility=False).fit()
   sarima.summary()
```

]:

Statespace Model Results

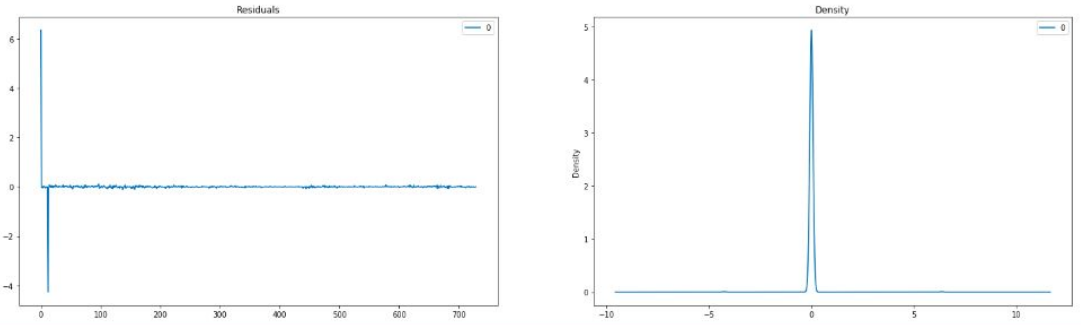
Dep. Variable:	0	**No. Observations:**	730
Model:	SARIMAX(0, 1, 0)x(0, 1, 1, 12)	**Log Likelihood**	1626.675
Date:	Mon, 02 Dec 2019	**AIC**	-3249.349
Time:	08:09:21	**BIC**	-3240.236
Sample:	0	**HQIC**	-3245.827
	- 730		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ma.S.L12	-1.0001	7.288	-0.137	0.891	-15.284	13.284
sigma2	0.0005	0.004	0.137	0.891	-0.007	0.008

Ljung-Box (Q):	43.91	**Jarque-Bera (JB):**	220.64
Prob(Q):	0.31	**Prob(JB):**	0.00
Heteroskedasticity (H):	0.44	**Skew:**	-0.20
Prob(H) (two-sided):	0.00	**Kurtosis:**	5.71

We now analyse the Residuals after the Box Cox transformation:
The Residuals now seem to be **normally distributed** and seem to have a **constant variance** too, as desired.

```
In [132]: # Plot residual errors
          residuals = pd.DataFrame(sarima.resid)
          fig, ax = plt.subplots(1,2)
          residuals.plot(title="Residuals", ax=ax[0])
          ## kde=kernel density estimation
          residuals.plot(kind='kde', title='Density', ax=ax[1],figsize=(25,7))
          plt.show()
```



The **mean** of the residuals come out to be 0.00274, which is nearly equal to zero as desired.

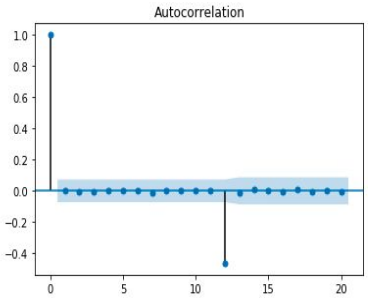
```
In [133]: residuals.mean()
Out[133]: 0    0.00274
          dtype: float64
```

After the Box-Cox transformation the residuals pass the Whiteness Test too.
The Box-Ljung p-value comes out to be 0.99999 which is sufficiently greater than 0.05, which is strong evidence for discrete white noise being a good fit to the residuals.

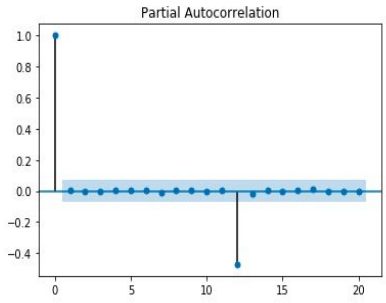
```
In [134]: sm.stats.acorr_ljungbox(residuals, lags=[10])
Out[134]: (array([0.14975099]), array([0.99999998]))
```

Analysing the ACF and PACF plots for the Box-Cox transformed data:

```
In [135]: plot_acf(residuals.dropna(), lags=20)
          plt.show()
```




```
In [136]: plot_pacf(residuals.dropna(), lags=20)
          plt.show()
```



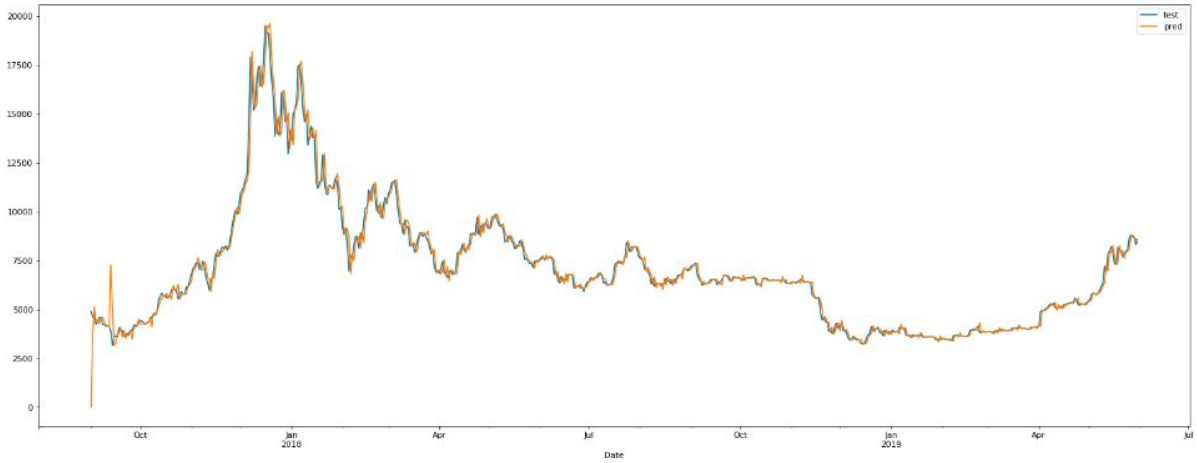
From the ACF graph we deduce that the residuals are now not correlated as the values now fall within the confidence boundaries.
Therefore, now the model which have chosen seems correct.

Now, finally we can make predictions on the test data and also perform forecasting.
After fitting the model we predict the values for the test data and obtain the following results:

```
In [62]: from sklearn.metrics import mean_squared_error
         pred = sarima.predict(tr_start,te_end)
```

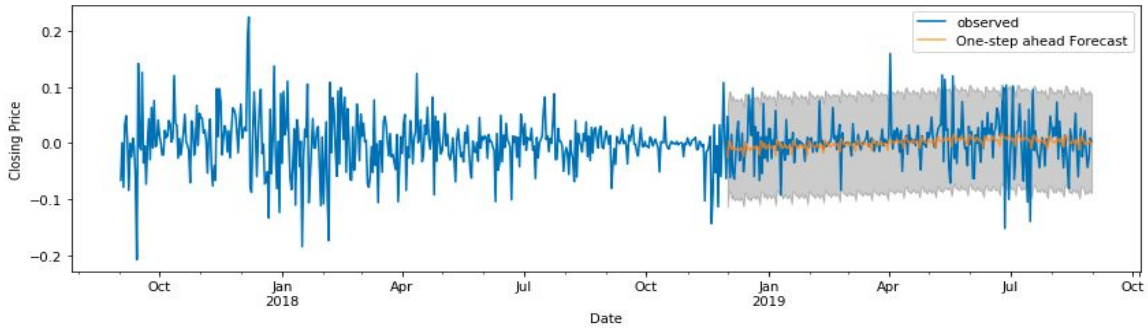
```
In [66]: tss=ts[:638]
         pd.DataFrame({'test':tss,'pred':pre}).plot(figsize=(25,10))
         plt.show()
```

We obtain the following result:



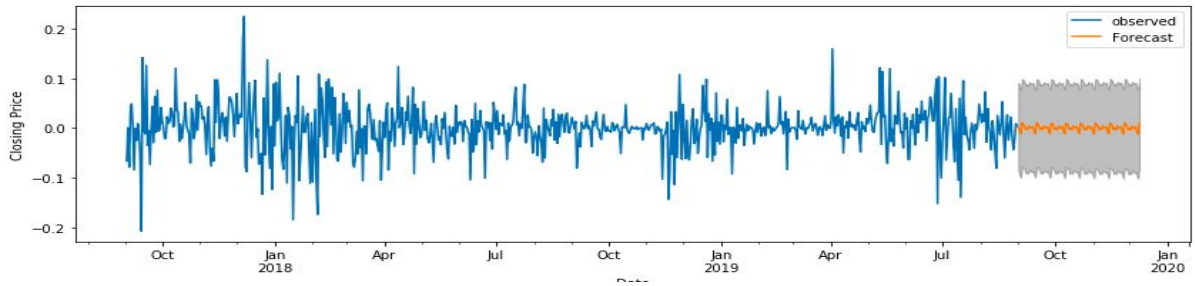
Here, we can observe that the blue line which is the test data and orange line is the predict data fit well. Thus, the predicted value is quite similar to the actual test value and is following all the trend and seasonality.
I also made some future forecast which is shown in the plot below.

```
pred = sarima1.get_prediction(start=pd.to_datetime('2018-12-01'), dynamic=False)
pred_ci = pred.conf_int()
ax = ts_log_diff['2017':].plot(label='observed')
pred.predicted_mean.plot(ax=ax, label='One-step ahead Forecast', alpha=.7, figsize=(14, 4))
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.2)
ax.set_xlabel('Date')
ax.set_ylabel('Closing Price')
plt.legend()
plt.show()
```



The above forecast is for one step ahead. The forecast below is for 12 steps ahead

```
pred_uc = sarima.get_forecast(steps=100)
pred_ci = pred_uc.conf_int()
ax = ts_log_diff.plot(label='observed', figsize=(14, 4))
pred_uc.predicted_mean.plot(ax=ax, label='Forecast')
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.25)
ax.set_xlabel('Date')
ax.set_ylabel('Closing Price')
plt.legend()
plt.show()
```



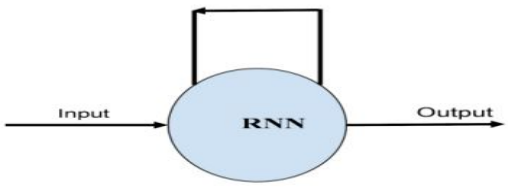
Therefore, we observe that the outcome of our model is keeping up with trends.
Also, the RMSE and MSE of the model is almost around 0 when rounded to two decimal positions which is pretty good.

```
y_forecasted = pred.predicted_mean
y_truth = ts_log_diff['2019-06-01':]
mse = ((y_forecasted - y_truth) ** 2).mean()
print('The Mean Squared Error is {}'.format(round(mse, 2)))
print('The Root Mean Squared Error is {}'.format(round(np.sqrt(mse), 2)))
```

```
The Mean Squared Error is 0.0
The Root Mean Squared Error is 0.05
```

Forecasting Using Machine Learning :
For performing this forecast I used the Keras with TensorFlow as it's backend. I did this for a multivariate time series taking into consideration all the time series data in the input.
I divided the data into test and train with first 630 values in the train data and the rest 99 values as the test data.

Forecasting using RNN(LSTMs)
To perform this I used the **RNN(Recurrent Neural Networks),** specifically the **LSTMs(Long Short term Memory).** The recurrent Neural Networks a special type of neural network designed for efficiently processing sequential data. This makes them a suitable candidate for a task such as time-series analysis. The "recurrent" part depicts the fact that these networks apply the same set of operations on all the nodes to model the temporal relationship of data samples. A recurrent neuron stores the state of a previous input and combine with the current input thereby preserving some relationship of the current input with the previous input.
Following is a general structure of a Recurrent Neural Network.



From the above diagram we deduce that the neuron keeps the account of previous input and also the present input.
Following are the steps for training the neural network:

Input xt is supplied to the network. The current state(ht) is calculated at each neuron using a combination of current and previous states

$h_t = f(h_t - 1,\ x_t)$

For the next time step current h_t becomes h_t -1.

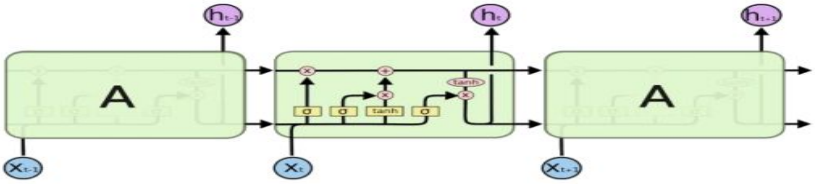
The number of steps depends on the problem to be solved we can combine the information from all previous states.

$h_t = tanh(w_h * h_t - 1 + w_x * x)$

Once the process is done for all the time steps the final current state defines the output y_t .

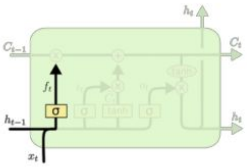
LSTM is an extension of the classic recurrent networks. The LSTMs also address the problem of vanishing gradients, which means that sometimes gradients tend to become 0 as the error keeps on propagating through the various layers recursively. The long-short term memory cell has the following gates an input gate, a forget gate and an output gate. These gates helps our neural network to learn what to save and what not to. A gate just like a MLP(multi layer perceptron. Instead of neurons, LSTM networks have memory blocks that are connected through layers.

The memory operates upon an input sequence and each gate within a block uses the sigmoid activation units to control whether they are triggered or not, making the change of state and addition of information flowing through the block conditional. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means ―let nothing through, ‖ while a value of one means ―let everything through! Below are the four interacting Layers of LSTMs.



An LSTM has three of these gates, to protect and control the cell state.

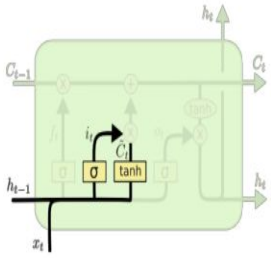
Forget Gate decides what information to discard from the block.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at ht−1 and xt, and outputs a number between 0 and 1 for each number in the cell state Ct−1. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."

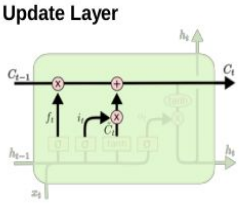
Input Gate decides whether or not to update the memory state on the basis of input conditionally.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

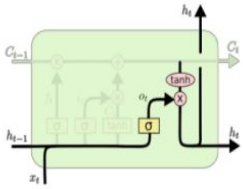
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Update Layer



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

It's now time to update the old cell state, Ct−1, into the new cell state Ct. The previous steps already decided what to do, we just need to actually do it.
We multiply the old state by ft, forgetting the things we decided to forget earlier. Then we add
it∗C~t. This is the new candidate values, scaled by how much we decided to update each state value.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between −1−1 and 11) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

Multivariate time series data means data where there is more than one observation for each time step. There are two main models that we may require with multivariate time series data; they are:

1. Multiple Input Series.
2. Multiple Parallel Series.

Our data falls under the Multiple Input Time Series, here the input is two or more time series but the output is to predict a single time series.

The first step was to scale the **MULTIVARIATE** data using the MinMaxScaler and save the newly scaled data into as a new reframed data, on which have performed the training and testing

```
In [8]: values = df.values


        ## full data without resampling
        #values = df.values

        # integer encode direction
        # ensure all data is float
        #values = values.astype('float32')
        # normalize features
        scaler = MinMaxScaler(feature_range=(0, 1))
        scaled = scaler.fit_transform(values)
        # frame as supervised learning
        reframed = series_to_supervised(scaled, 1, 1)

        # drop columns we don't want to predict
        reframed.drop(reframed.columns[[6,7,8,10,11]], axis=1, inplace=True)
        print(reframed.head())
```

	var1(t-1)	var2(t-1)	var3(t-1)	var4(t-1)	var5(t-1)	var6(t-1)	var4(t)
1	0.094145	0.096150	0.108059	0.106291	0.041298	0.104434	0.087124
2	0.106387	0.101088	0.095001	0.087124	0.044074	0.085578	0.087380
3	0.087003	0.085568	0.091778	0.087380	0.026280	0.085863	0.066169
4	0.087393	0.078285	0.072487	0.066169	0.050055	0.064986	0.074749
5	0.065115	0.068543	0.065605	0.074749	0.043529	0.073485	0.088247

I began with splitting the data into test and train data. The train data for training the model and the test data for predicting and checking the errors. The first 630 rows in y multivariate data is the training data and the remaining 100 are the test values, on which I have performed the prediction and calculated the RMSE for both the ANN and LSTM deep neural network models. Also, the scaling makes calculations easier now we have the values which are normalised and lie between 0 and 1.

```
values = reframed.values


train = values[:630, :]
test = values[630:,:]
##test = values[n_train_time:n_test_time, :]
# split into input and outputs
train_X, train_y = train[:, :-1], train[:, -1]
test_X, test_y = test[:, :-1], test[:, -1]
# reshape input to be 3D [samples, timesteps, features]
train_X = train_X.reshape((train_X.shape[0], 1, train_X.shape[1]))
test_X = test_X.reshape((test_X.shape[0],1, test_X.shape[1]))
print(train_X.shape, train_y.shape, test_X.shape, test_y.shape)
```

```
(630, 1, 6) (630,) (99, 1, 6) (99,)
```

In the next step I defined a sequential LSTM model with 50 Epochs, the optimiser I chose for my model is ADAM and the loss per epoch was calculated in the form of mean squared error.

```
model = Sequential()
model.add(LSTM(50, input_shape=(train_X.shape[1], train_X.shape[2]),return_sequences= False))
model.add(Dropout(0.5))
# model.add(LSTM(50))
# model.add(Dropout(0.5))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

The next part of the code is to save the history of loss for each Epoch, to make a prediction, to invert the scaling for the forecast and also to store the min and max errors in mSE form.

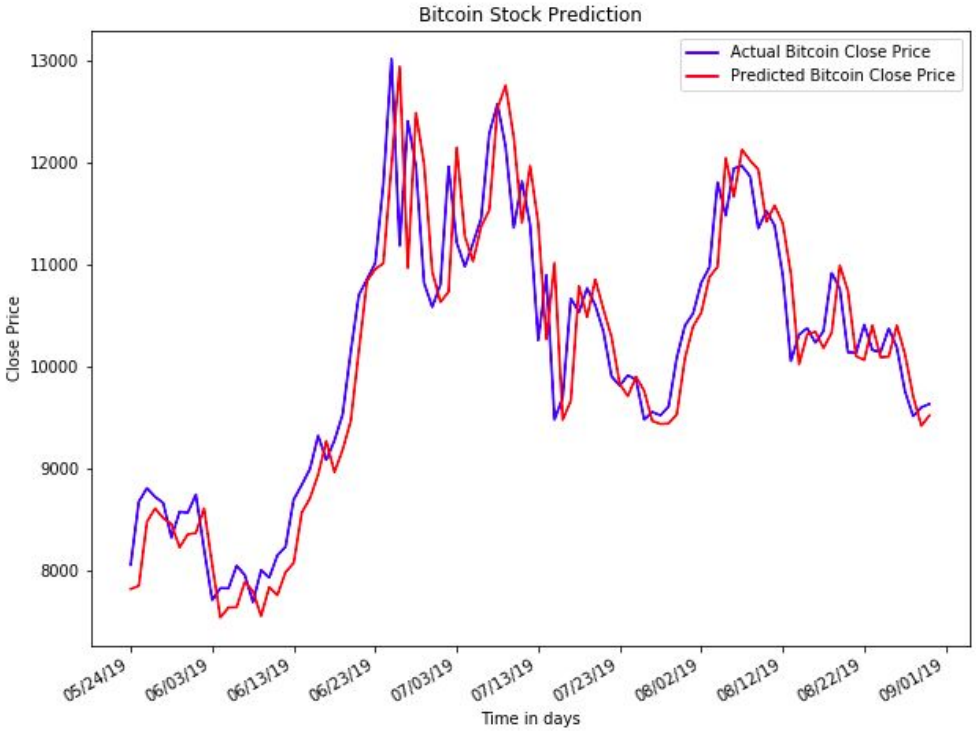
```
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()

# make a prediction
yhat = model.predict(test_X)
test_X = test_X.reshape((test_X.shape[0], test_X.shape[2]))
# invert scaling for forecast
inv_yhat = np.concatenate((yhat, test_X[:, 1:]), axis=1)
inv_yhat = scaler.inverse_transform(inv_yhat)
inv_yhat = inv_yhat[:,0]
# invert scaling for actual
test_y = test_y.reshape((len(test_y), 1))
inv_y = np.concatenate((test_y, test_X[:, 1:]), axis=1)
inv_y = scaler.inverse_transform(inv_y)
inv_y = inv_y[:,0]
# calculate RMSE
rmse = np.sqrt(mean_squared_error(inv_y, inv_yhat))
print('Test RMSE: %.3f' % rmse)
for xx in range(len(inv_y)):
    error = abs(inv_y[xx]-inv_yhat[xx])
    positive_error.append(error)
max_err= max(positive_error)
min_err =min(positive_error)
```

We now finally begin the training on the 630 dat points for 50 Epochs, below are the results of some Epochs.

```
Train on 630 samples, validate on 99 samples
Epoch 1/50
630/630 [==============================] - ETA: 0s - loss: 0.047 - 2s 3ms/step - loss: 0.0451 - val_loss: 0.0183
Epoch 2/50
630/630 [==============================] - 1s 1ms/step - loss: 0.0164 - val_loss: 0.0069
Epoch 3/50
630/630 [==============================] - 1s 1ms/step - loss: 0.0108 - val_loss: 0.0030
Epoch 4/50
630/630 [==============================] - ETA: 0s - loss: 0.006 - 1s 894us/step - loss: 0.0065 - val_loss: 0.00
19
Epoch 5/50
630/630 [==============================] - 1s 1ms/step - loss: 0.0047 - val_loss: 0.0013
Epoch 6/50
630/630 [==============================] - 1s 1ms/step - loss: 0.0040 - val_loss: 0.0012
Epoch 7/50
630/630 [==============================] - 1s 1ms/step - loss: 0.0034 - val_loss: 0.0012
Epoch 8/50
630/630 [==============================] - 1s 1ms/step - loss: 0.0036 - val_loss: 0.0012
Epoch 9/50
```

Next we test the data for the test dataset and obtain the following graph.
Here we can observe that the predicted price is very close to the actual price.



Lastly we calculate the error in the prediction which comes out to be around **5.8343**, which is very less and hence the prediction has come out well for this machine learning model and this multivariate data.

```
In [69]: error_list = list()
         for ii in range(len(inv_y)):
             err = (abs(inv_yhat[ii]-inv_y[ii])/inv_y[ii])*100
             error_list.append(err)
         sum(error_list)/len(error_list)

Out[69]: 5.834347229173019
```

Forecasting using ANN(Artificial Neural Network)

I used a basic Sequential model with as shown below, with batch size 8 and 50 epochs. The results from this model were not as good as the LSTMs and the MSE was also much larger.

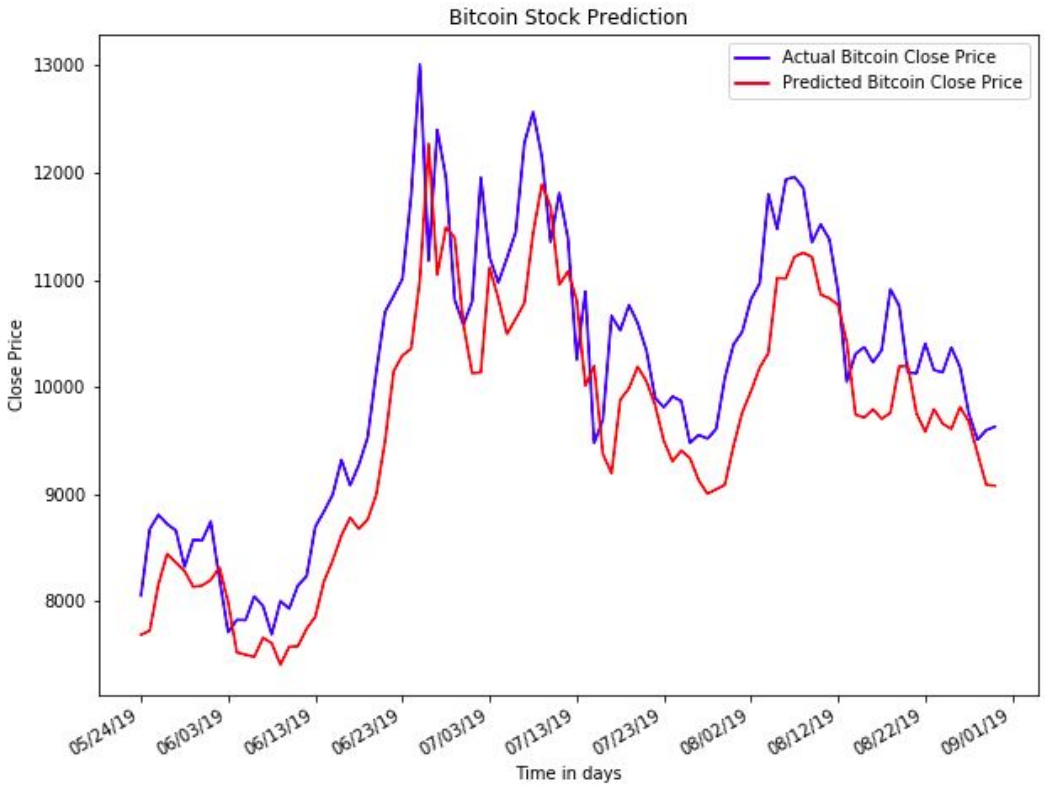
```
In [10]: model = Sequential([
             Dense(50, activation='relu', input_shape=(6,)),
             Dense(50, activation='relu'),
             Dense(1, activation='sigmoid'),
         ])

In [11]: model.compile(optimizer='adam',
                       loss='mse')

In [12]: hist = model.fit(train_X, train_y,
                          batch_size=8, epochs=50,
                          validation_data=(test_X, test_y))

         Train on 630 samples, validate on 99 samples
         Epoch 1/50
         630/630 [==============================] - 5s 8ms/step - loss: 0.0631 - val_loss: 0.0222
         Epoch 2/50
         630/630 [==============================] - 0s 554us/step - loss: 0.0089 - val_loss: 0.0132
         Epoch 3/50
         630/630 [==============================] - 0s 552us/step - loss: 0.0022 - val_loss: 0.0034
         Epoch 4/50
         630/630 [==============================] - ETA: 0s - loss: 0.001 - 0s 545us/step - loss: 0.0017 - val_loss: 0.002
         9
         Epoch 5/50
```

Following is the prediction made using the ANN model:



The RMSE after prediction was much larger than the LSTMs and SARIMA model.

```
# # calculate RMSE
rmse = np.sqrt(mean_squared_error(inv_y, inv_yhat))
print('Test RMSE: %.3f' % rmse)
```

```
Test RMSE: 526.597
```

Conclusion:

Following is the table for RMSE of the various models we used:

Models	**SARIMA(0,1,0)(0,1,1,12)**	**LSTMs**	**ANN**
Root Mean Squared Error	**0.05**	**5.8343**	**526.597**

From the table above we can say that the **SARIMA(0,1,0)(0,1,1,12)** has the least RMSE which is 0.05, then comes the **LSTMs** with RMSE 5.8343 and the last is **ANN** model with an accuracy of 526.597.

Thus, for this data which is small in number(means has less data points) we observe that SARIMA works better than the Neural Network Models RNNs and ANNs. Thus, the time series model is better for the short term forecasting whereas the Neural Networks perform better for the long term forecasting.
From the experiments we can say that SARIMA model yields better results for the short term forecasting than the LSTM model, but the LSTMs give much better results for the long term forecasting and also for a bigger dataset.

Applications:
This Bitcoin price prediction on the basis of the previous trends and seasonality helps us predict/forecast the future prices of the Bitcoin on the basis of which people can invest in Bitcoins. We can decide the best time to sale or purchase the Bitcoins on the basis of the price. We can also make out that how promising the Bitcoins are and what are the future prospects of the Bitcoins. This prediction is going to be very beneficial for the people who frequently invest in Bitcoins. This will reduce our probabilities of loss and also we will be able to plan our future investments.