**Introduction**

Time series analysis comprises methods for analyzing time series data in order to extract meaningful statistics and other characteristics of the data. Time series forecasting is the use of a model to predict future values based on previously observed values. Whether we wish to predict the trend in financial markets or electricity consumption, time is an important factor that must now be considered in our models. For example, it would be interesting to forecast at what hour during the day is there going to be a peak consumption in electricity, such as to adjust the price or the production of electricity.

**About Dataset**

The dataset used is stock market data of the Nifty-50 index from NSE (National Stock Exchange) India over the last 20 years (2000 - 2019)

The historic VWAP (Volume Weighted Average Price) is the target variable to predict. VWAP is a trading benchmark used by traders that gives the average price the stock has traded at throughout the day, based on both volume and price. Read more about the dataset:

I am using Reliance stock prices .

Inspired from -

**Reliance Industries Limited**

Reliance Industries Limited (RIL) is an Indian multinational conglomerate company headquartered in Mumbai, Maharashtra, India. Reliance owns businesses across India engaged in energy, petrochemicals, textiles, natural resources, retail, and telecommunications. Reliance is one of the most profitable companies in India,the largest publicly traded company in India by market capitalization,and the largest company in India as measured by revenue after recently surpassing the government-controlled Indian Oil Corporation.On 22 June 2020, Reliance Industries became the first Indian company to exceed US$150 billion in market capitalization after its market capitalization hit ₹11,43,667 crore on the BSE. The company is ranked 96th on the Fortune Global 500 list of the world's biggest corporations as of 2020.It is ranked 8th among the Top 250 Global Energy Companies by Platts as of 2016. Reliance continues to be India's largest exporter, accounting for 8% of India's total merchandise exports with a value of ₹1,47,755 crore and access to markets in 108 countries. Reliance is responsible for almost 5% of the government of India's total revenues from customs and excise duty. It is also the highest income tax payer in the private sector in India.

**This notebook will the cover -**

* Data Preparation
* Exploratory Data Analysis
* Feature Engineering
* AUTO-ARIMA Model
* Analyzing residuals
* Evaluating Model

**Data Preparation**

**Import Libraries**

pip install pmdarima

**import** pandas **as** pd

**import** matplotlib.pyplot **as** plt

**import** math

**import** seaborn **as** sns

**import** missingno **as** msno

**import** plotly.express **as** px

**import** plotly.graph\_objects **as** go

**import** matplotlib.dates **as** mdates

**import** scipy.stats

**from** sklearn.metrics **import** mean\_absolute\_error, mean\_squared\_error

**import** pylab

sns.set(style**=**'white')

**from** pmdarima **import** auto\_arima

**from** statsmodels.tsa.stattools **import** adfuller

**import** statsmodels.api **as** sm

**from** statsmodels.tsa.seasonal **import** seasonal\_decompose

**import** lightgbm **as** lgb

add Codeadd Markdown

**Import dataset**

df**=**pd.read\_csv('../input/nifty50-stock-market-data/RELIANCE.csv')

add Codeadd Markdown

**Converting Date into DateTime format**

df['Date']**=**pd.to\_datetime(df['Date'])

df.set\_index(['Date'],inplace**=True**)

df.head()

add Codeadd Markdown

**data summary**

df.describe()

df.shape

**Check for missing values**

**def** missing\_values\_table(df):

*# Total missing values*

mis\_val **=** df.isnull().sum()

*# Percentage of missing values*

mis\_val\_percent **=** 100 **\*** df.isnull().sum() **/** len(df)

*# Make a table with the results*

mis\_val\_table **=** pd.concat([mis\_val, mis\_val\_percent], axis**=**1)

*# Rename the columns*

mis\_val\_table\_ren\_columns **=** mis\_val\_table.rename(

columns **=** {0 : 'Missing Values', 1 : '% of Total Values'})

*# Sort the table by percentage of missing descending*

mis\_val\_table\_ren\_columns **=** mis\_val\_table\_ren\_columns[

mis\_val\_table\_ren\_columns.iloc[:,1] **!=** 0].sort\_values(

'% of Total Values', ascending**=False**).round(1)

*# Print some summary information*

print ("Your selected dataframe has " **+** str(df.shape[1]) **+** " columns.\n"

"There are " **+** str(mis\_val\_table\_ren\_columns.shape[0]) **+**

" columns that have missing values.")

*# Return the dataframe with missing information*

**return** mis\_val\_table\_ren\_columns

missing\_table**=**missing\_values\_table(df)

missing\_table

add Codeadd Markdown

* There are missing vales in Trades , Deliverable Volumne and % deliverable

add Codeadd Markdown

**Visualizing the locations of the missing data**

msno.matrix(df)

add Codeadd Markdown

* As you can see all the starting values are missing in columns Trades , Deliverable Volume and %Deliverble

add Codeadd Markdown

**Handling missing values**

df.Trades.plot()

df.Trades[:2850]

add Codeadd Markdown

* we don't have data of Trades on and before 31-05-2011

*# removing missing columns*

​

df.drop(['Trades','Deliverable Volume','%Deliverble'],axis**=**1,inplace**=True**)

​

**Features:**

1. Series: Here EQ stands for equity series of stock market.
2. Prev Close: The closing price of the stock for the day before.
3. Open,High, Low, Last, Close: The opening price, highest price, lowest price, last price and closing price of ICICI shares on the current day.
4. **VWAP**: Volume Weighted Average Price,the **target variable** to predict. VWAP is a trading benchmark used by traders that gives the average price the stock has traded at throughout the day, based on both volume and price.
5. Volume: Volume of shares traded on the current day.
6. Turnover: It is a measure of stock liquidity calculated by dividing the total number of shares traded over a period by the average number of shares outstanding for the period.
7. Trades: total number of trades on the current day.
8. Deliverable Volume: is the quantity of shares which actually move from one set of people to another set of people.
9. Deliverable(%): Deliverable volume in percentage.

**Exploratory Data Analysis**

**Plotting VWAP(Volume Weighted Average Price) over time**

fig **=** go.Figure([go.Scatter(x**=**df.index, y**=**df['VWAP'])])

fig.update\_layout(

autosize**=False**,

width**=**1000,

height**=**500,

title**=**'VWAP over time',

template**=**"simple\_white",

)

fig.update\_xaxes(title**=**"Date")

fig.update\_yaxes(title**=**"VWAP")

fig.show()

add Codeadd Markdown

* There is steady increase in prices upto year 2008
* Stock price fell after jan 2008 and attain pick again in may-june 2009 after that it fell again .

add Codeadd Markdown

**Visualising using KDEs**

Summarizing the data with Density plots to see where the mass of the data is located

sns.kdeplot(df['VWAP'],shade**=True**)

add Codeadd Markdown

* There are two picks in VWAP prices

add Codeadd Markdown

**VWAP in 2019**

fig **=** go.Figure([go.Scatter(x**=**df.loc['2019', 'VWAP'].index,y**=**df.loc['2020', 'VWAP'])])

fig.update\_layout(

autosize**=False**,

width**=**1000,

height**=**500,

title**=**'VWAP in 2019',

template**=**"simple\_white",

)

fig.update\_xaxes(title**=**"Date")

fig.update\_yaxes(title**=**"VWAP")

​

fig.show()

add Codeadd Markdown

* There was many dips in year 2019 like in May-June , Aug-Sept and in end of the Sept month

add Codeadd Markdown

**VWAP in 2020**

fig **=** go.Figure([go.Scatter(x**=**df.loc['2020', 'VWAP'].index,y**=**df.loc['2020', 'VWAP'])])

fig.update\_layout(

autosize**=False**,

width**=**1000,

height**=**500,

title**=**'VWAP in 2020',

template**=**"simple\_white",

)

fig.update\_xaxes(title**=**"Date")

fig.update\_yaxes(title**=**"VWAP")

fig.show()

add Codeadd Markdown

* as you can see there is deep in the month of march and april.
* Steady increase after month of april .

add Codeadd Markdown

**Open,close,High,low prices over time**

cols\_plot **=** ['Open', 'Close', 'High','Low']

axes **=** df[cols\_plot].plot(figsize**=**(11, 9), subplots**=True**)

**for** ax **in** axes:

ax.set\_ylabel('Daily trade')

add Codeadd Markdown

* All are following same pattern

add Codeadd Markdown

**Volume over Time**

fig **=** go.Figure([go.Scatter(x**=**df.index, y**=**df['Volume'])])

fig.update\_layout(

autosize**=False**,

width**=**1000,

height**=**500,

template**=**'simple\_white',

title**=**'Volume over time'

)

fig.update\_xaxes(title**=**"Date")

fig.update\_yaxes(title**=**"Volume")

fig.show()

add Codeadd Markdown

**Volume in 2020**

fig **=** go.Figure([go.Scatter(x**=**df.loc['2020', 'Volume'].index,y**=**df.loc['2020', 'Volume'])])

fig.update\_layout(

autosize**=False**,

width**=**1000,

height**=**500,

template**=**'simple\_white',

title**=**'Volume in 2020'

)

fig.update\_xaxes(title**=**"Date")

fig.update\_yaxes(title**=**"Volume")

fig.show()

add Codeadd Markdown

**Q-Q plot of VWAP**

used to determine whether dataset is distributed a certain way

scipy.stats.probplot(df.VWAP,plot**=**pylab)

pylab.show()

add Codeadd Markdown

* data is not normally distributed , however this is what we usually expect from timeseries

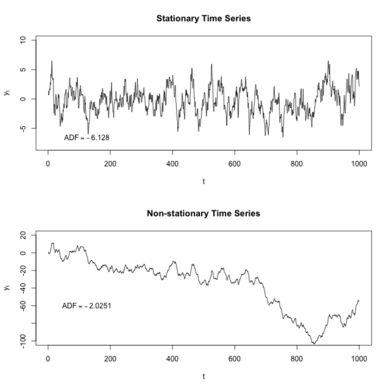
add Codeadd Markdown

**Stationarity**

add Codeadd Markdown

In the most intuitive sense, stationarity means that the statistical properties of a process generating a time series do not change over time. It does not mean that the series does not change over time, just that the way it changes does not itself change over time. The algebraic equivalent is thus a linear function, perhaps, and not a constant one; the value of a linear function changes as 𝒙 grows, but the way it changes remains constant — it has a constant slope; one value that captures that rate of change.

add Codeadd Markdown

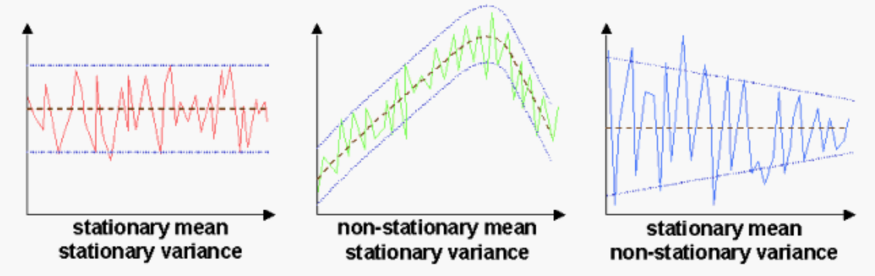


add Codeadd Markdown

Formally, the process {xᵢ ; i∈ℤ} is weakly stationary if:

1. The first moment of xᵢ is constant; i.e. ∀t, E[xᵢ]=𝜇
2. The second moment of xᵢ is finite for all t; i.e. ∀t, E[xᵢ²]<∞ (which also implies of course E[(xᵢ-𝜇)²]<∞; i.e. that variance is finite for all t)
3. The cross moment — i.e. the auto-covariance — depends only on the difference u-v; i.e. ∀u,v,a, cov(xᵤ, xᵥ)=cov(xᵤ₊ₐ, xᵥ₊ₐ)

add Codeadd Markdown



add Codeadd Markdown

Note - If you want to know more about stationarity , you can refer this [link](http://https/towardsdatascience.com/stationarity-in-time-series-analysis-90c94f27322#:~:text=In%20the%20most%20intuitive%20sense,not%20itself%20change%20over%20time.)

add Codeadd Markdown

**Check Stationarity i.e Dicky Fuller Test**

add Codeadd Markdown

The Augmented Dickey-Fuller test is a type of statistical test called a unit root test.

The intuition behind a unit root test is that it determines how strongly a time series is defined by a trend It uses an autoregressive model and optimizes an information criterion across multiple different lag values.

The null hypothesis of the test is that the time series can be represented by a unit root, that it is not stationary (has some time-dependent structure). The alternate hypothesis (rejecting the null hypothesis) is that the time series is stationary.

**Null Hypothesis (H0)**: If failed to be rejected, it suggests the time series has a unit root, meaning it is non-stationary. It has some time dependent structure.

**Alternate Hypothesis (H1)**: The null hypothesis is rejected; it suggests the time series does not have a unit root, meaning it is stationary. It does not have time-dependent structure.

We interpret this result using the p-value from the test. A p-value below a threshold (such as 5% or 1%) suggests we reject the null hypothesis (stationary), otherwise a p-value above the threshold suggests we fail to reject the null hypothesis (non-stationary).

p-value > 0.05: Fail to reject the null hypothesis (H0), the data has a unit root and is non-stationary. p-value <= 0.05: Reject the null hypothesis (H0), the data does not have a unit root and is stationary.

**def** dicky\_fuller\_test(x):

result **=** adfuller(x)

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))

**if** result[1]**>**0.05:

print("Fail to reject the null hypothesis (H0), the data is non-stationary")

**else**:

print("Reject the null hypothesis (H0), the data is stationary.")

dicky\_fuller\_test(df['VWAP'])

add Codeadd Markdown

**Seasonal Decompose**

**from** statsmodels.tsa.seasonal **import** seasonal\_decompose

**from** dateutil.parser **import** parse

​

plt.rcParams.update({'figure.figsize': (10,10)})

y **=** df['VWAP'].to\_frame()

​

​

*# Multiplicative Decomposition*

result\_mul **=** seasonal\_decompose(y, model**=**'multiplicative',period **=** 52)

​

*# Additive Decomposition*

result\_add **=** seasonal\_decompose(y, model**=**'additive',period **=** 52)

​

*# Plot*

plt.rcParams.update({'figure.figsize': (10,10)})

result\_mul.plot().suptitle('Multiplicative Decompose', fontsize**=**22)

result\_add.plot().suptitle('Additive Decompose', fontsize**=**22)

plt.show()

add Codeadd Markdown

* cyclic patter is shown in every 30 days (monthly)

add Codeadd Markdown

**Convert Stationary into Non Stationary**

add Codeadd Markdown

**Differencing**

df['vwap\_diff']**=**df['VWAP']**-**df['VWAP'].shift(1)

fig **=** go.Figure([go.Scatter(x**=**df.index,y**=**df.VWAP)])

fig.update\_layout(

autosize**=False**,

width**=**1000,

height**=**500,

template**=**'simple\_white',

title**=**'VWAP over time ')

fig.show()

fig **=** go.Figure([go.Scatter(x**=**df.index,y**=**df.vwap\_diff)])

fig.update\_layout(

autosize**=False**,

width**=**1000,

height**=**500,

template**=**'simple\_white',

title**=**'difference VWAP over time ')

fig.show()

add Codeadd Markdown

* We don't actually need to convert the time series data into stationary data. For study purpose,I have explained how to check stationarity and how to convert non-stationary data into stationary data

add Codeadd Markdown

**Plotting ACF and PACF**

add Codeadd Markdown

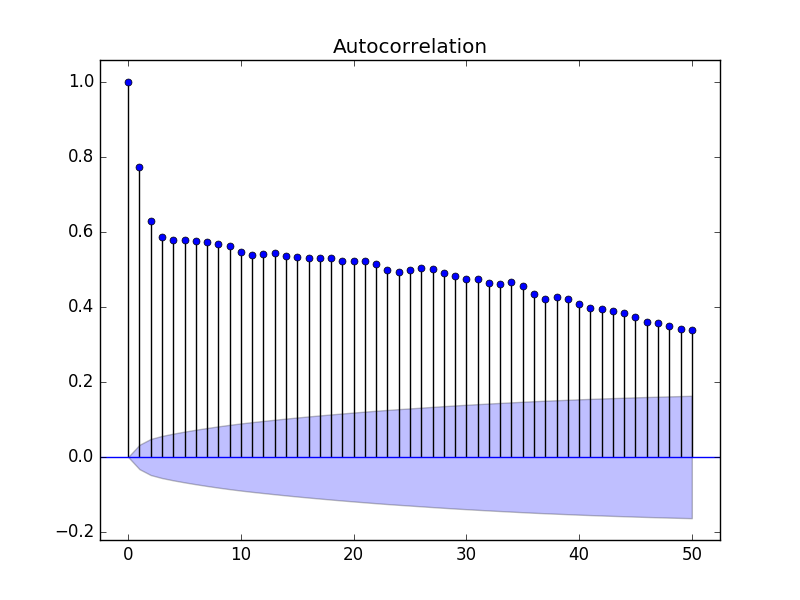
**Autocorrelation** and **partial autocorrelation** plots are heavily used in time series analysis and forecasting.

These are plots that graphically summarize the strength of a relationship with an observation in a time series with observations at prior time steps.

**Statistical correlation** summarizes the strength of the relationship between two variables.

We can calculate the correlation for time series observations with observations with previous time steps, called lags. Because the correlation of the time series observations is calculated with values of the same series at previous times, this is called a **serial correlation, or an autocorrelation.**

A plot of the autocorrelation of a time series by lag is called the AutoCorrelation Function, or the acronym ACF. This plot is sometimes called a **correlogram or an autocorrelation plot**.



A **partial autocorrelation** is a summary of the relationship between an observation in a time series with observations at prior time steps with the relationships of intervening observations removed.

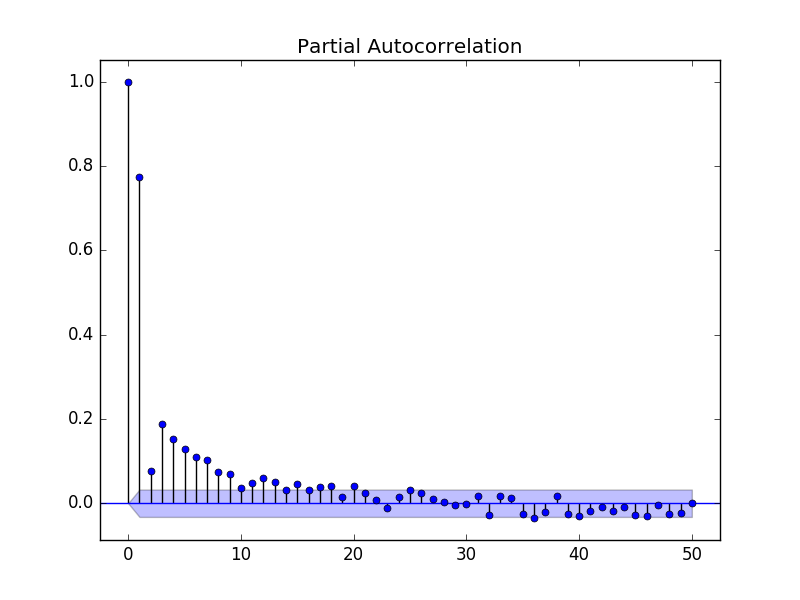
The autocorrelation for an observation and an observation at a prior time step is comprised of both the direct correlation and indirect correlations. These indirect correlations are a linear function of the correlation of the observation, with observations at intervening time steps.

It is these indirect correlations that the partial autocorrelation function seeks to remove. Without going into the math, this is the intuition for the partial autocorrelation.

A **partial autocorrelation** is a summary of the relationship between an observation in a time series with observations at prior time steps with the relationships of intervening observations removed.

The autocorrelation for an observation and an observation at a prior time step is comprised of both the direct correlation and indirect correlations. These indirect correlations are a linear function of the correlation of the observation, with observations at intervening time steps.

It is these indirect correlations that the partial autocorrelation function seeks to remove. Without going into the math, this is the intuition for the partial autocorrelation.



sm.graphics.tsa.plot\_acf(df['VWAP'].iloc[1:], lags**=**40,title**=**'auto correlation of VWAP',zero**=False**)

plt.show()

sm.graphics.tsa.plot\_acf(df['vwap\_diff'].iloc[7:], lags**=**40,title**=**'auto correlation of difference VWAP',zero**=False**)

plt.show()

sm.graphics.tsa.plot\_pacf(df['VWAP'].iloc[1:], lags**=**40,title**=**'partial auto correlation of VWAP',zero**=False**)

plt.show()

sm.graphics.tsa.plot\_pacf(df['vwap\_diff'].iloc[1:], lags**=**40,title**=**'partial autocorrelation of difference VWAP ',zero**=False**)

plt.show()

add Codeadd Markdown

**Feature Engineering**

Adding lag values of High, Low, Volume,Turnover, will use three sets of lagged values, one previous day, one looking back 7 days and another looking back 30 days as a proxy for last week and last month metrics.

df.head()

df**=**df.reset\_index()

lag\_features **=** ["High", "Low", "Volume", "Turnover","Close"]

window1 **=** 3

window2 **=** 7

window3 **=** 30

​

df\_rolled\_3d **=** df[lag\_features].rolling(window**=**window1, min\_periods**=**0)

df\_rolled\_7d **=** df[lag\_features].rolling(window**=**window2, min\_periods**=**0)

df\_rolled\_30d **=** df[lag\_features].rolling(window**=**window3, min\_periods**=**0)

​

df\_mean\_3d **=** df\_rolled\_3d.mean().shift(1).reset\_index().astype(np.float32)

df\_mean\_7d **=** df\_rolled\_7d.mean().shift(1).reset\_index().astype(np.float32)

df\_mean\_30d **=** df\_rolled\_30d.mean().shift(1).reset\_index().astype(np.float32)

​

df\_std\_3d **=** df\_rolled\_3d.std().shift(1).reset\_index().astype(np.float32)

df\_std\_7d **=** df\_rolled\_7d.std().shift(1).reset\_index().astype(np.float32)

df\_std\_30d **=** df\_rolled\_30d.std().shift(1).reset\_index().astype(np.float32)

​

**for** feature **in** lag\_features:

df[f"{feature}\_mean\_lag{window1}"] **=** df\_mean\_3d[feature]

df[f"{feature}\_mean\_lag{window2}"] **=** df\_mean\_7d[feature]

df[f"{feature}\_mean\_lag{window3}"] **=** df\_mean\_30d[feature]

df[f"{feature}\_std\_lag{window1}"] **=** df\_std\_3d[feature]

df[f"{feature}\_std\_lag{window2}"] **=** df\_std\_7d[feature]

df[f"{feature}\_std\_lag{window3}"] **=** df\_std\_30d[feature]

​

df.fillna(df.mean(), inplace**=True**)

​

df.set\_index("Date", drop**=False**, inplace**=True**)

​

df.Date **=** pd.to\_datetime(df.Date, format**=**"%Y-%m-%d")

df["month"] **=** df.Date.dt.month

df["week"] **=** df.Date.dt.week

df["day"] **=** df.Date.dt.day

df["day\_of\_week"] **=** df.Date.dt.dayofweek

​

df.head()

​

df\_train **=** df[df.Date **<** "2019"]

df\_valid **=** df[df.Date **>=** "2019"]

​

exogenous\_features **=** ["High\_mean\_lag3", "High\_std\_lag3", "Low\_mean\_lag3", "Low\_std\_lag3",

"Volume\_mean\_lag3", "Volume\_std\_lag3", "Turnover\_mean\_lag3",

"Turnover\_std\_lag3","High\_mean\_lag7", "High\_std\_lag7", "Low\_mean\_lag7", "Low\_std\_lag7",

"Volume\_mean\_lag7", "Volume\_std\_lag7", "Turnover\_mean\_lag7",

"Turnover\_std\_lag7","High\_mean\_lag30", "High\_std\_lag30", "Low\_mean\_lag30", "Low\_std\_lag30",

"Volume\_mean\_lag30", "Volume\_std\_lag30", "Turnover\_mean\_lag30",

"Close\_mean\_lag3", "Close\_mean\_lag7","Close\_mean\_lag30","Close\_std\_lag3","Close\_std\_lag7","Close\_std\_lag30",

"Turnover\_std\_lag30","month","week","day","day\_of\_week"]

​

df\_valid['Date'].describe()

**AutoRegressor:**

A model that uses the dependent relationship between an observation and some number of lagged observations.

"We can use statistical measures to calculate the correlation between the output variable and values at previous time steps at various different lags. The stronger the correlation between the output variable and a specific lagged variable, the more weight that autoregression model can put on that variable when modeling."-<https://machinelearningmastery.com/autoregression-models-time-series-forecasting-python/>

AR(1) model  
Rt = μ + ϕRt-1 + εt

As RHS has only one lagged value(Rt-1)this is called AR model of order 1 where μ is mean and ε is noise at time t If ϕ = 1, it is random walk. Else if ϕ = 0, it is white noise. Else if -1 < ϕ < 1, it is stationary. If ϕ is -ve, there is men reversion. If ϕ is +ve, there is momentum.

AR(2) model  
Rt = μ + ϕ1Rt-1 + ϕ2Rt-2 + εt

AR(3) model  
Rt = μ + ϕ1Rt-1 + ϕ2Rt-2 + ϕ3Rt-3 + εt

**from** statsmodels.tsa.ar\_model **import** AutoReg

model **=** AutoReg(df\_train.VWAP,lags**=**3, exog**=**df\_train[exogenous\_features])

res **=** model.fit()

print(res.summary())

print("μ={} ,ϕ={}".format(res.params[0],res.params[1]))

res **=** model.fit(cov\_type**=**"HC0")

print(res.summary())

print("μ={} ,ϕ={}".format(res.params[0],res.params[1]))

fig **=** plt.figure(figsize**=**(16,9))

fig **=** res.plot\_diagnostics(fig**=**fig, lags**=**30)

add Codeadd Markdown

**AUTO-ARIMA Model**

model **=** auto\_arima(df\_train.VWAP, exogenous**=**df\_train[exogenous\_features], trace**=True**, error\_action**=**"ignore", suppress\_warnings**=True**)

model.fit(df\_train.VWAP, exogenous**=**df\_train[exogenous\_features])

​

forecast **=** model.predict(n\_periods**=**len(df\_valid), exogenous**=**df\_valid[exogenous\_features])

df\_valid["Forecast\_ARIMAX"] **=** forecast

model.summary()

df\_valid[["VWAP", "Forecast\_ARIMAX"]].plot(figsize**=**(14, 7))

print("RMSE of Auto ARIMAX:", np.sqrt(mean\_squared\_error(df\_valid.VWAP, df\_valid.Forecast\_ARIMAX)))

print("\nMAE of Auto ARIMAX:", mean\_absolute\_error(df\_valid.VWAP, df\_valid.Forecast\_ARIMAX))

The Auto ARIMAX model seems to do a fairly good job in predicting the stock price given data till the previous day. Can other models beat this benchmark?

**Facebook Prophet**

Prophet is an open-source time series model developed by Facebook. It was released in early 2017. An exerpt from the homepage:

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

Read more about Prophet: <https://facebook.github.io/prophet/>

I also shared a starter code [Prophet's Prophecy](https://www.kaggle.com/rohanrao/ashrae-prophet-s-prophecy) for using Prophet in the ASHRAE competition on Kaggle.

Note that the default parameters are used for Prophet. They can be tuned to improve the results.

**from** fbprophet **import** Prophet

​

model\_fbp **=** Prophet()

**for** feature **in** exogenous\_features:

model\_fbp.add\_regressor(feature)

​

model\_fbp.fit(df\_train[["Date", "VWAP"] **+** exogenous\_features].rename(columns**=**{"Date": "ds", "VWAP": "y"}))

​

forecast **=** model\_fbp.predict(df\_valid[["Date", "VWAP"] **+** exogenous\_features].rename(columns**=**{"Date": "ds"}))

df\_valid["Forecast\_Prophet"] **=** forecast.yhat.values

model\_fbp.plot\_components(forecast)

df\_valid[["VWAP", "Forecast\_ARIMAX", "Forecast\_Prophet"]].plot(figsize**=**(14, 7))

print("RMSE of Auto ARIMAX:", np.sqrt(mean\_squared\_error(df\_valid.VWAP, df\_valid.Forecast\_ARIMAX)))

print("RMSE of Prophet:", np.sqrt(mean\_squared\_error(df\_valid.VWAP, df\_valid.Forecast\_Prophet)))

print("\nMAE of Auto ARIMAX:", mean\_absolute\_error(df\_valid.VWAP, df\_valid.Forecast\_ARIMAX))

print("MAE of Prophet:", mean\_absolute\_error(df\_valid.VWAP, df\_valid.Forecast\_Prophet))

add Codeadd Markdown

**LightGBM**

Time series problems are popularly converted into a tabular i.i.d. structure and fed into boosting models like [LightGBM](https://lightgbm.readthedocs.io/en/latest/) and [XGBoost](https://xgboost.readthedocs.io/en/latest/).

There is loss of information in terms of knowing the order of data points in the time series but it can be circumvented by the datetime features to capture this information to some extent.

Note that the default parameters are used for LightGBM. They can be tuned to improve the results.

params **=** {"objective": "regression"}

​

dtrain **=** lgb.Dataset(df\_train[exogenous\_features], label**=**df\_train.VWAP.values)

dvalid **=** lgb.Dataset(df\_valid[exogenous\_features])

​

model\_lgb **=** lgb.train(params, train\_set**=**dtrain)

​

forecast **=** model\_lgb.predict(df\_valid[exogenous\_features])

df\_valid["Forecast\_LightGBM"] **=** forecast

df\_valid[["VWAP", "Forecast\_ARIMAX", "Forecast\_Prophet", "Forecast\_LightGBM"]].plot(figsize**=**(14, 7))

print("RMSE of Auto ARIMAX:", np.sqrt(mean\_squared\_error(df\_valid.VWAP, df\_valid.Forecast\_ARIMAX)))

print("RMSE of Prophet:", np.sqrt(mean\_squared\_error(df\_valid.VWAP, df\_valid.Forecast\_Prophet)))

print("RMSE of LightGBM:", np.sqrt(mean\_squared\_error(df\_valid.VWAP, df\_valid.Forecast\_LightGBM)))

print("\nMAE of Auto ARIMAX:", mean\_absolute\_error(df\_valid.VWAP, df\_valid.Forecast\_ARIMAX))

print("MAE of Prophet:", mean\_absolute\_error(df\_valid.VWAP, df\_valid.Forecast\_Prophet))

print("MAE of LightGBM:", mean\_absolute\_error(df\_valid.VWAP, df\_valid.Forecast\_LightGBM))

add Codeadd Markdown

ARIMA, Prophet and LightGBM models give a fair rmse and mae score. Let's see if deep learning models can do a better job in fitting to our dataset.

add Codeadd Markdown

**Deep Learning Models**

Let's explore some deep learning models on the dataset. We'll try LTSM and RNN. RNN, a model designed for allowing information to persist in short term memory to predict subsequent values, should be best suited to our dataset, since we can see that autocorrelations don't show signs of seasonality, it tends to be manipulated by recent previous prices. We'll also apply LTSM, the special RNN model, which might reveal some interesting long term dependencies or relations in the dataset. Let's explore!

RNN performs rather poorly, in comparison to LTSM model, Though the RMSE score turned out to be the same approximately, MAE has increased by 3-4%, as we can see the results from the graphs above. We can say that, when we move from RNN to LSTM, we are introducing more & more controlling knobs, which control the flow and mixing of Inputs as per trained Weights. And thus, bringing in more flexibility in controlling the outputs. Thus, we obtain better results in LTSM.

add Codeadd Markdown

**Analyzing residuals**

**Evaluating Model**

add Codeadd Markdown

**Conclusions and Tips**

* Auto ARIMAX is a great baseline model but newer algorithms like Facebook's Prophet are extremely powerful and are getting cleverer by the day. Don't feel afraid to try out new techniques.
* Setting up an appropriate validation framework is extremely important. It enables you to try and experiment various models and objectively compare them.
* Lag-based features are very useful in providing trends information about the time series data. Rolling statistics are a common way of generating these.
* Exogenous regressors help in providing external information about the time series. They tend to be very important in most models.
* Boosting models like LightGBM are constrained to predict within the range of values of the target variable in the training data and don't extrapolate when there is strong trend.
* Converting a time series to stationary and then modelling is a common approach for building solutions and can significantly improve results.