**CAPSTONE PROJECT ON**

**NIFTY50 STOCK CLOSE PRICE PREDICTION**

**CHAPTER 1- OBJECTIVES AND INTRODUCTION**

1.1 OBJECTIVE :

1. To perform time series analysis on the NIFTY stock price and forecasting using univariate ARIMA and ARIMAX modeling technique.
2. We further simplified the problem to predict the direction of Close price movements in the next N days.This is represented as a classification task where there are two possible outcomes (either the index went up in the next day or it went down).

1.2 INTRODUCTION:

1.2.1 NIFTY

The Nifty 50 is a stock market index comprised of the shares of 50 of India's most well-diversified firms, covering various economic sectors such as financial services, engineering, pharmaceuticals, and information technology. The weighting of these 50 stocks is determined by their free float market capitalization.

This is a time-series activity which we come across in our daily lives.

1.2.2 Time Series

Time series mean that a series of data points indexed in time order. The following are some of the most frequently asked questions: what will happen with our metrics in the next day/week/month/etc., how many users will instal our app, how much time will they spend online, how many actions will users complete, and so on. We can address these prediction tasks in a variety of ways, depending on the situation.

Depending on the appropriate quality of the prediction, the duration of the forecast period, and, of course, the amount of time we have to select features and tune parameters to achieve desired results, we can approach these prediction tasks in a variety of ways.

Forecasting : The science of forecasting is the art of predicting the future. Businesses can use historical data to consider patterns, make predictions about what will happen and when, and then incorporate the knowledge into their future plans for everything from product demand to marketing.

Many of the areas that naturally generate time series data have forecasting issues. Retail sales, medical research, power planning, sensor network tracking, financial analysis, social activity mining, and database systems are examples of these fields. Forecasting, for example, is critical to automating and optimising operating processes in most industries so that data-driven decisions can be made.

To solve such problem we can have two kinds of approaches

1. Time Series Approach
2. Machine Learning Approach

The next chapters have the following sections which are the steps involved for solving the forecasting problem,

Section 1 - Data collection

Section 2 - Data preparation

Section 3 - Exploratory data analysis

Section 4 - Feature Engineering

Section 5 - Working different models

Section 6 - Evaluating model

**CHAPTER 2 - DATA COLLECTION AND DATA PREPARATION**

2.1 Data Collection

The goal of the forecast is to find future observations today. To do that we require relevant and accurate data. Therefore data collection is an important step.

The Data of Nifty50 is collected from the National Stock Exchange price website. <https://www1.nseindia.com/products/content/equities/indices/historical_index_data.htm>

One can also get the past 15 years of data from yahoo finance website <https://in.finance.yahoo.com/quote/%5ENSEI/history/>.

Now we have our data in a .csv file format.

Here in the performed analysis 20 years of data with four features high, low, open and close is used.

Description of columns in the file:

Date — Date of trade

Open — The open is the starting period of trading on a securities exchange or organized over-the-counter market.

High — Highest price at which a stock traded during the course of the trading day.

Low — Lowest price at which a stock traded during the course of the trading day.

Close — The close is a reference to the end of a trading session in the financial markets when the markets close for the day.

Mount the drive and load the csv file

# Mounting drive

from google.colab import drive

drive.mount('/content/drive')

missing\_values = ['N/a', 'na', 'np-nan']

After mounting the drive the next step is to import the required libraries. Python has a wide number of libraries which makes the work easier. Here pandas, numpy, matlplotlib, seaborn, math, pylab, pmdarima, prophet etc.

#Loading data

nifty = pd.read\_csv("/content/drive/MyDrive/Almabetter/Cohort Nilgiri/capstone-2/ Nifty50\_data.csv", na\_values= missing\_values, header=0, index\_col=0, parse\_dates=True, squeeze=True)

The parameter parse\_dates makes sure that Date column is Datetime format.

2.2 Data Preparation

Once you have raw data, you must deal with issues such as missing data and ensure that the data is prepared for forecasting models in such a way that it is amendable to them.

Handling missing values :

Values that are reported as missing may be due to a variety of factors. The absence of a transaction, as well as potential calculation errors, may result in missing values.

Check for how many missing values are there in the data

nifty.isnull().sum()

There are 35 missing values found in which the entire row is null.

We can opt any of two methods in common.

The first method first is simply to drop the values as the observations with null values are low in number.

nifty.dropna(inplace=True)

The other method that can be used is to use interpolate. If one opts not to drop this can be used.

nifty = nifty.interpolate()

In which the null value is filled by the average of above and below values. This method can be used here because the values of today and tomorrow are almost in the same range.

**CHAPTER 3- EXPLORATORY DATA ANALYSIS**

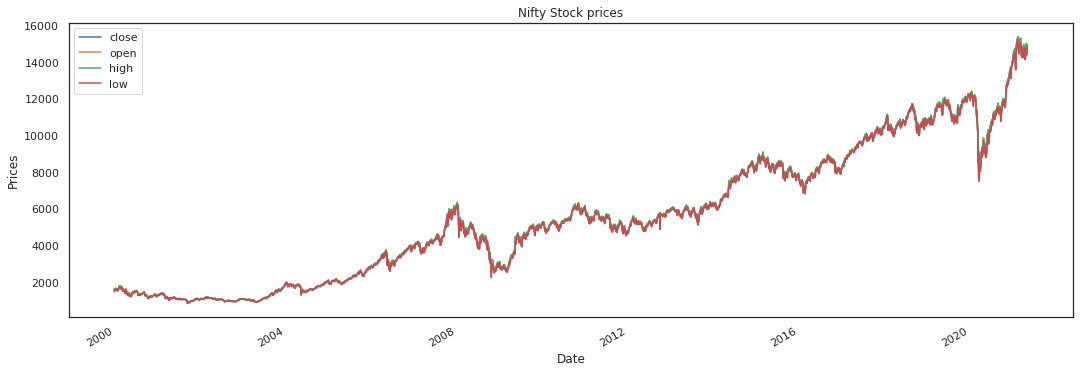
The primary goal of EDA is to support in the analysis of data prior to making any conclusions. It may aid in the detection of apparent errors, as well as a deeper understanding of data patterns, the detection of outliers or anomalous events, and the discovery of interesting relationships between variables.

3.1 Plotting a multi line plot

# Plotting a multi line plot

plt.figure(figsize=(36, 10))

lines= nifty[:].plot.line()



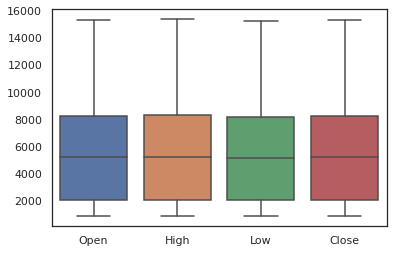
From the above graphs, we have observed that-

There was a drastic drop in stock prices in the 2007-2009 period.This can be attributed to the Great Recession that happened during this period. Also, there is a drop in stock prices in the year 2016. This can be attributed to Demonetisation drive by the central government.Again ,there is a drastic drop in stock prices in 2020. This is due to the global breakdown amid coronavirus pandemic induced lockdown in India.By the end of 2020, the stock price started rising.This can be attributed to the lifting of lockdown in the country and across the world.

3.2 Box Plot

# box plot

sns.boxplot(data=nifty[['Open','High','Low','Close']])

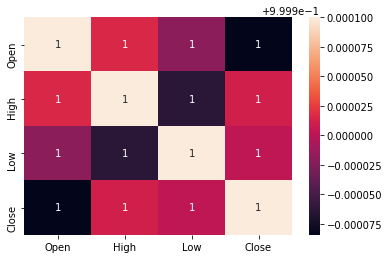


From the above box plots we can infer that the data of the features of a taken are more or less similar. There are no outlier observations which needs to be appreciated.

3.3 Correlation Plot

# Get correlation among different features viz. Open,High,Low and Close

sns.heatmap(nifty.corr(),annot =True)



The correlation seen between these variables is almost 1.

Understanding more about the Close feature as our objective is to predict the Close feature and Target variable.

3.4 Line plot

#plot close price

close = nifty['Close']

plt.figure(figsize=(6,4))

plt.grid(True)

plt.xlabel('Date')

plt.ylabel('Close Prices')

plt.plot( close.index, close.values, marker='', color='#FF2511', linewidth=3)

plt.title('Nifty closing price')

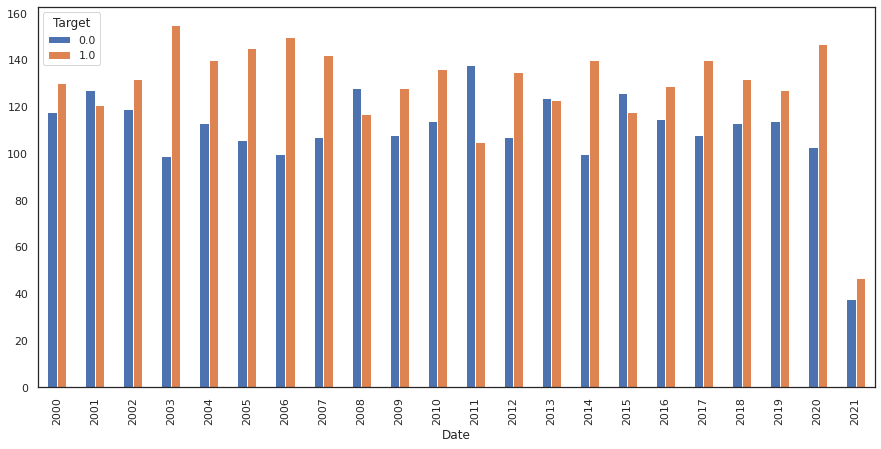
plt.show()



3.5 Bar plot

GB = final\_df.groupby([final\_df.index.year])

GB['Target'].value\_counts().unstack().plot(kind= 'bar',figsize=(15,7))



This bar plot represents the total number of up and down seen in each year of past 21 years.

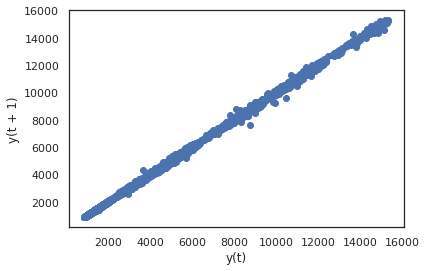
3.6 Lag plot

# Lag plot

from pandas.plotting import lag\_plot

lag\_plot(df['Close'])

plt.show()



Lag Scatter plots : Relation between a observation and its previous observation.

Previous observations are termed as lags. Lag at one step is lag1 and two steps is lag2. Lag\_plot plots the observation at time t on the x-axis and the lag1 observation (t-1) on the y-axis.

In time series problem we need to make sure that our current day data is related to previous data points and there data points should not be random. If data points are randomly selscted that it become hard to forcast.

**CHAPTER 4 : FEATURE ENGINEERING**

The features we have are low in number and we don't have that data of the day when we actually want to do prediction. So here comes the importance of feature engineering

Here no.of features are increased using rolling, shift, mean, standard deviation

def moving\_avg(df,col, day):

var\_name = col + str(day)

df[var\_name + '\_ma']= df[col]-df[col].rolling(window=day,min\_periods=1).mean()

df[var\_name+ '\_ewma']= df[col]-df[col].ewm(com=day).mean()

return df

# finding mean average for

days= [3,7,15,30]

cols= ['High','Low',"Open",'Close']

for col in cols:

for day in days:

moving\_avg(final\_df,col,day)

exogenous\_features= final\_df.columns.drop(['Open','High','Low','Close','Close First Difference'])

**CHAPTER 5: WORKING WITH DIFFERENT MODELS**

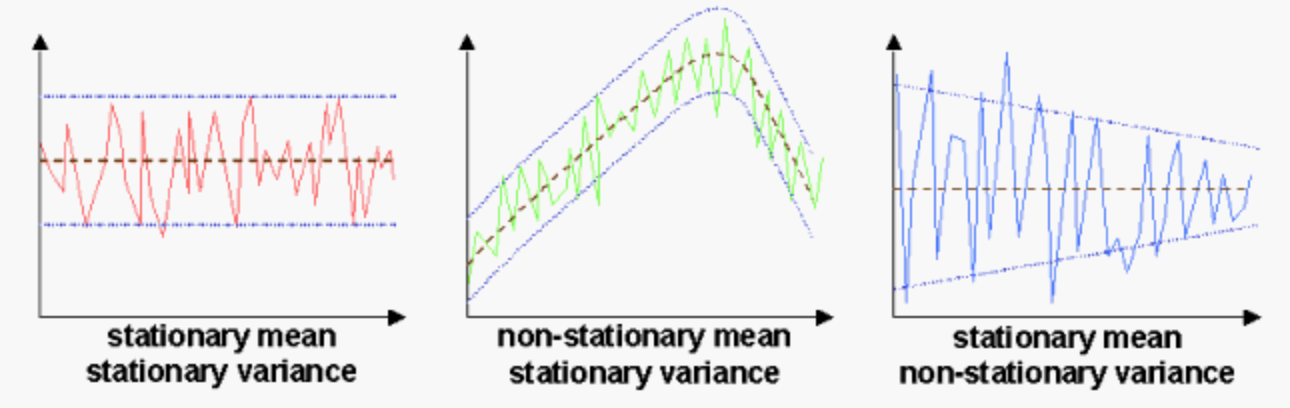
We have various models and approaches to perform forecasting.

1. Time-Series Approach :
2. Machine- Learning Approach

5.1 Time-Series Approach

Checking Stationarity :

Before we begin modelling, it's worth noting that one of the most significant properties of time series is stationarity. If a process is stationary, it means that its statistical properties, such as mean and variance, do not change over time. (Homoscedasticity refers to the consistency of variance.) The covariance function should not be affected by time; instead, it should be affected only by the distance between observations.



On a stationary series, making predictions is simple since we can conclude that future statistical properties would be similar to those currently observed.

The majority of time-series models attempt to predict certain properties in some way (mean or variance, for example). If the series was not stationary, future predictions will be incorrect.

The Dickey-Fuller test for time series stationarity is based on the principle of (testing the presence of a unit root). We name such series as integrated of order 1 if we can get a stationary series from a non-stationary series by using the first difference.

Dicky fuller test helps one to check on stationarity. In this it is assumed that the null hypothesis H0, ‘data is stationary’. If the p value is greater than 0.05 we fail to reject H0 else H0 is rejected which is desired.

#Function for performing dicky fuller test

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.")

The null hypothesis fails to get rejected which shows it is non stationary.

# function to plot auto-correlation and partial auto-correlation plots

def cor\_plots(x):

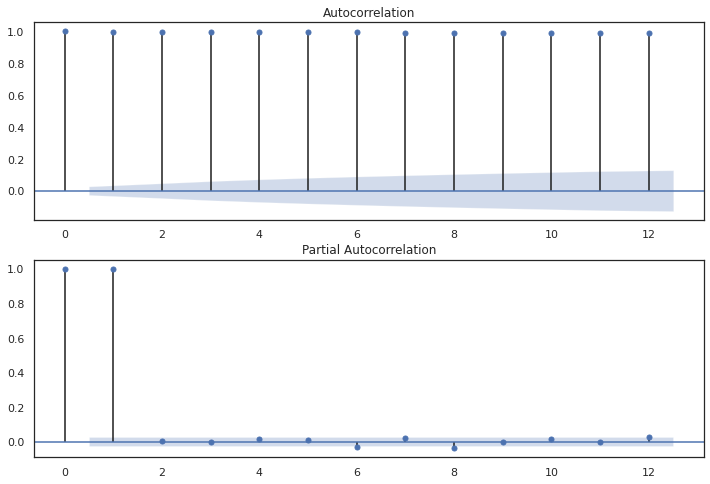
fig = plt.figure(figsize=(12,8))

ax1 = fig.add\_subplot(211)

fig = sm.graphics.tsa.plot\_acf(nifty[x].dropna(),lags=12,ax=ax1)

ax2 = fig.add\_subplot(212)

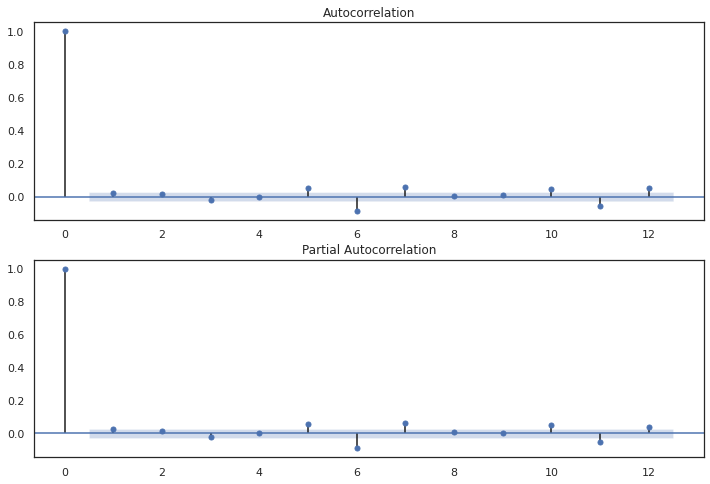
fig = sm.graphics.tsa.plot\_pacf(nifty[x].dropna(),lags=12,ax=ax2)



The variance is constant, and the mean is also slightly stable. The only remaining issue is seasonality, which must be addressed prior to modelling. To do so, use the "seasonal differencing," which is simply the series subtracted from itself with a lag equal to the seasonal period.

# Differencing

nifty['Close\_1'] = nifty['Close'] - nifty['Close'].shift(1)



It is stationary, according to the Dickey-Fuller measure, and the number of large peaks in ACF has decreased. Finally, we can begin modelling!

# Close variable over time on differencing

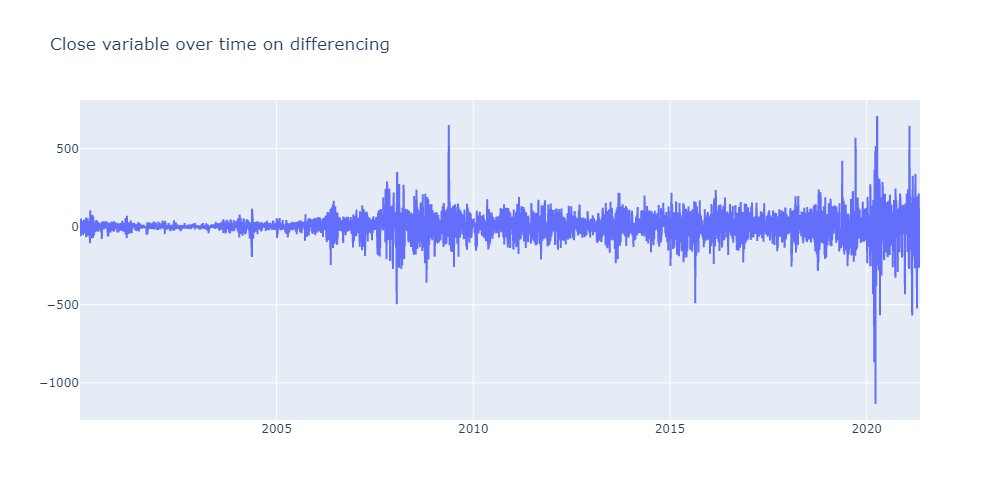
import plotly.graph\_objects as go

fig = go.Figure([go.Scatter(x=nifty.index,y=nifty['Close\_1'])])

fig.update\_layout(width=1000, height=500,

title='Close variable over time on differencing ')

fig.show()



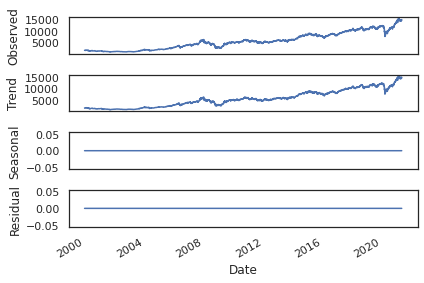
Seasonal Decomposition

# decomposition

result = seasonal\_decompose(df['Close'], model='additive',freq=1)

result.plot()

plt.show()



We will decompose the time series to find the Trend, Seasonality , residual in our data.

We can have two decomposition models

Additive: = Trend + Seasonal + Random

Multiplicative: = Trend \* Seasonal \* Random

ARIMA-family

We will explain this model by building up letter by letter. SARIMA(p, d, q)(P, D, Q, s), Seasonal Auto-regression Moving Average model:

AR - autoregression model i.e. regression of the time series onto itself. The basic assumption is that the current series values depend on its previous values with some lag (or several lags). The maximum lag in the model is referred to as p. To determine the initial p, you need to look at the PACF plot and find the biggest significant lag after which most other lags become insignificant.

MA - moving average model. Without going into too much detail, this models the error of the time series, again with the assumption that the current error depends on the previous with some lag, which is referred to as q. The initial value can be found on the ACF plot with the same logic as before.

Let's combine our first 4 letters:

AR(p) + MA(q) = ARMA(p, q)

What we have here is the Autoregressive–moving-average model! If the series is stationary, it can be approximated with these 4 letters. Let's continue.

I(d) - order of integration. This is simply the number of nonseasonal differences needed to make the series stationary. In our case, it's just 1 because we used first differences.

Adding this letter to the four gives us the $ARIMA$ model which can handle non-stationary data with the help of nonseasonal differences. Great, one more letter to go!

S(s) - this is responsible for seasonality and equals the season period length of the series

With this, we have three parameters: (P, D, Q)

P - order of autoregression for the seasonal component of the model, which can be derived from PACF. But you need to look at the number of significant lags, which are the multiples of the season period length. For example, if the period equals 24 and we see the 24-th and 48-th lags are significant in the PACF, that means the initial P should be 2.

Q - similar logic using the ACF plot instead.

D - order of seasonal integration. This can be equal to 1 or 0, depending on whether seasonal differences were applied or not.

Now we can test for various models.

1. Auto-regression model
2. ARIMA
3. ARIMAX
4. Facebook Prophet

**5.2.1 Auto-regression model**

# Auto regression model

from statsmodels.tsa.ar\_model import AutoReg

df\_train = df[df.index < "2019"]

df\_valid = df[df.index >= "2019"]

model = AutoReg(nifty\_train.Close,lags=3, exog=nifty\_train[exogenous\_features])

res = model.fit()

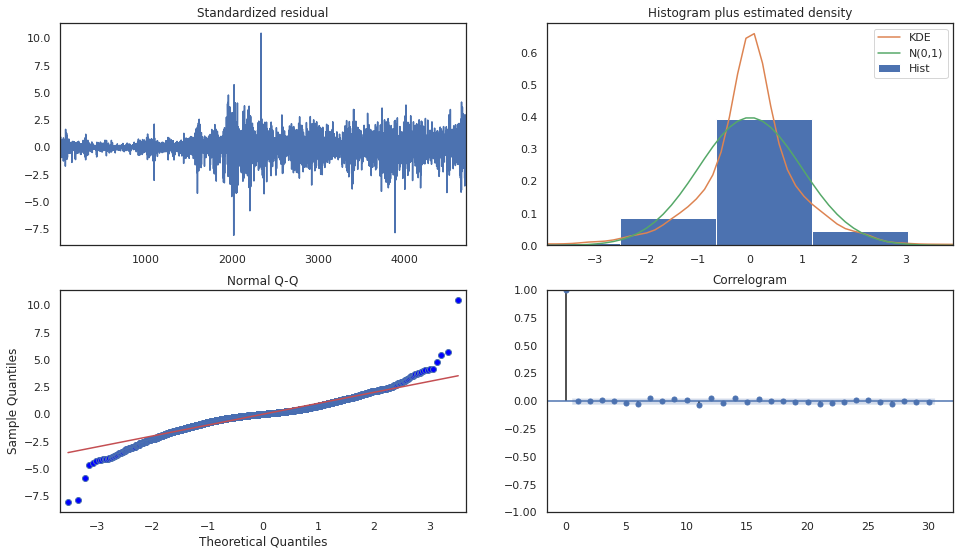
print(res.summary())

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

# Plot diagnostics of auto-regression model

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

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



Interpretation of above plots :

To ensure that the residuals of our model are uncorrelated and normally distributed with zero-mean.

In this case, our model diagnostics suggests that the model residuals are normally distributed based on the following.

Residuals are nearly normally distributed as KDE and N lines are closely followed but with different peaks.

The qq-plot shows that the ordered distribution of residuals (blue dots) follows the linear trend of the samples taken from a standard normal distribution with N(0, 1) with slight deviations at times.

The residuals over time don’t display any obvious seasonality and appear to be white noise. This is confirmed by the autocorrelation (i.e. correlogram) plot which shows that the time series residuals have low correlation with lagged versions of itself.

**5.2.2 ARIMA**

For the parameters description and documentation - <https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arima.auto_arima.html>

!pip install pmdarima

from pmdarima import auto\_arima

df= nifty.copy()

df.dropna(inplace=True)

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

df\_train = df[df.index < "2019"]

df\_valid = df[df.index >= "2019"]

model = auto\_arima(df\_train.Close, trace=True, error\_action="ignore", suppress\_warnings=True)

model.fit(df\_train.Close)

forecast = model.predict(n\_periods=len(df\_valid))

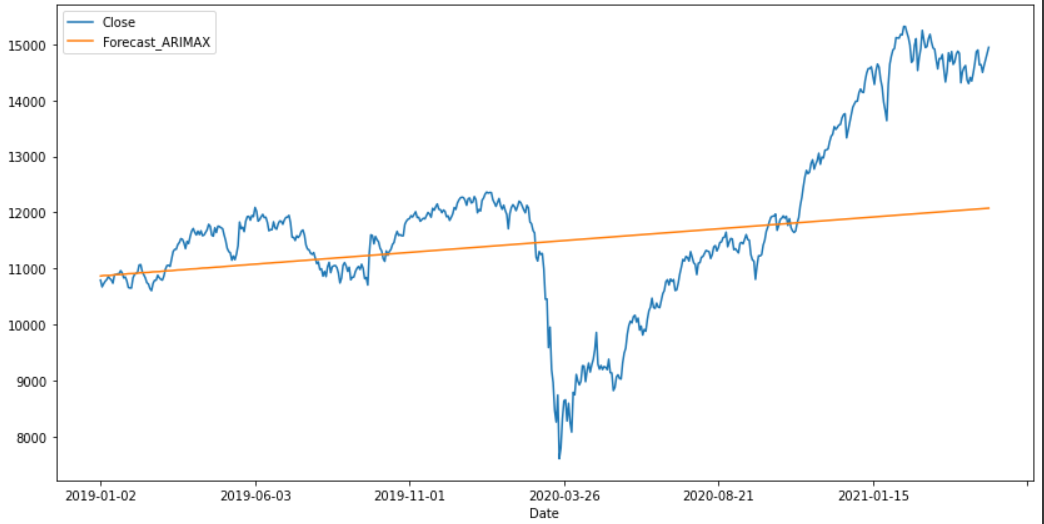
df\_valid["Forecast\_ARIMAX"] = forecast

Data is splitted into train and valid sets. Upto 2019 it is taken as train data and after that the data is taken as test data. After that train data is fit to the auto-arima model and predicted for valid set.

The forecast values are stored in the Forecast\_ARIMAX feature.

df\_train.plot(figsize=(14, 7))

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



# Plot forecasting Close price

forecast\_df = pd.DataFrame(forecast,index = df\_valid.index,columns=['Prediction'])

plt.plot(df\_train['Close'],label='Close Training Data')

plt.plot(df\_valid['Close'],label='Close Test Data')

plt.plot(forecast\_df['Prediction'],label='Close Forecasting Data')

plt.legend()



**5.2.3 ARIMAX**

The code is basically the same as ARIMA but we add exogenous features and fit the model.

df\_train = df[df.index < "2019"]

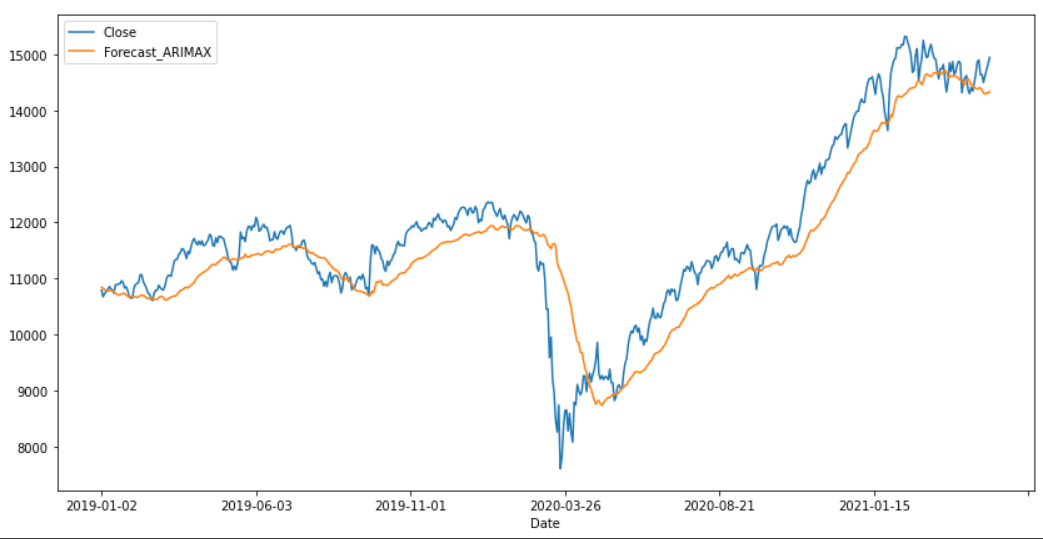
df\_valid = df[df.index >= "2019"]

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

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

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

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



**5.2.4 Facebook Prophet**

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.

To start with prophet <https://facebook.github.io/prophet/docs/quick_start.html#python-api> gives details on how to use it.

!pip install prophet

#Renaming columns

df = nifty\_train[["Date", "Close"] + exogenous\_features].rename(columns={"Date": "ds", "Close": "y"})

# Importing prophet

from prophet import Prophet

# fitting model

m = Prophet()

m.fit(df)

# Making data frame for future dates

future = m.make\_future\_dataframe(periods=10)

# predicting for future dates

forecast = m.predict(future)

x = forecast[['ds', 'yhat', 'yhat\_lower', 'yhat\_upper']]

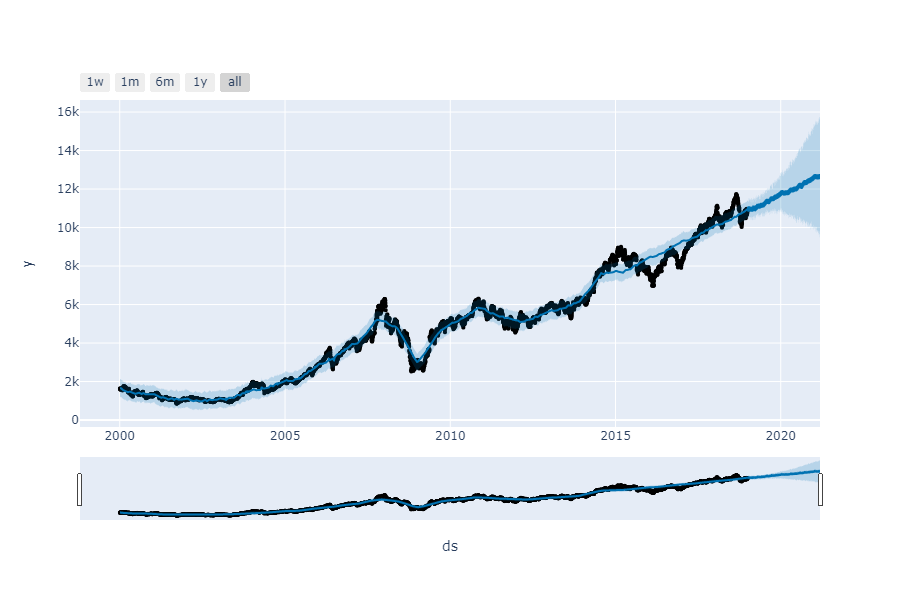
# Storing values of Forecast prophet

x.drop(['yhat\_lower','yhat\_upper'],axis =1,inplace = False)

df = df.append(x)

df = df.rename(columns = {'yhat':'Forecast\_Prophet'} )

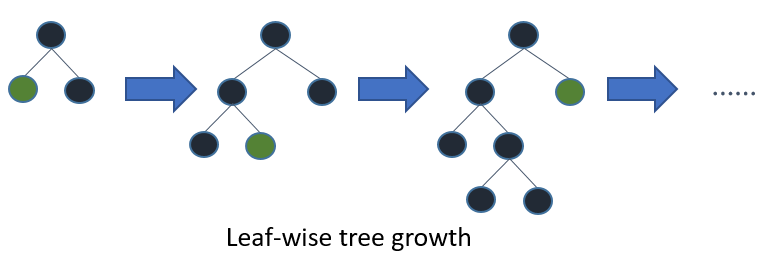
plot\_plotly(m, forecast)



5.3 Machine Learning Approach :

**5.3.1 Light GBM**

LightGBM (Light Gradient Boosting Machine) is a free and open source distributed gradient boosting platform for machine learning that was created by Microsoft. It is used for ranking, classification, and other machine learning tasks and is based on decision tree algorithms. Quality and scalability are at the forefront of the production process.



Trees are grown leaf-by-leaf by LightGBM (best-first). It will expand the leaf with the greatest delta loss. Leaf-wise algorithms have a lower loss than level-wise algorithms when leaf is fixed.

When data is small, leaf-wise can trigger over-fitting, so LightGBM includes the max depth parameter to restrict tree depth. Even when max depth is defined, trees continue to grow leaf-wise.

Resource : <https://lightgbm.readthedocs.io/en/latest/Quick-Start.html>

One can get complete documentation in the above link.

Steps performed :

1. Splitting data into train (16 years), valid (3 years), test (2 years) data sets.
2. Fining dependent and independent variables
3. Defining data
4. Find optimal set of parameters
5. Update model parameters
6. Training model on optimal parameters
7. Evaluating the performances

lgb\_train = lgb.Dataset(X\_train, y\_train)

lgb\_eval = lgb.Dataset(X\_val, y\_val, reference=lgb\_train)

lgb\_model = lgb.train(opt\_params, lgb\_train, valid\_sets=lgb\_eval)

**5.3.2 XG-Boost**

XGBoost is a distributed gradient boosting library that has been optimised for performance, flexibility, and portability. It uses the Gradient Boosting paradigm to implement machine learning algorithms. XGBoost is a parallel tree boosting (also known as GBDT, GBM) algorithm that solves a variety of data science problems quickly and accurately.

<https://xgboost.readthedocs.io/en/latest/python/index.html>

Extreme Gradient Boosting (XGBoost) is just an extension of gradient boosting with the following added advantages:

1. Regularization: Standard GBM implementation has no regularization like XGBoost, therefore it also helps to reduce overfitting. In fact, XGBoost is also known as ‘regularized boosting‘ technique.
2. Parallel Processing: XGBoost implements parallel processing and is blazingly faster as compared to GBM. But hang on, we know that boosting is a sequential process so how can it be parallelized? We know that each tree can be built only after the previous one, but to make a tree it uses all the cores of the system. XGBoost also supports implementation on Hadoop.
3. High Flexibility: XGBoost allows users to define custom optimization objectives and evaluation criteria. This adds a whole new dimension to the model and there is no limit to what we can do.
4. Handling Missing Values: XGBoost has an in-built routine to handle missing values. User is required to supply a different value than other observations and pass that as a parameter. XGBoost tries different things as it encounters a missing value on each node and learns which path to take for missing values in future.
5. Tree Pruning: A GBM would stop splitting a node when it encounters a negative loss in the split. Thus it is more of a greedy algorithm. XGBoost on the other hand makes splits up to the max\_depth specified and then starts pruning the tree backwards and removes splits beyond which there is no positive gain. Another advantage is that sometimes a split of negative loss say -2 may be followed by a split of positive loss +10. GBM would stop as it encounters -2. But XGBoost will go deeper and it will see a combined effect of +8 of the split and keep both.
6. Built-in Cross-Validation: XGBoost allows users to run a cross-validation at each iteration of the boosting process and thus it is easy to get the exact optimum number of boosting iterations in a single run. This is unlike GBM where we have to run a grid-search and only a limited value can be tested.
7. Continue on Existing Model: Users can start training an XGBoost model from its last iteration of previous run. This can be of significant advantage in certain specific applications. GBM implementation of sklearn also has this feature so they are even on this point.

import xgboost as xgb

tscv = TimeSeriesSplit(5)

xgb\_model = xgb.XGBClassifier(gamma=20)

parameters = {'objective' :['binary:logistic'],

'learning\_rate' : [0.1,0.3],

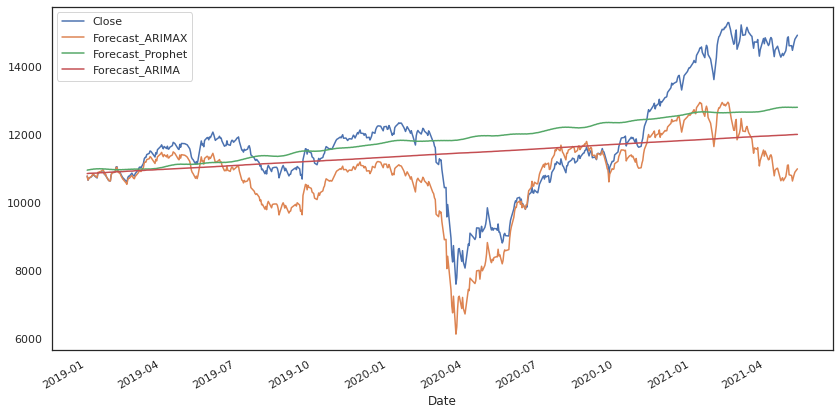
'max\_depth' : [3,6]}

xgb\_fit = GridSearchCV(xgb\_model,parameters,n\_jobs=-1,cv=tscv,scoring='neg\_log\_loss',verbose=20,refit=True)

xgb\_fit.fit(X\_train,y\_train)

**CHAPTER 6 - COMPARISON AND EVALUATION**

Comparison of time-series models on validation set



Forecast\_ARIMA - It is a straight line forecast. It doesn't care about residuals but following trend.

Forecast\_ARIMAX - It is a good forecast model. But the values are a little far from original. This model takes care about trend and residuals also.

Forecast\_Prophet - This model gives an interesting forecast if the model is not having sudden peaks.

Evaluating LGB and XGB

# Convert the probs to classes

train\_preds\_lgb = np.where(train\_preds\_lgb > 0.5,1,0)

val\_preds\_lgb = np.where(val\_preds\_lgb > 0.5,1,0)

test\_preds\_lgb = np.where(test\_preds\_lgb > 0.5,1,0)

print(accuracy\_score(y\_train,train\_preds\_lgb))

print(accuracy\_score(y\_val,val\_preds\_lgb))

print(accuracy\_score(y\_test,test\_preds\_lgb))

# Confusion Matrix

conf\_matrix\_train = confusion\_matrix(y\_train,train\_preds\_lgb)

conf\_matrix\_val = confusion\_matrix(y\_val,val\_preds\_lgb)

conf\_matrix\_test = confusion\_matrix(y\_test,test\_preds\_lgb)

print("The Confusion Matrix for Train Set \n",conf\_matrix\_train)

print("\n")

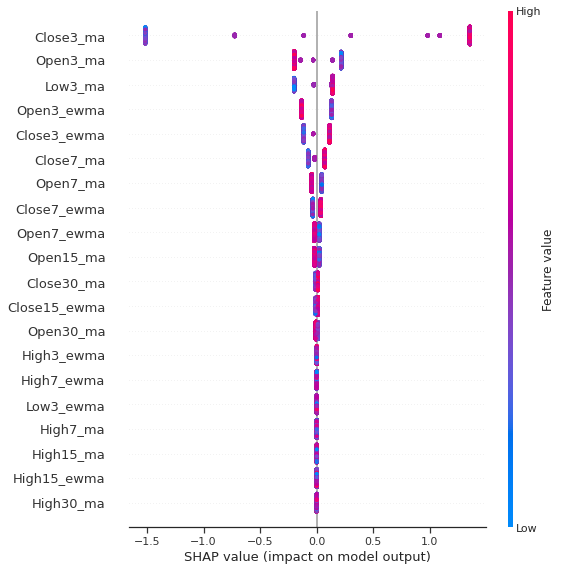
print("The Confusion Matrix for Validation Set \n",conf\_matrix\_val)

print("\n")

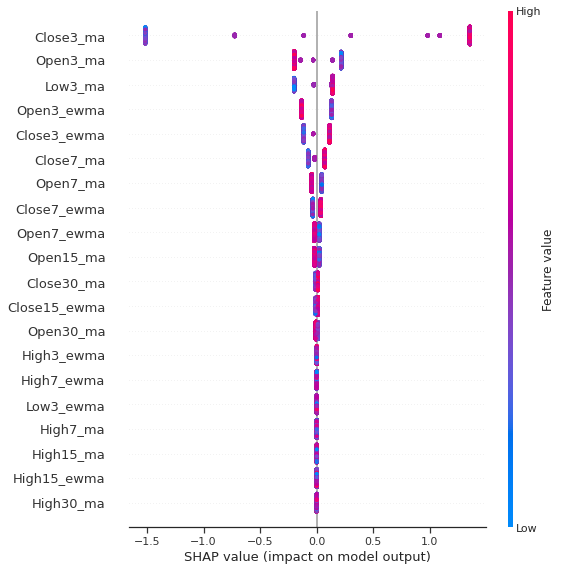
print("The Confusion Matrix for Test Set \n",conf\_matrix\_test)

Feature Importance using Shap

On Training Set



On Validation set



CONCLUSION :

* While using regression models, residuals are quite high.
* Difficult to predict numerical target value.
* While using classification models, we have a higher chance of predicting the outcome.

Improvement to the model prediction:

1. Forecast for more than 1 day for unseen data.
2. More features can be included to increase model efficiency.