Stock Predictions: A cognizant way of investments

The Best Way to Predict Your Future Is to Create It

Advising Professor Prof. Nik Brown

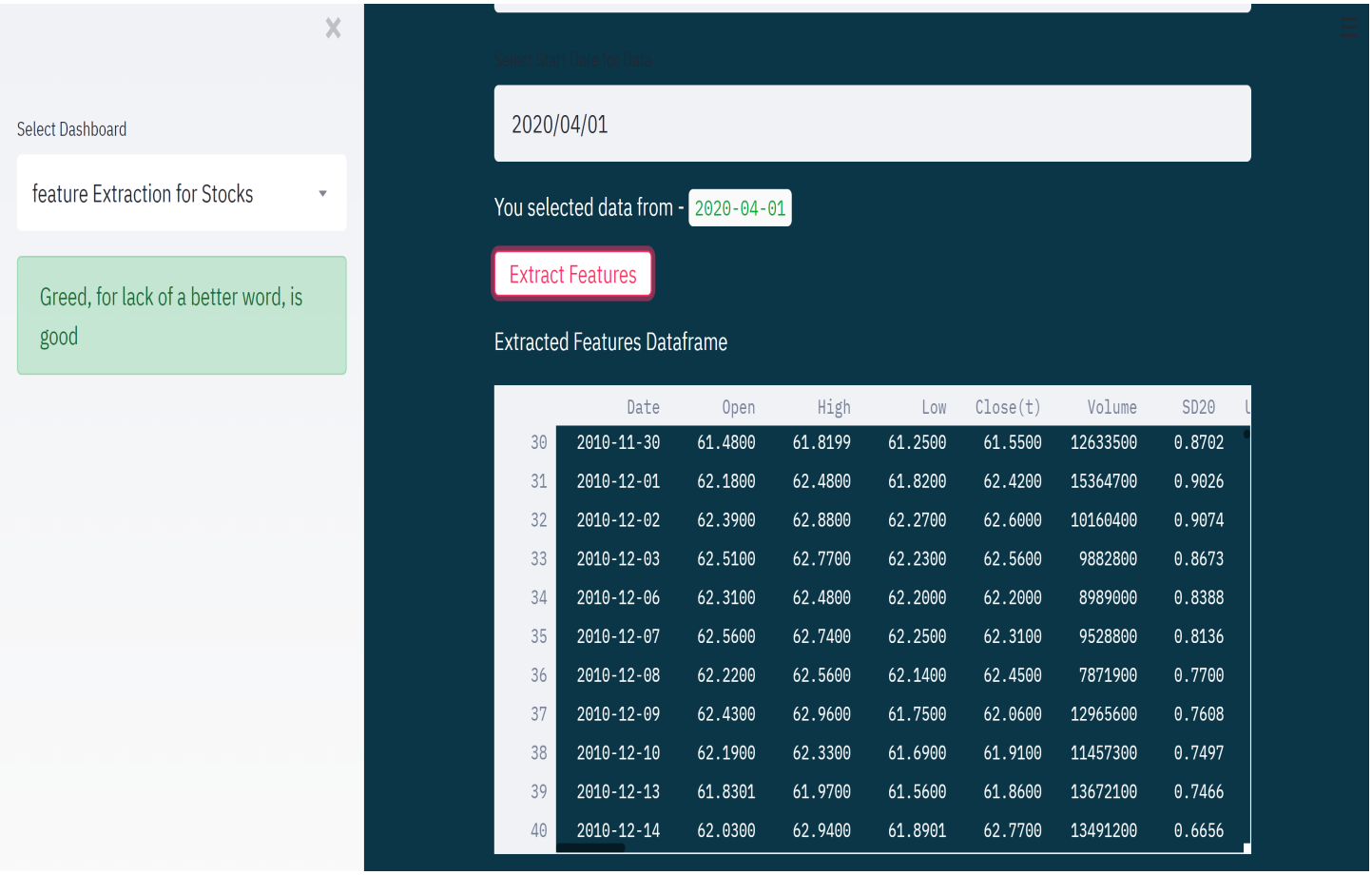
Avinash Chourasiya 001427579

Nikhil Kohli 001873199

END RESULT

We have deployed our model in Heroku and giving a user an extensive feature extraction and model training capability: https://stock-prediction-dashboard.herokuapp.com/

The blogpost to follow for a summary of our work can be found at: https://medium.com/@nikhilkohli1992/extracting-features-for-stock-prediction-streamlit-based-application-a97afc55d926



	Date	Open	High	Low	Close(t)	Volume	SD20
30	2010-11-30	61.4800	61.8199	61.2500	61.5500	12633500	0.8702
31	2010-12-01	62.1800	62.4800	61.8200	62.4200	15364700	0.9026
32	2010-12-02	62.3900	62.8800	62.2700	62.6000	10160400	0.9074
33	2010-12-03	62.5100	62.7700	62.2300	62.5600	9882800	0.8673
34	2010-12-06	62.3100	62.4800	62.2000	62.2000	8989000	0.8388
35	2010-12-07	62.5600	62.7400	62.2500	62.3100	9528800	0.8136
36	2010-12-08	62.2200	62.5600	62.1400	62.4500	7871900	0.7700
37	2010-12-09	62.4300	62.9600	61.7500	62.0600	12965600	0.7608
38	2010-12-10	62.1900	62.3300	61.6900	61.9100	11457300	0.7497
39	2010-12-13	61.8301	61.9700	61.5600	61.8600	13672100	0.7466
40	2010-12-14	62.0300	62.9400	61.8901	62.7700	13491200	0.6656

Please follow this book to gain an in-depth knowledge of the work we did.

TABLE OF CONTENTS

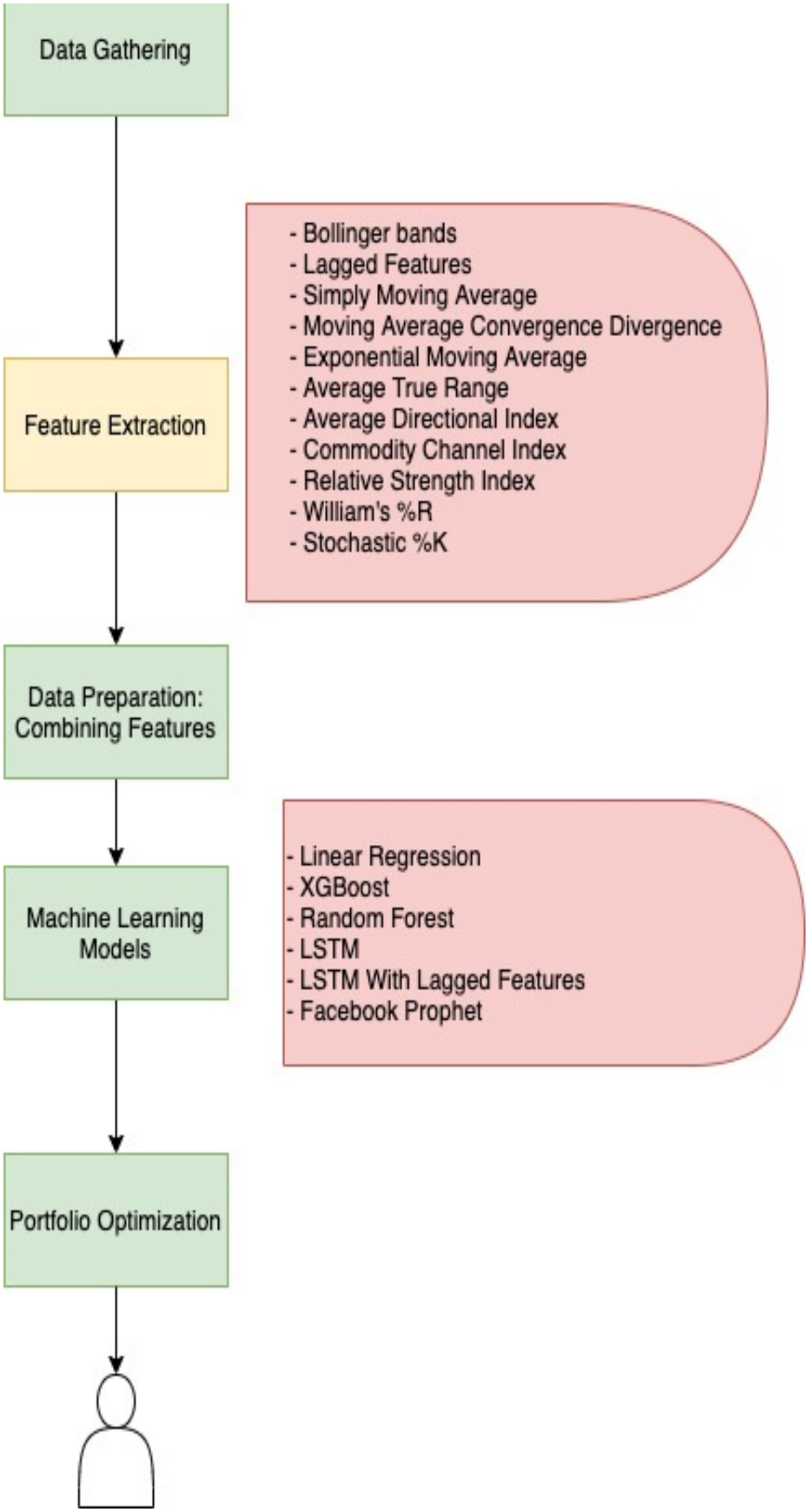
ABSTRACT

The ubiquity of data today enables investors at any scale to make better investment decisions. The challenge is ingesting and interpreting the data to determine which data is useful, finding the signal in this sea of information.

Stock market prediction is the act of trying to determine the future value of a company stock or other financial instrument traded on an exchange. The successful prediction of a stock's future price could yield significant profit. The efficient-market hypothesis suggests that stock prices reflect all currently available information and any price changes that are not based on newly revealed information thus are inherently unpredictable. Others disagree and those with this viewpoint possess myriad methods and technologies which purportedly allow them to gain future price information.

We fall in the second categories and in this project, we will be using the power of Machine Learning and Artificial Intelligence to predict the stock prices.

Flow Diagram



INTRODUCTION

Forecasting Future Stock prices is a very hard problem to solve. An efficient Predictive model to correctly forecast future trend is crucial for Hedge funds and algorithmic trading. Especially in the case of Algorithmic Trading where error should me minimal as millions of dollars are at stake for each trade.

Portfolio Optimization strategies needs to be back tested on historical data after predicting future stock prices.

Stock prices depends upon many factors like the Market behaviors, other stocks, Index funds, Global news etc. We will try to capture many of these in our features.

In this project, we will look at this problem in many ways to Predict the Closing Prices -

- We will start with Extracting Features and see which performs well for predicting each stock. We will extract various Technical Indicators described below.
- Then check correlation and Perform feature selection using RFECV Recursive feature Elimination using Random Forest to select best features.
- Then we will create a pipeline for this feature extraction and convert the entire code into Pipeline so anyone can easily run it and get the extracted features data for each stock.
- Next we will use Time Lagged data as a feature and create features based on previous day closing prices, Previous days Index funds prices.
- Then we will train 4 different Algorithms - Linear Regression, Random Forest, XG Boost, LSTM and GRU for forecasting next day price and test and evaluate it on historical stock data.
- We will also create a **Pipeline** for this to train many stocks with many algorithms in just one go.
- We will Evaluate the data on around 2 years of data which is a long period, so if our models are closer overall, means we are doing great. Metrics we will use are MAE, MAPE, R2 and RMSE. Final Metrics which we will look at to compare models is MAE (Mean Absolute Error)
- We will also check feature importance of various features using Random forest and XG boost in this.
- We will pick the best algorithm from these and will tune the Number of lagged days to consider for forecasting for each type like Stock price, other index Funds previous prices.
- For LSTM, we will use Lagged previous days prices for a lookback period of 30-60 days.
- Then we will create a Portfolio of these stocks and will build a strategy using Sharpe ratio to optimize the portfolio and allocate the money of a fund effectively.
- As a future scope, we will also try to create a dashboard to Show the comparison of 2 portfolios before and after optimization.

Data Loading:

We will be using Alpha vantage API to extract the stocks prices for previous 15 years

What is Alpha Vantage?

Alpha Vantage (Composed of a tight-knit community of researchers, engineers, and business professionals, Alpha Vantage Inc. is a leading provider of free APIs for real-time and historical data on stocks, physical currencies, and digital/crypto currencies.

Getting data from alpha vantage is really easy. We just need an API key which is free of cost

[1]:
```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
from alpha_vantage.timeseries import TimeSeries

%matplotlib inline
```

[3]:
```
os.chdir(r'N:\STOCK ADVISOR BOT')
```

[4]:
```
ALPHA_VANTAGE_API_KEY = 'XAGC5LBB1SI9RDLW'
ts = TimeSeries(key= ALPHA_VANTAGE_API_KEY, output_format='pandas')
```

[5]:
```
df_NFLX, NFLX_info = ts.get_daily('NFLX', outputsize='full')
```

[6]:
```
df_NFLX
```

[6]:

date	1. open	2. high	3. low	4. close	5. volume
2020-04-09	371.0600	372.10	363.03	370.72	7415166.0
2020-04-08	374.0100	378.39	368.31	371.12	6908879.0
2020-04-07	380.0000	381.33	369.34	372.28	7046438.0
2020-04-06	365.2200	380.29	361.71	379.96	8183921.0
2020-04-03	367.3604	370.90	357.51	361.76	4860768.0
...	...	...	...	...	...
2002-05-30	15.5100	15.51	15.00	15.00	725300.0
2002-05-29	16.3000	16.30	15.20	15.45	482700.0
2002-05-28	16.9900	17.25	16.20	16.20	472100.0
2002-05-24	17.0000	17.15	16.76	16.94	793200.0
2002-05-23	16.1900	17.40	16.04	16.75	7485000.0

4502 rows × 5 columns

Check one of the loaded data:

[7]:
```
NFLX_info
```

[7]:
```
{'1. Information': 'Daily Prices (open, high, low, close) and Volumes',
 '2. Symbol': 'NFLX',
 '3. Last Refreshed': '2020-04-09',
 '4. Output Size': 'Full size',
 '5. Time Zone': 'US/Eastern'}
```

[8]:
```
df_NFLX = df_NFLX.rename(columns={'1. open' : 'Open', '2. high': 'High', '3. low':'Low', '4. close': 'Close', '5. volume': 'Volume' })
df_NFLX = df_NFLX.rename_axis(['Date'])
df_NFLX
```

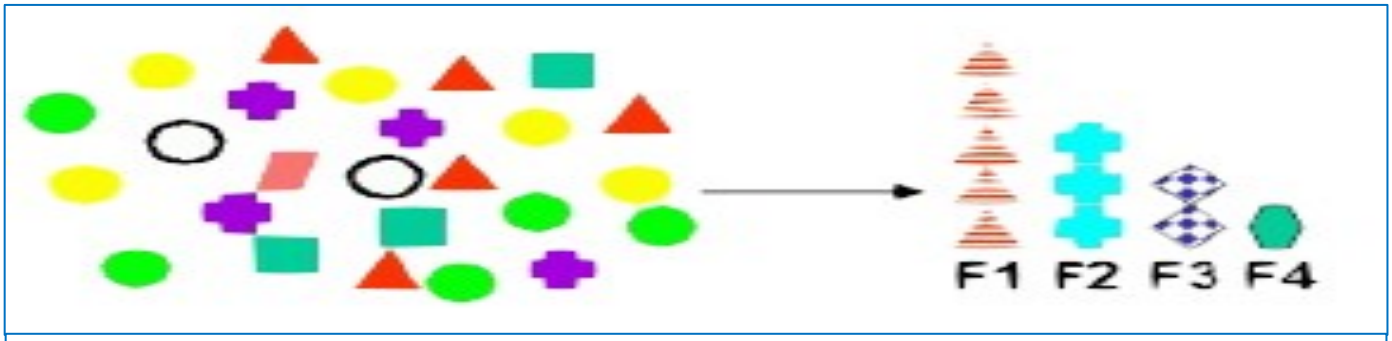
[8]:

Date	Open	High	Low	Close	Volume
2020-04-09	371.0600	372.10	363.03	370.72	7415166.0
2020-04-08	374.0100	378.39	368.31	371.12	6908879.0
2020-04-07	380.0000	381.33	369.34	372.28	7046438.0
2020-04-06	365.2200	380.29	361.71	379.96	8183921.0
2020-04-03	367.3604	370.90	357.51	361.76	4860768.0
...	...	...	...	...	...
2002-05-30	15.5100	15.51	15.00	15.00	725300.0
2002-05-29	16.3000	16.30	15.20	15.45	482700.0
2002-05-28	16.9900	17.25	16.20	16.20	472100.0
2002-05-24	17.0000	17.15	16.76	16.94	793200.0
2002-05-23	16.1900	17.40	16.04	16.75	7485000.0

4502 rows × 5 columns

```
[10]: NFLX['Close'].plot(figsize=(10, 7))
plt.title("Netflix Stock Price", fontsize=17)
plt.ylabel('Price', fontsize=14)
plt.xlabel('Time', fontsize=14)
plt.grid(which="major", color='k', linestyle='-.', linewidth=0.5)
plt.show()
```

Feature Extraction



"When the input data to an algorithm is too large to be processed and it is suspected to be redundant (e.g. the same measurement in both feet and meters, or the repetitiveness of images presented as pixels), then it can be transformed into a reduced set of features (also named a feature vector). Determining a subset of the initial features is called feature selection. The selected features are expected to contain the relevant information from the input data, so that the desired task can be performed by using this reduced representation instead of the complete initial data". - Wikipedia

We will create features using various Technical indicators and Lagged prices as described below. All of these features have something to offer for forecasting. Some tells us about the trend, some gives us a signal if the stock is overbought or oversold, some portrays the strength of the price trend.

Feature Extraction for Predictinig Stock prices

1. Using Index Fund Nasdaq-100 ETF QQQ's Previous Day & Moving Average price as a feature

[11]:
```
QQQ, QQQ_info = ts.get_daily('QQQ', outputsize='full')
QQQ = QQQ.rename(columns={'1. open' : 'Open', '2. high': 'High', '3. low':'Low', '4. close': 'QQQ_Close', '5. volume': 'Volume' })
QQQ = QQQ.rename_axis(['Date'])
QQQ = QQQ.drop(columns=['Open', 'High', 'Low', 'Volume'])
```

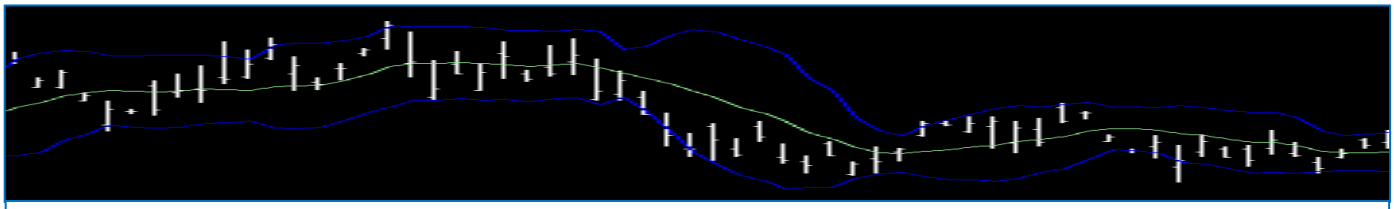
[12]:
```
#sorting index
QQQ = QQQ.sort_index(ascending=True, axis=0)
#slicing the data for 15 years from '2004-01-02' to today
QQQ = QQQ.loc['2004-01-02':]
QQQ
```

[12]:

Date	QQQ_Close
2004-01-02	36.36
2004-01-05	37.09
2004-01-06	37.34
2004-01-07	37.68
2004-01-08	37.98
...	...
2020-04-03	183.37
2020-04-06	196.48
2020-04-07	196.40
2020-04-08	200.57
2020-04-09	200.86

4096 rows × 1 columns

Bollinger Bands



A Bollinger Band® is a technical analysis tool defined by a set of lines plotted two standard deviations (positively and negatively) away from a simple moving average (SMA) of the Stocks' price. Bollinger Bands allow traders to monitor and take advantage of shifts in price volatilities

Main Components of a Bollinger Bands

- Upper Band: The upper band is simply two standard deviations above the moving average of a stock's price.
- Middle Band: The middle band is simply the moving average of the stock's price.
- Lower Band: Two standard deviations below the moving average is the lower band.

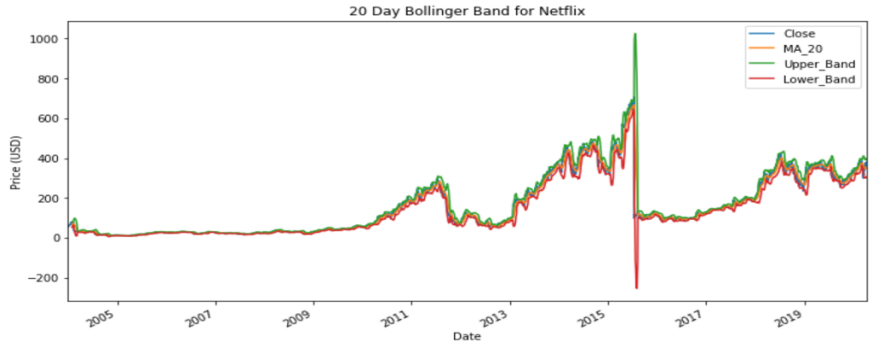
```
[15]: NFLX['MA_20'] = NFLX.Close.rolling(window=20).mean()
      NFLX['SD20'] = NFLX.Close.rolling(window=20).std()
      NFLX['Upper_Band'] = NFLX.Close.rolling(window=20).mean() + (NFLX['SD20']*2)
      NFLX['Lower_Band'] = NFLX.Close.rolling(window=20).mean() - (NFLX['SD20']*2)
      NFLX.tail()
```

[15]:

Date	Open	High	Low	Close	Volume	MA_20	SD20	Upper_Band	Lower_Band
2020-04-03	367.3604	370.90	357.51	361.76	4860768.0	346.67375	21.757366	390.188483	303.159017
2020-04-06	365.2200	380.29	361.71	379.96	8183921.0	348.34725	22.994514	394.336277	302.358223
2020-04-07	380.0000	381.33	369.34	372.28	7046438.0	348.75475	23.358269	395.471288	302.038212
2020-04-08	374.0100	378.39	368.31	371.12	6908879.0	349.81475	23.888932	397.592614	302.036886
2020-04-09	371.0600	372.10	363.03	370.72	7415166.0	352.58825	22.862750	398.313751	306.862749

Check the prepared features:

```
[16]: NFLX[['Close', 'MA_20', 'Upper_Band', 'Lower_Band']].plot(figsize=(12,6))
      plt.title('20 Day Bollinger Band for Netflix')
      plt.ylabel('Price (USD)')
      plt.show();
```



Lagged Features:

Lag is essentially delay. Just as correlation shows how much two timeseries are similar, autocorrelation describes how similar the time series is with itself.

Consider a discrete sequence of values, for lag 1, you compare your time series with a lagged time series, in other words you shift the time series by 1 before comparing it with itself. Proceed doing this for the entire length of time series by shifting it by 1 every time. You now have autocorrelation function.

Shifting for Lagged data - Adding Previous Day prices

[17]:
```
NFLX['NFLX_Close(t-1)'] = NFLX.Close.shift(periods=1)
NFLX['NFLX_Close(t-2)'] = NFLX.Close.shift(periods=2)
NFLX['NFLX_Close(t-5)'] = NFLX.Close.shift(periods=5)
NFLX['NFLX_Close(t-10)'] = NFLX.Close.shift(periods=10)
NFLX['NFLX_Open(t-1)'] = NFLX.Open.shift(periods=1)
```

[18]:
```
NFLX.head(20)
```

[18]:

Date	Open	High	Low	Close	Volume	MA_20	SD20	Upper_Band	Lower_Band	NFLX_Close(t-1)	NFLX_Close(t-2)	NFLX_Close(t-5)	NFLX_Close(t-10)	NFLX_Open(t-1)
2004-01-02	57.500	57.79	53.790	54.830	3587900.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2004-01-05	54.500	56.86	54.250	55.900	1800500.0	NaN	NaN	NaN	NaN	54.830	NaN	NaN	NaN	57.500
2004-01-06	55.250	60.75	55.120	59.610	3759500.0	NaN	NaN	NaN	NaN	55.900	54.830	NaN	NaN	54.500
2004-01-07	60.300	62.58	59.490	62.240	4888000.0	NaN	NaN	NaN	NaN	59.610	55.900	NaN	NaN	55.250
2004-01-08	63.550	63.90	60.079	62.060	2655600.0	NaN	NaN	NaN	NaN	62.240	59.610	NaN	NaN	60.300
2004-01-09	60.720	64.10	60.420	62.000	2359300.0	NaN	NaN	NaN	NaN	62.060	62.240	54.83	NaN	63.550
2004-01-12	62.240	64.60	61.130	64.050	1918400.0	NaN	NaN	NaN	NaN	62.000	62.060	55.90	NaN	60.720
2004-01-13	64.380	65.52	63.380	65.310	2354500.0	NaN	NaN	NaN	NaN	64.050	62.000	59.61	NaN	62.240
2004-01-14	64.966	66.15	64.200	65.100	1647900.0	NaN	NaN	NaN	NaN	65.310	64.050	62.24	NaN	64.380

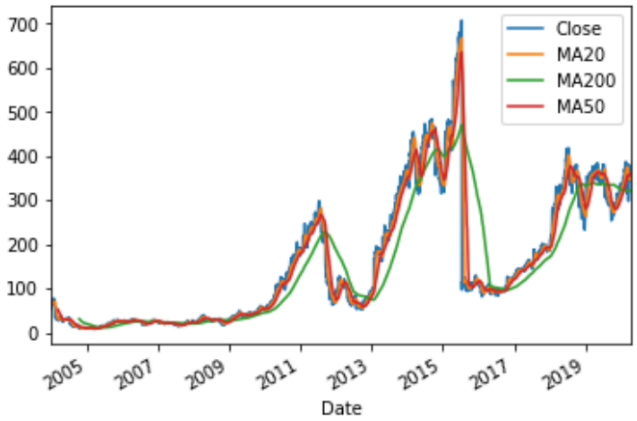
Simple Moving Average (SMA)

A simple moving average (SMA) calculates the average of a selected range of prices, usually closing prices, by the number of periods in that range. SMA is basically the average price of the given time period, with equal weighting given to the price of each period.

Formula: **SMA = (Sum (Price, n)) / n**

[19]:
```
NFLX['MA5'] = NFLX.Close.rolling(window=5).mean()
NFLX['MA10'] = NFLX.Close.rolling(window=10).mean()
NFLX['MA20'] = NFLX.Close.rolling(window=20).mean()
NFLX['MA50'] = NFLX.Close.rolling(window=50).mean()
NFLX['MA200'] = NFLX.Close.rolling(window=200).mean()
```

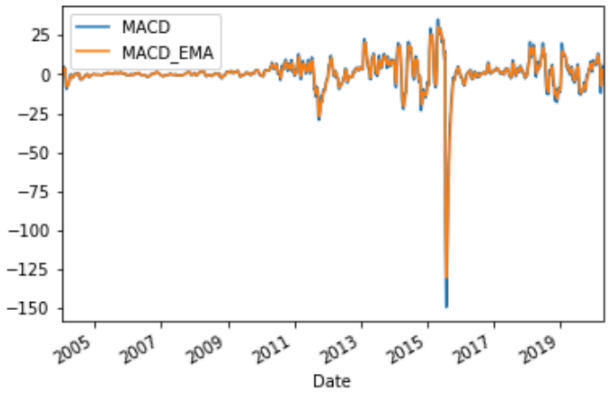
[20]:
```
NFLX[['Close', 'MA20', 'MA200', 'MA50']].plot()
plt.show()
```



Moving Average Convergance Divergance

[21]:
```
NFLX['EMA_12'] = NFLX.Close.ewm(span=12, adjust=False).mean()
NFLX['EMA_26'] = NFLX.Close.ewm(span=26, adjust=False).mean()
NFLX['MACD'] = NFLX['EMA_12'] - NFLX['EMA_26']

NFLX['MACD_EMA'] = NFLX.MACD.ewm(span=9, adjust=False).mean()
```

[22]:
```
NFLX[['MACD', 'MACD_EMA']].plot()
plt.show()
```



Exponential moving average (EMA)

An exponential moving average (EMA) is a type of moving average (MA) that places a greater weight and significance on the most recent data points. The exponential moving average is also referred to as the exponentially weighted moving average. An exponentially weighted moving average reacts more significantly to recent price changes than a simple moving average (SMA), which applies an equal weight to all observations in the period.

[23]:
```
NFLX['EMA10'] = NFLX.Close.ewm(span=5, adjust=False).mean().fillna(0)
NFLX['EMA20'] = NFLX.Close.ewm(span=5, adjust=False).mean().fillna(0)
NFLX['EMA50'] = NFLX.Close.ewm(span=5, adjust=False).mean().fillna(0)
NFLX['EMA100'] = NFLX.Close.ewm(span=5, adjust=False).mean().fillna(0)
NFLX['EMA200'] = NFLX.Close.ewm(span=5, adjust=False).mean().fillna(0)
```

Average true range (ATR)

The average true range (ATR) is a technical analysis indicator that measures market volatility by decomposing the entire range of an asset price for that period. ATR measures market volatility. It is typically derived from the 14-day moving average of a series of true range indicators.

[24]:
```
import talib
```

[25]:
```
NFLX['ATR'] = talib.ATR(NFLX['High'].values, NFLX['Low'].values, NFLX['Close'].values, timeperiod=14)
```

Average Directional Index (ADX)

ADX stands for Average Directional Movement Index and can be used to help measure the overall strength of a trend. The ADX indicator is an average of expanding price range values. ADX indicates the strength of a trend in price time series. It is a combination of the negative and positive directional movements indicators computed over a period of n past days corresponding to the input window length (typically 14 days)

[26]:
```
NFLX['ADX'] = talib.ADX(NFLX['High'], NFLX['Low'], NFLX['Close'], timeperiod=14)
```

Commodity Channel Index (CCI)

Commodity Channel Index (CCI) is a momentum-based oscillator used to help determine when an investment vehicle is reaching a condition of being overbought or oversold. It is also used to assess price trend direction and strength.

CCI = (typical price − ma) / (0.015 * mean deviation) typical price = (high + low + close) / 3 p = number of periods (20 commonly used) ma = moving average moving average = typical price / p mean deviation = (typical price — MA) / p

[27]:
```
tp = (NFLX['High'] + NFLX['Low'] + NFLX['Close']) /3
ma = tp/20
md = (tp-ma)/20
NFLX['CCI'] = (tp-ma)/(0.015 * md)
```

Rate-of-change (ROC)

ROC measures the percentage change in price between the current price and the price a certain number of periods ago.

[28]:
```
NFLX['ROC'] = ((NFLX['Close'] - NFLX['Close'].shift(10)) / (NFLX['Close'].shift(10)))*100
```

Relative Strength Index (RSI)

RSI compares the size of recent gains to recent losses; it is intended to reveal the strength or weakness of a price trend from a range of closing prices over a time period.

[29]:
```
NFLX['RSI'] = talib.RSI(NFLX.Close.values, timeperiod=14)
```

William's %R

Williams %R, also known as the Williams Percent Range, is a type of momentum indicator that moves between 0 and -100 and measures overbought and oversold levels. The Williams %R may be used to find entry and exit points in the market.

[30]:
```
NFLX['William%R'] = talib.WILLR(NFLX.High.values, NFLX.Low.values, NFLX.Close.values, 14)
```

Stochastic %K

A stochastic oscillator is a momentum indicator comparing a particular closing price of a security to a range of its prices over a certain period of time. It compares a close price and its price interval during a period of n past days and gives a signal meaning that a stock is oversold or overbought.

[31]:
```
NFLX['SO%K'] = ((NFLX.Close - NFLX.Low.rolling(window=14).min()) / (NFLX.High.rolling(window=14).max() - NFLX.Low.rolling(window=14).min())) * 10
```

3. **Let's also take data from S&P 500**

3. Using S&P 500 Index

[34]:
```
SnP, SnP_info = ts.get_daily('INX', outputsize='full')
SnP = SnP.rename(columns={'1. open' : 'Open', '2. high': 'High', '3. low':'Low', '4. close': 'SnP_Close', '5. volume': 'Volume' })
SnP = SnP.rename_axis(['Date'])
SnP = SnP.drop(columns=['Open', 'High', 'Low', 'Volume'])
```

[35]:
```
#sorting index
SnP = SnP.sort_index(ascending=True, axis=0)
#slicing the data for 15 years from '2004-01-02' to today
SnP = SnP.loc['2004-01-02':]
SnP
```

[35]:

Date	SnP_Close
2004-01-02	1108.4800
2004-01-05	1122.2200
2004-01-06	1123.6700
2004-01-07	1126.3300
2004-01-08	1131.9200
...	...

Data Preparation:

As of now we have created features using various methods and sources. It's time that we merge everything into a single data-frame and use it in our prediction

Merging of all columns

[37]:
```
NFLX = NFLX.merge(QQQ, left_index=True, right_index=True)
NFLX
```

[37]:

Date	Open	High	Low	Close	Volume	MA_20	SD20	Upper_Band	Lower_Band	NFLX_Close(t-1)	...	SO%K	per_change	STD5	QQQ_Close	QQQ(t-1)
2004-01-02	57.5000	57.79	53.790	54.83	3587900.0	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	36.36	NaN
2004-01-05	54.5000	56.86	54.250	55.90	1800500.0	NaN	NaN	NaN	NaN	54.83	...	NaN	0.019515	NaN	37.09	36.36
2004																

[38]:
```
NFLX = NFLX.merge(SnP, left_index=True, right_index=True)
NFLX
```

[38]:

Date	Open	High	Low	Close	Volume	MA_20	SD20	Upper_Band	Lower_Band	NFLX_Close(t-1)	...	QQQ_Close	QQQ(t-1)	QQQ(t-2)	QQQ(t-5)	QQQ_MA10	QQQ
2004-01-02	57.5000	57.79	53.790	54.83	3587900.0	NaN	NaN	NaN	NaN	NaN	...	36.36	NaN	NaN	NaN	NaN	
2004-01-05	54.5000	56.86	54.250	55.90	1800500.0	NaN	NaN	NaN	NaN	54.83	...	37.09	36.36	NaN	NaN	NaN	
2004-01-06	55.2500	60.75	55.120	59.61	3759500.0	NaN	NaN	NaN	NaN	55.90	...	37.34	37.09	36.36	NaN	NaN	

We will not be using every feature, hence removing few of the unwanted columns:

[39]:
```
NFLX.columns
```

[39]:
```
Index(['Open', 'High', 'Low', 'Close', 'Volume', 'MA_20', 'SD20', 'Upper_Band',
       'Lower_Band', 'NFLX_Close(t-1)', 'NFLX_Close(t-2)', 'NFLX_Close(t-5)',
       'NFLX_Close(t-10)', 'NFLX_Open(t-1)', 'MA5', 'MA10', 'MA20', 'MA50',
       'MA200', 'EMA_12', 'EMA_26', 'MACD', 'MACD_EMA', 'EMA10', 'EMA20',
       'EMA50', 'EMA100', 'EMA200', 'ATR', 'ADX', 'CCI', 'ROC', 'RSI',
       'William%R', 'SO%K', 'per_change', 'STD5', 'QQQ_Close', 'QQQ(t-1)',
       'QQQ(t-2)', 'QQQ(t-5)', 'QQQ_MA10', 'QQQ_MA20', 'QQQ_MA50', 'SnP_Close',
       'SnP(t-1))', 'SnP(t-5)'],
      dtype='object')
```

[40]:
```
# Remove unwanted columns
NFLX = NFLX.drop(columns=['MA_20', 'per_change', 'EMA_12', 'EMA_26'])
```

Adding the Next day Close Price Column which needs to be predicted using ***Machine Learning Models***

[42]:
```
NFLX['NFLX_Close(t+1)'] = NFLX.Close.shift(-1)
```

[43]:
```
NFLX
```

[43]:

ıd	Lower_Band	NFLX_Close(t-1)	NFLX_Close(t-2)	...	QQQ(t-5)	QQQ_MA10	QQQ_MA20	QQQ_MA50	SnP_Close	SnP(t-1))	SnP(t-5)	ForceIndex1	ForceIndex20	NFLX_Close(t+1)
ıN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	1108.4800	NaN	NaN	NaN	NaN	55.90
ıN	NaN	54.83	NaN	...	NaN	NaN	NaN	NaN	1122.2200	1108.4800	NaN	1.926535e+06	NaN	59.61
ıN	NaN	55.90	54.83	...	NaN	NaN	NaN	NaN	1123.6700	1122.2200	NaN	1.394775e+07	NaN	62.24
ıN	NaN	59.61	55.90	...	NaN	NaN	NaN	NaN	1126.3300	1123.6700	NaN	1.285544e+07	NaN	62.06

Let's also break down dates into much granular pieces and use it as features:

[74]:
```
from datetime import datetime

def extract_date_features(date_val):

    Day = date_val.day
    DayofWeek = date_val.dayofweek
    Dayofyear = date_val.dayofyear
    Week = date_val.week
    Is_month_end = date_val.is_month_end.real
    Is_month_start = date_val.is_month_start.real
    Is_quarter_end = date_val.is_quarter_end.real
    Is_quarter_start = date_val.is_quarter_start.real
    Is_year_end = date_val.is_year_end.real
    Is_year_start = date_val.is_year_start.real
    Is_leap_year = date_val.is_leap_year.real
    Year = date_val.year
    Month = date_val.month

    return Day, DayofWeek, Dayofyear, Week, Is_month_end, Is_month_start, Is_quarter_end, Is_quarter_start, Is_year_end, Is_year_start, Is_leap_y
```

[77]:
```
funct = lambda x: pd.Series(extract_date_features(x))
NFLX[['Day', 'DayofWeek', 'DayofYear', 'Week', 'Is_month_end', 'Is_month_start',
      'Is_quarter_end', 'Is_quarter_start', 'Is_year_end', 'Is_year_start', 'Is_leap_year', 'Year', 'Month']] = NFLX.Date_Col.apply(funct)
```

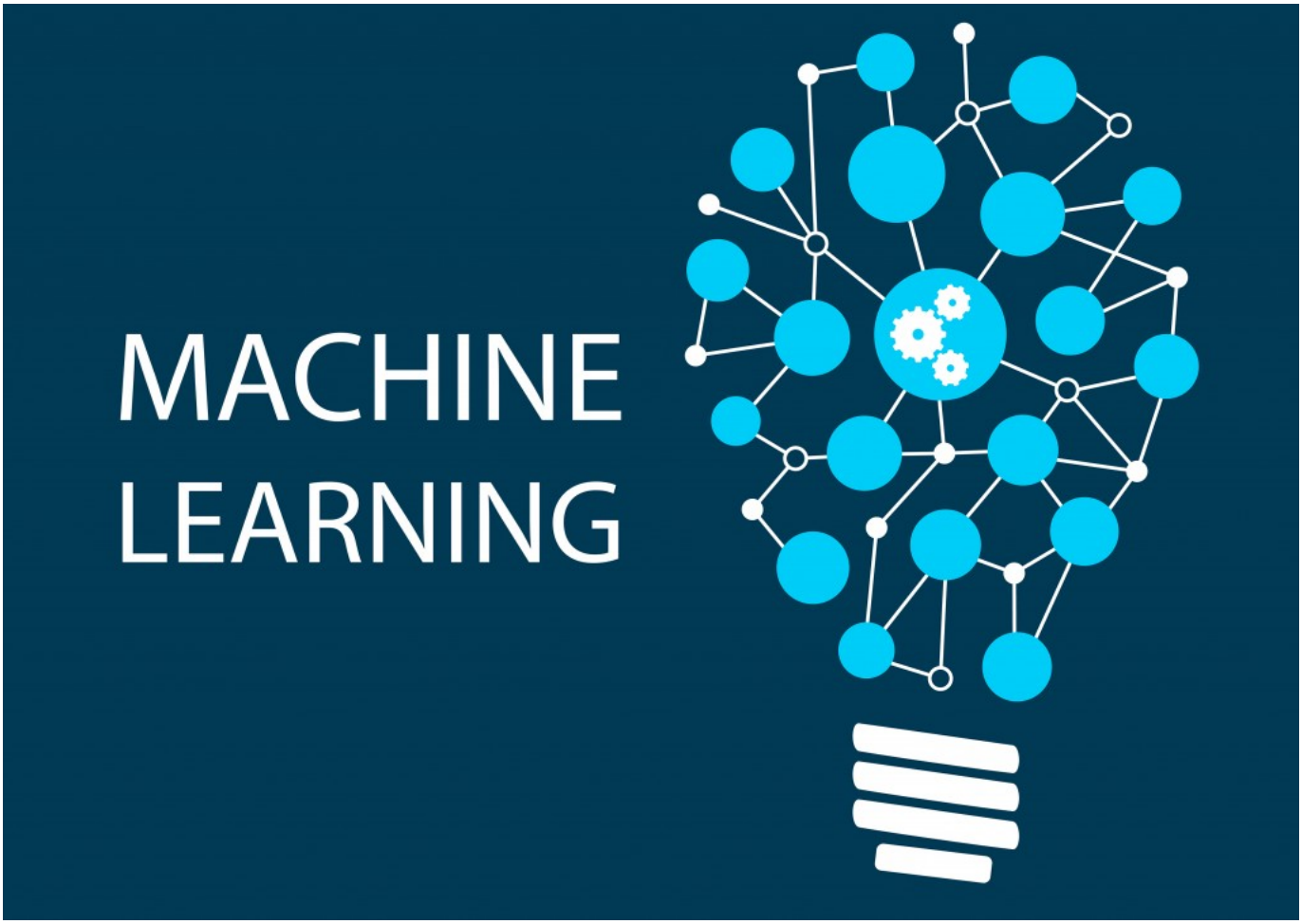
[78]:
```
NFLX
```

[78]:

High	Low	NFLX_Close(t)	Volume	SD20	Upper_Band	Lower_Band	NFLX_Close(t-1)	NFLX_Close(t-2)	...	Week	Is_month_end	Is_month_start	Is_quarter_end	Is_quarter_s
10.66	9.85	9.87	12295000.0	2.216304	20.726608	11.861392	10.30	17.43	...	43	0	0	0	
10.09	9.67	9.93	7863200.0	2.610476	21.135952	10.694048	9.87	10.30	...	43	0	0	0	

We have extracted some 55 new features for Predicting stock prices. We created many of them in which some analyses short term trend, some tells us about long term trends. We will now build Machine Learning models based on these features

Machine Learning Models:



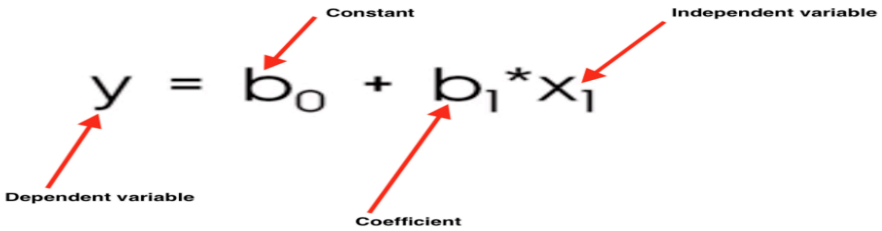
we will only consider the effect of previous many days of close prices as a input. As the best feature is the previous day prices, we will lag the data and try to tune the lookback days for improving the predictions.

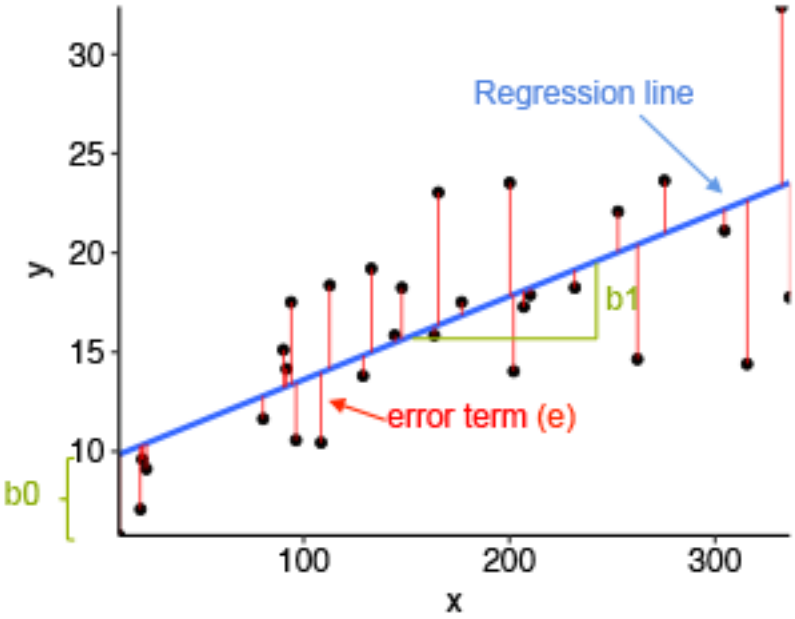
1. **Linear Regression:**

Linear regression is one of the most common supervised machine learning algorithms which shows the relationship between predictor variable and outcome variable

What good are regression models?

Regression models are used for predicting a real value, for example, salary or height. If your independent variable is time, then you are forecasting future values. Otherwise, your model is predicting present but unknown values. Our independent variable is time and hence it would be good for prediction.





Let's start building our model, we will use scikit-learn library and import the LinearRegression model from there:

Linear Regression

[63]:
```
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(X_train, Y_train)
```

[63]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

[64]:
```
print('LR Coefficients: \n', lr.coef_)
print('LR Intercept: \n', lr.intercept_)
```

```
LR Coefficients:
 [ 1.01498376e+00 -5.83723661e-02 -2.85129084e-02  6.90369215e-02
  2.85240973e-03 -7.00725926e-02  6.21425968e-02  3.12018578e-02
  1.61083992e-02 -3.21957547e-02 -3.53224986e-02  4.37226520e-03
  8.81393784e-03 -2.63556968e-02  1.69483195e-02  4.84697149e-02
 -5.18875705e-03 -5.22203430e-02 -7.20629444e-04  2.76041102e-02
  1.38226424e-01 -1.15417351e-02 -1.85852452e-01  1.28453801e-02
  4.43705483e-02  2.12218025e-03 -9.29770013e-04 -1.92769627e-03
  2.18568349e-02 -2.90492690e-03 -5.28196961e-02  5.76264350e-03
  7.63500684e-02 -5.03075672e-03 -1.02944251e-01  1.12795740e-03
  8.34729809e-02 -5.19942659e-03 -1.87264063e-02  4.81263366e-03
 -6.72307538e-02  2.28140298e-03  4.16139387e-03  5.73501569e-02
  4.83620721e-02  8.35904119e-02 -4.82460138e-02 -2.61396561e-02
 -4.60028208e-02 -8.06425980e-02]
LR Intercept:
 -0.0009187445617939716
```

[65]:
```
print("Performance (R^2): ", lr.score(X_train, Y_train))
```

```
Performance (R^2):  0.9973047458771007
```

We have tried various metrics for checking the predictions that we got. The below screenshot shows results of various metrics, like R-Squared, Explained variation, MAPE, MSE, MAE, RMSE and MAE.

[67]:
```
Y_train_pred = lr.predict(X_train)
Y_val_pred = lr.predict(X_val)
Y_test_pred = lr.predict(X_test)


print("Training R-squared: ",round(metrics.r2_score(Y_train,Y_train_pred),2))
print("Training Explained Variation: ",round(metrics.explained_variance_score(Y_train,Y_train_pred),2))
print('Training MAPE:', round(get_mape(Y_train,Y_train_pred), 2))
print('Training Mean Squared Error:', round(metrics.mean_squared_error(Y_train,Y_train_pred), 2))
print("Training RMSE: ",round(np.sqrt(metrics.mean_squared_error(Y_train,Y_train_pred)),2))
print("Training MAE: ",round(metrics.mean_absolute_error(Y_train,Y_train_pred),2))

print(' ')

print("Validation R-squared: ",round(metrics.r2_score(Y_val,Y_val_pred),2))
print("Validation Explained Variation: ",round(metrics.explained_variance_score(Y_val,Y_val_pred),2))
print('Validation MAPE:', round(get_mape(Y_val,Y_val_pred), 2))
print('Validation Mean Squared Error:', round(metrics.mean_squared_error(Y_train,Y_train_pred), 2))
print("Validation RMSE: ",round(np.sqrt(metrics.mean_squared_error(Y_val,Y_val_pred)),2))
print("Validation MAE: ",round(metrics.mean_absolute_error(Y_val,Y_val_pred),2))

print(' ')

print("Test R-squared: ",round(metrics.r2_score(Y_test,Y_test_pred),2))
print("Test Explained Variation: ",round(metrics.explained_variance_score(Y_test,Y_test_pred),2))
print('Test MAPE:', round(get_mape(Y_test,Y_test_pred), 2))
print('Test Mean Squared Error:', round(metrics.mean_squared_error(Y_test,Y_test_pred), 2))
print("Test RMSE: ",round(np.sqrt(metrics.mean_squared_error(Y_test,Y_test_pred)),2))
print("Test MAE: ",round(metrics.mean_absolute_error(Y_test,Y_test_pred),2))
```

```
Training R-squared:  1.0
Training Explained Variation:  1.0
Training MAPE: 1.43
Training Mean Squared Error: 0.93
Training RMSE:  0.97
Training MAE:  0.65

Validation R-squared:  0.94
Validation Explained Variation:  0.94
Validation MAPE: 0.97
Validation Mean Squared Error: 0.93
Validation RMSE:  1.41
Validation MAE:  1.06

Test R-squared:  0.83
```

Let's plot the actual and predicted values, as visual representation will help us better understand the model:

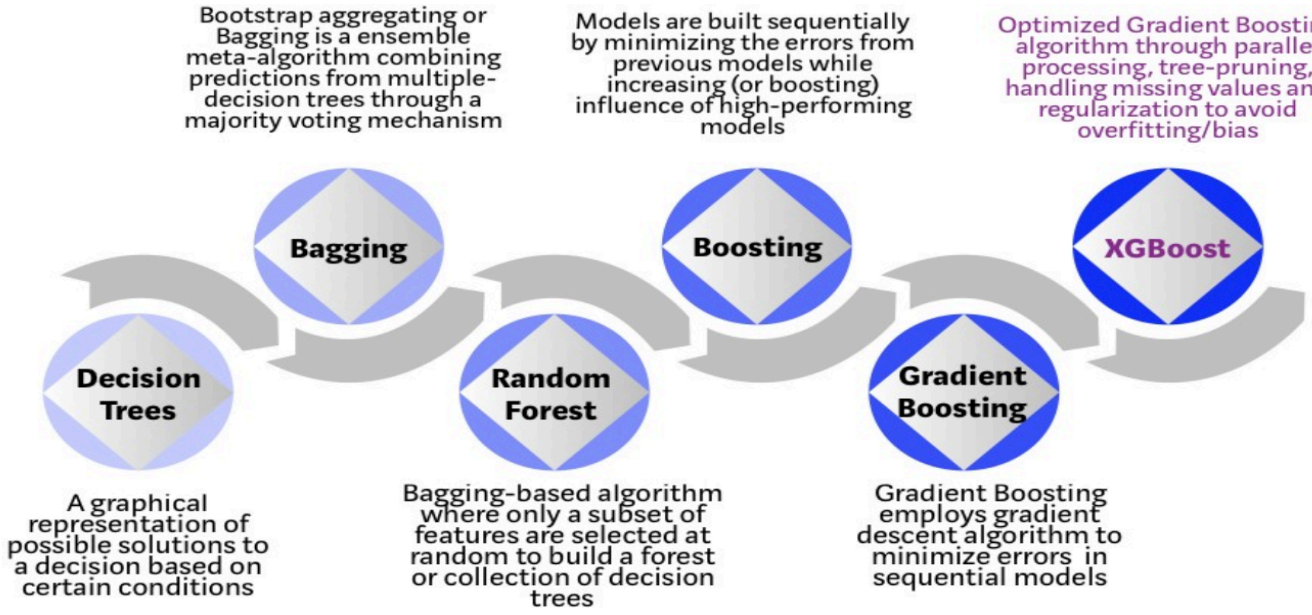
[70]:
```
df_pred[['Actual', 'Predicted']].plot()
```

[70]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1bca8a6b588>
```

2. XGBoost:

What is XGBoost?

XGBoost is a decision-tree-based ensemble Machine Learning algorithm that leverages a gradient boosting framework. When we have small to medium structured data tree-based algorithm tends to work really well.



Evolution of XGBoost Algorithm from Decision Trees

The above picture shows the evolution of Extreme Gradient Boost.

Let's prepare our model:

[71]:
```
from xgboost import XGBRegressor
```

[75]:
```
n_estimators = 200              # Number of boosted trees to fit. default = 100
max_depth = 30                  # Maximum tree depth for base learners. default = 3
learning_rate = 0.1             # Boosting learning rate (xgb's "eta"). default = 0.1
min_child_weight = 1            # Minimum sum of instance weight(hessian) needed in a child. default = 1
subsample = 1                   # Subsample ratio of the training instance. default = 1
colsample_bytree = 1            # Subsample ratio of columns when constructing each tree. default = 1
colsample_bylevel = 1           # Subsample ratio of columns for each split, in each level. default = 1
gamma = 0                       # Minimum loss reduction required to make a further partition on a leaf node of the tree. default=0

model_seed = 42
```

[76]:
```
xgb = XGBRegressor(seed=model_seed,
                   n_estimators=n_estimators,
                   max_depth=max_depth,
                   learning_rate=learning_rate,
                   min_child_weight=min_child_weight,
                   subsample=subsample,
                   colsample_bytree=colsample_bytree,
                   colsample_bylevel=colsample_bylevel,
                   gamma=gamma)
xgb.fit(X_train, Y_train)
```

[76]:
```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=30, min_child_weight=1, missing=None, n_estimators=200,
       n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=42, silent=True,
       subsample=1)
```

Now we will fit our training data into the model:

[76]:
```
xgb = XGBRegressor(seed=model_seed,
                        n_estimators=n_estimators,
                        max_depth=max_depth,
                        learning_rate=learning_rate,
                        min_child_weight=min_child_weight,
                        subsample=subsample,
                        colsample_bytree=colsample_bytree,
                        colsample_bylevel=colsample_bylevel,
                        gamma=gamma)
xgb.fit(X_train, Y_train)
```

[76]:
```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=30, min_child_weight=1, missing=None, n_estimators=200,
       n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=42, silent=True,
       subsample=1)
```

Let's check the output against various metrics that we defined earlier:

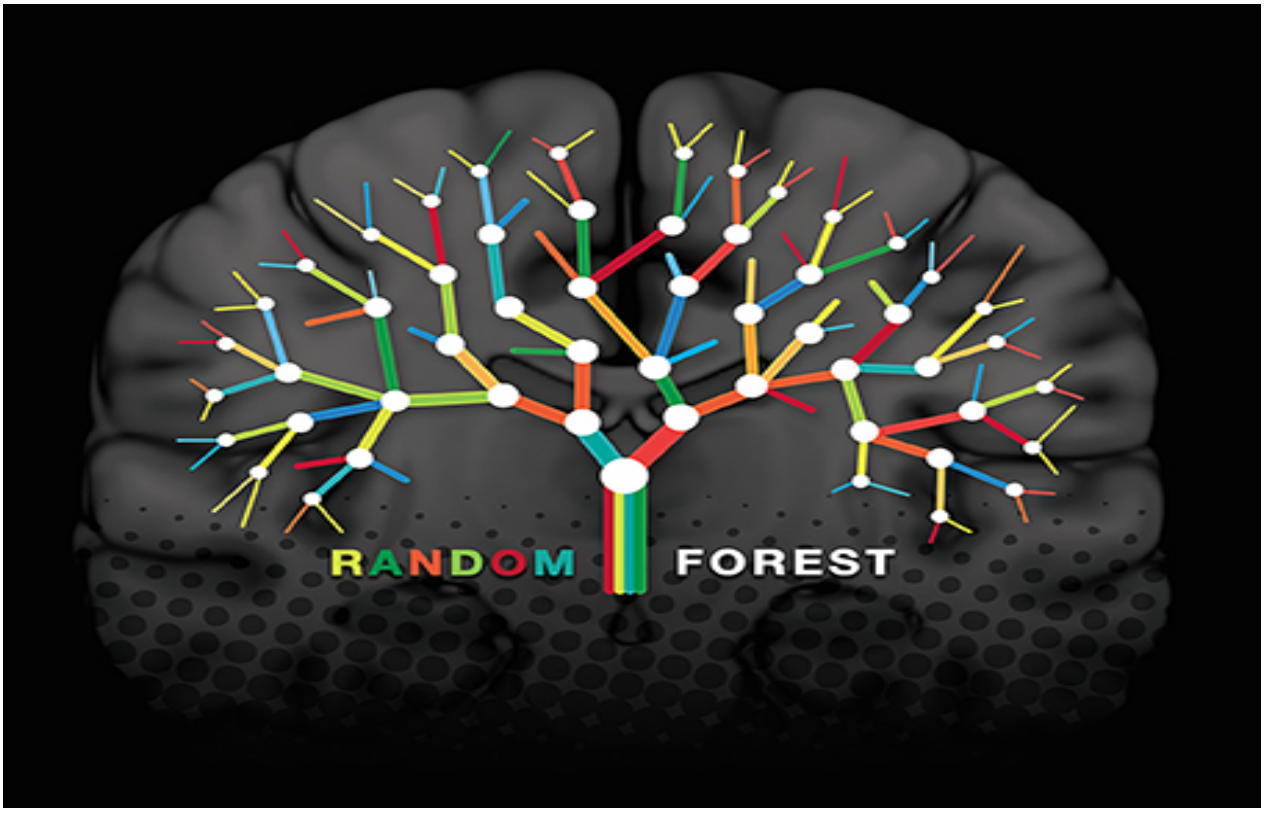
```
print("Training R-squared: ",round(metrics.r2_score(Y_train,Y_train_pred),2))
print("Training Explained Variation: ",round(metrics.explained_variance_score(Y_train,Y_train_pred),2))
print('Training MAPE:', round(get_mape(Y_train,Y_train_pred), 2))
print('Training Mean Squared Error:', round(metrics.mean_squared_error(Y_train,Y_train_pred), 2))
print("Training RMSE: ",round(np.sqrt(metrics.mean_squared_error(Y_train,Y_train_pred)),2))
print("Training MAE: ",round(metrics.mean_absolute_error(Y_train,Y_train_pred),2))

print(' ')

print("Validation R-squared: ",round(metrics.r2_score(Y_val,Y_val_pred),2))
print("Validation Explained Variation: ",round(metrics.explained_variance_score(Y_val,Y_val_pred),2))
print('Validation MAPE:', round(get_mape(Y_val,Y_val_pred), 2))
print('Validation Mean Squared Error:', round(metrics.mean_squared_error(Y_train,Y_train_pred), 2))
print("Validation RMSE: ",round(np.sqrt(metrics.mean_squared_error(Y_val,Y_val_pred)),2))
print("Validation MAE: ",round(metrics.mean_absolute_error(Y_val,Y_val_pred),2))

print(' ')

print("Test R-squared: ",round(metrics.r2_score(Y_test,Y_test_pred),2))
print("Test Explained Variation: ",round(metrics.explained_variance_score(Y_test,Y_test_pred),2))
print('Test MAPE:', round(get_mape(Y_test,Y_test_pred), 2))
print('Test Mean Squared Error:', round(metrics.mean_squared_error(Y_test,Y_test_pred), 2))
print("Test RMSE: ",round(np.sqrt(metrics.mean_squared_error(Y_test,Y_test_pred)),2))
print("Test MAE: ",round(metrics.mean_absolute_error(Y_test,Y_test_pred),2))
```

```
Training R-squared:  1.0
Training Explained Variation:  1.0
Training MAPE: 0.0
Training Mean Squared Error: 0.0
Training RMSE:  0.0
Training MAE:  0.0

Validation R-squared:  0.6
Validation Explained Variation:  0.61
Validation MAPE: 2.49
Validation Mean Squared Error: 0.0
Validation RMSE:  3.76
Validation MAE:  2.79

Test R-squared:  -44.77
Test Explained Variation:  -0.07
Test MAPE: 17.79
Test Mean Squared Error: 591.09
Test RMSE:  24.31
Test MAE:  24.03
```

Visual comparison of actual and predicted results:

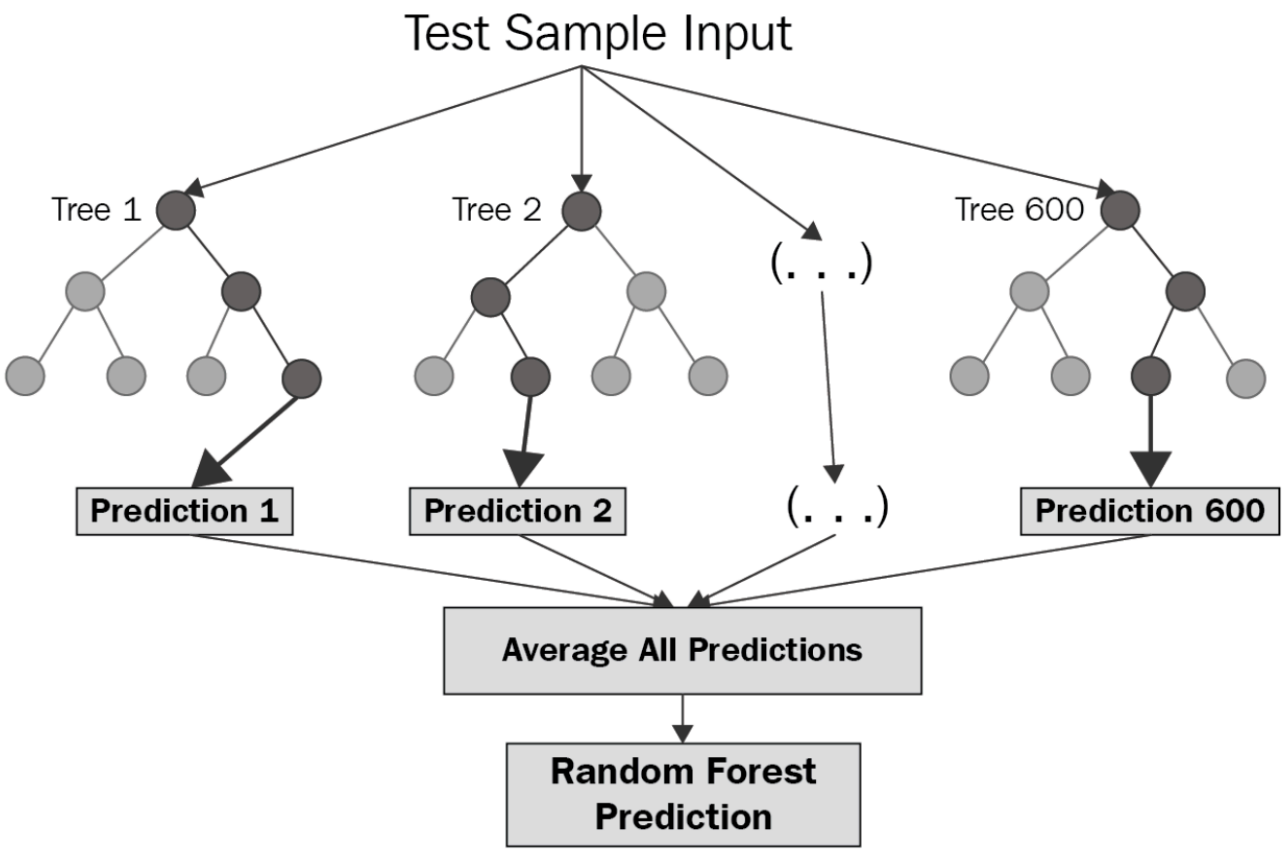
```
[79]: df_pred[['Actual', 'Predicted']].plot()
[79]: <matplotlib.axes._subplots.AxesSubplot at 0x1bca8b46b70>
```

3. **Random Forest**:



This Photo by Unknown Author is licensed under CC BY

Random forests are a supervised learning algorithm. It can be used both for classification and regression. It is also the most flexible and easy to use the algorithm. A forest is comprised of trees. It is said that the more trees it has, the more robust a forest is. Random forests create decision trees on randomly selected data samples, gets a prediction from each tree and selects the best solution by means of voting. It also provides a pretty good indicator of the feature importance. Random Forest Classifier is an ensemble algorithm. Ensemble algorithms are those which combines more than one algorithm of a same or different kind for classifying objects. For example, running prediction over Naive Bayes, SVM, and Decision Tree and then taking a vote for final consideration of class for a test object.



Random Forest Structure

Let's create our Random Forest Model:

Random Forest

```
[83]: rf = RandomForestRegressor(n_estimators=100, max_depth=50, random_state=42)
      rf.fit(X_train, Y_train)
```

```
[83]: RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=50,
                  max_features='auto', max_leaf_nodes=None,
                  min_impurity_decrease=0.0, min_impurity_split=None,
                  min_samples_leaf=1, min_samples_split=2,
                  min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
                  oob_score=False, random_state=42, verbose=0, warm_start=False)
```

We will now again check our models with the metrics that we defined earlier:

[84]:
```
Y_train_pred = rf.predict(X_train)
Y_val_pred = rf.predict(X_val)
Y_test_pred = rf.predict(X_test)


print("Training R-squared: ",round(metrics.r2_score(Y_train,Y_train_pred),2))
print("Training Explained Variation: ",round(metrics.explained_variance_score(Y_train,Y_train_pred),2))
print('Training MAPE:', round(get_mape(Y_train,Y_train_pred), 2))
print('Training Mean Squared Error:', round(metrics.mean_squared_error(Y_train,Y_train_pred), 2))
print("Training RMSE: ",round(np.sqrt(metrics.mean_squared_error(Y_train,Y_train_pred)),2))
print("Training MAE: ",round(metrics.mean_absolute_error(Y_train,Y_train_pred),2))

print(' ')

print("Validation R-squared: ",round(metrics.r2_score(Y_val,Y_val_pred),2))
print("Validation Explained Variation: ",round(metrics.explained_variance_score(Y_val,Y_val_pred),2))
print('Validation MAPE:', round(get_mape(Y_val,Y_val_pred), 2))
print('Validation Mean Squared Error:', round(metrics.mean_squared_error(Y_train,Y_train_pred), 2))
print("Validation RMSE: ",round(np.sqrt(metrics.mean_squared_error(Y_val,Y_val_pred)),2))
print("Validation MAE: ",round(metrics.mean_absolute_error(Y_val,Y_val_pred),2))

print(' ')

print("Test R-squared: ",round(metrics.r2_score(Y_test,Y_test_pred),2))
print("Test Explained Variation: ",round(metrics.explained_variance_score(Y_test,Y_test_pred),2))
print('Test MAPE:', round(get_mape(Y_test,Y_test_pred), 2))
print('Test Mean Squared Error:', round(metrics.mean_squared_error(Y_test,Y_test_pred), 2))
print("Test RMSE: ",round(np.sqrt(metrics.mean_squared_error(Y_test,Y_test_pred)),2))
print("Test MAE: ",round(metrics.mean_absolute_error(Y_test,Y_test_pred),2))
```

```
Training R-squared:  1.0
Training Explained Variation:  1.0
Training MAPE: 0.57
Training Mean Squared Error: 0.15
Training RMSE:  0.39
Training MAE:  0.26

Validation R-squared:  0.72
Validation Explained Variation:  0.73
Validation MAPE: 2.09
Validation Mean Squared Error: 0.15
Validation RMSE:  3.11
Validation MAE:  2.3



Test R-squared:  -29.79
Test Explained Variation:  -0.02
Test MAPE: 14.51
Test Mean Squared Error: 397.58
Test RMSE:  19.94
Test MAE:  19.61
```

Let's check the visual representation of predicted and actual values:

[86]:
```
df_pred[['Actual', 'Predicted']].plot()
```

[86]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1bca8c04eb8>
```

4. Long Short-Term Memory (LSTM)

"LSTM are a special type of RNNs. It is designed to tackle long term dependency problems. Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections. It can not only process single data points (such as images), but also entire sequences of data (such as speech or video). For example, LSTM is applicable to tasks such as unsegmented, connected handwriting recognition, speech recognition and anomaly detection in network traffic or IDS's (intrusion detection systems)." – Wikipedia

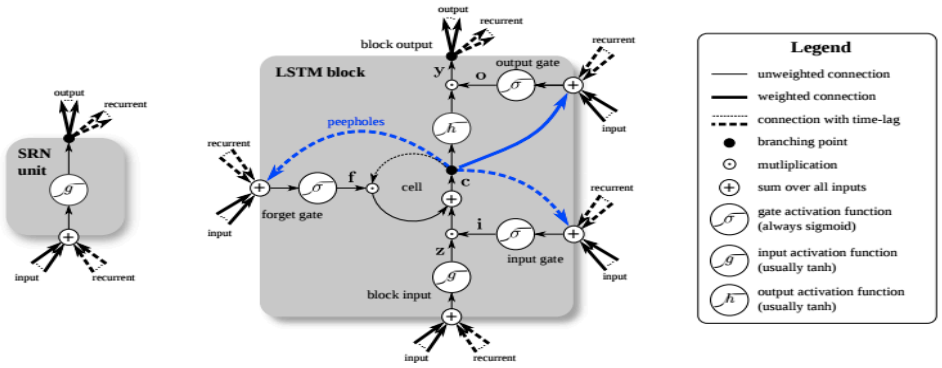


Figure 1. Detailed schematic of the Simple Recurrent Network (SRN) unit (left) and a Long Short-Term Memory block (right) as used in the hidden layers of a recurrent neural network.

Let’s build our LSTM model:

[23]:
```
from keras.models import Sequential
from keras.layers import Dense, Dropout, LSTM, GRU
from keras.optimizers import Adam
```

[24]:
```
lstm = Sequential()
lstm.add(LSTM(64, return_sequences=True, recurrent_dropout=0.2, dropout=0.2, input_shape=(X_train.shape[1],1)))
lstm.add(LSTM(32, recurrent_dropout=0.2, dropout=0.2))
lstm.add(Dense(1))

lstm.compile(loss='mean_squared_error',
             optimizer='adam',
            metrics = ['mae'])
lstm.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 50, 64)            16896
_________________________________________________________________
lstm_2 (LSTM)                (None, 32)                12416
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 29,345
Trainable params: 29,345
Non-trainable params: 0
_________________________________________________________________
```

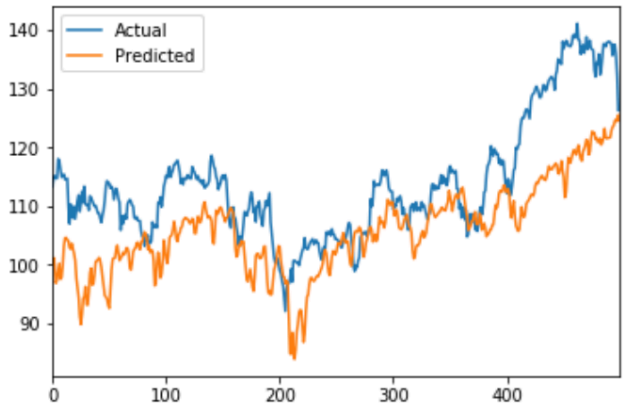
Next step is to run our model. Here we are training it for 30 epochs.

[25]:
```
history_lstm = lstm.fit(X_train, Y_train, batch_size=16, epochs=30, validation_data=(X_test, Y_test), shuffle=False)
```

```
Train on 3345 samples, validate on 500 samples
Epoch 1/30
3345/3345 [==============================] - 19s 6ms/step - loss: 0.0040 - mean_absolute_error: 0.0503 - val_loss: 0.0208 - val_mean_absolute_e
rror: 0.1323
Epoch 2/30
3345/3345 [==============================] - 22s 7ms/step - loss: 0.0129 - mean_absolute_error: 0.0757 - val_loss: 0.0072 - val_mean_absolute_e
rror: 0.0705
Epoch 3/30
3345/3345 [==============================] - 21s 6ms/step - loss: 0.0120 - mean_absolute_error: 0.0728 - val_loss: 0.0065 - val_mean_absolute_e
rror: 0.0592
Epoch 4/30
3345/3345 [==============================] - 21s 6ms/step - loss: 0.0114 - mean_absolute_error: 0.0722 - val_loss: 0.0064 - val_mean_absolute_e
rror: 0.0586
Epoch 5/30
3345/3345 [==============================] - 20s 6ms/step - loss: 0.0127 - mean_absolute_error: 0.0784 - val_loss: 0.0176 - val_mean_absolute_e
rror: 0.1222
Epoch 6/30
3345/3345 [==============================] - 21s 6ms/step - loss: 0.0178 - mean_absolute_error: 0.0901 - val_loss: 0.0358 - val_mean_absolute_e
rror: 0.1704
Epoch 7/30
3345/3345 [==============================] - 21s 6ms/step - loss: 0.0197 - mean_absolute_error: 0.0989 - val_loss: 0.0873 - val_mean_absolute_e
rror: 0.2836
Epoch 8/30
3345/3345 [==============================] - 21s 6ms/step - loss: 0.0207 - mean_absolute_error: 0.1075 - val_loss: 0.1140 - val_mean_absolute_e
rror: 0.3274
Epoch 9/30
3345/3345 [==============================] - 21s 6ms/step - loss: 0.0183 - mean_absolute_error: 0.1027 - val_loss: 0.1174 - val_mean_absolute_e
rror: 0.3325
Epoch 10/30
3345/3345 [==============================] - 21s 6ms/step - loss: 0.0184 - mean_absolute_error: 0.1026 - val_loss: 0.1223 - val_mean_absolute_e
rror: 0.3398
Epoch 11/30
3345/3345 [==============================] - 21s 6ms/step - loss: 0.0195 - mean_absolute_error: 0.1106 - val_loss: 0.0079 - val_mean_absolute_e
rror: 0.0755
Epoch 12/30
3345/3345 [==============================] - 21s 6ms/step - loss: 0.0123 - mean_absolute_error: 0.0816 - val_loss: 0.0146 - val_mean_absolute_e
rror: 0.1123
Epoch 13/30
3345/3345 [==============================] - 21s 6ms/step - loss: 0.0102 - mean_absolute_error: 0.0757 - val_loss: 0.0128 - val_mean_absolute_e
rror: 0.0972
```

Now let's visualize our model to see how it is performing against predicted and actual results.

```
[36]: df_pred[['Actual', 'Predicted']].plot()

[36]: <matplotlib.axes._subplots.AxesSubplot at 0x2a87de02eb8>
```



Random Forest, XG Boost and Linear regression seems to work very well on such a long data as well. We have taken around 12-15 years of historical data which was divided into Training, Validation and Testing.

These algorithms are able to catch the trend and move close to the Actual closing price in the right direction with a very low Mean Absolute error (MAE). The Evaluation Metrics is above to see how each algorithm performed on each stock. They worked so well because of the Feature Extraction we did to extract around 60 features including lagged Index funds prices, Technical Indicators like Exponential Moving Average, RSI, ADR, Willam's R, bollinger bands and many more.

LSTMs followed the trend but did not perform well on price prediction as expected, as the data is not using any sequence of previous many time steps. and LSTM will work better on time series data once we provide it with previous 30-60 days of lookback data for every prediction.

LSTM with Multiple Time steps of Close Price as a Time Series Sequence

Previously, we tried different algorithms like XG Boost, Random Forest and LSTM using Extensive feature extraction where we extracted around 60 features using different Technical Indicators as well as prices of other related Index funds like Nasdaq-100 QQQ ETF, SNP 500 Index and DJIA Index fund. Linear Regression, Random Forest and XG Boost performed really well, on an average an MAE (Mean Absolute error) of around 0.36. We also checked for Feature importance using RF. But LSTM did not perform good with these features. As expected, as we did not feed any long sequence of previous data. The Indicators for a previous day are not enough for Long short-term memory.

Now, we will focus on the Close price sequence of previous days. And we will also tune the number of lagged days to consider for prediction. With LSTM, we will try a lookback of 30 days approx. 1+ months to predict the price of next day.

Create Lagged Features for getting Previous day prices for a lookback of 30 days

[5]:
```
def prepare_lagged_features(df_Stock, lag_stock, lag_index):

        print('Preparing Lagged Features for Stock, Index Funds.....')
        lags = range(1, lag_stock+1)
        lag_cols= ['Close']
        df_Stock=df_Stock.assign(**{
            '{}(t-{})'.format(col, l): df_Stock[col].shift(l)
            for l in lags
            for col in lag_cols
        })


        lags = range(1, lag_index+1)
        lag_cols= ['QQQ_Close','SnP_Close','DJIA_Close']
        df_Stock= df_Stock.assign(**{
            '{}(t-{})'.format(col, l): df_Stock[col].shift(l)
            for l in lags
            for col in lag_cols
        })

        df_Stock = df_Stock.drop(columns=lag_cols)


        remove_lags_na = max(lag_stock, lag_index) + 1
        print('Removing NAN rows - ', str(remove_lags_na))
        df_Stock = df_Stock.iloc[remove_lags_na:,]
        return df_Stock
```

[6]:
```
df_Stock = prepare_lagged_features(df_Stock, lag_stock = 20, lag_index = 5)
df_Stock.head()
```

```
Preparing Lagged Features for Stock, Index Funds.....
Removing NAN rows -  21
```

[6]:

	Open	High	Low	Close	Diff	High-low	Close(t-1)	Close(t-2)	Close(t-3)	Close(t-4)	...	DJIA_Close(t-2)	QQQ_Close(t-3)	SnP_Close(t-3)	DJIA_Close(t-3)	QQQ_Close(t-4)	SnP_Close(t-4)
21	95.25	95.50	94.65	94.89	-0.36	0.85	95.92	95.32	94.79	93.61	...	10539.0098	38.29	1173.4800	10469.8398	37.79	1162.9100
22	95.30	96.63	95.30	95.46	0.16	1.33	94.89	95.92	95.32	94.79	...	10550.2402	38.66	1184.1700	10539.0098	38.29	1173.4800

Scale features for LSTM Training

[12]:
```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(Y_train)
```

[13]:
```
x_train, y_train = [], []
for i in range(60,len(Y_train)):
    x_train.append(scaled_data[i-60:i,0])
    y_train.append(scaled_data[i,0])
x_train, y_train = np.array(x_train), np.array(y_train)
x_train = np.reshape(x_train, (x_train.shape[0],x_train.shape[1],1))
```

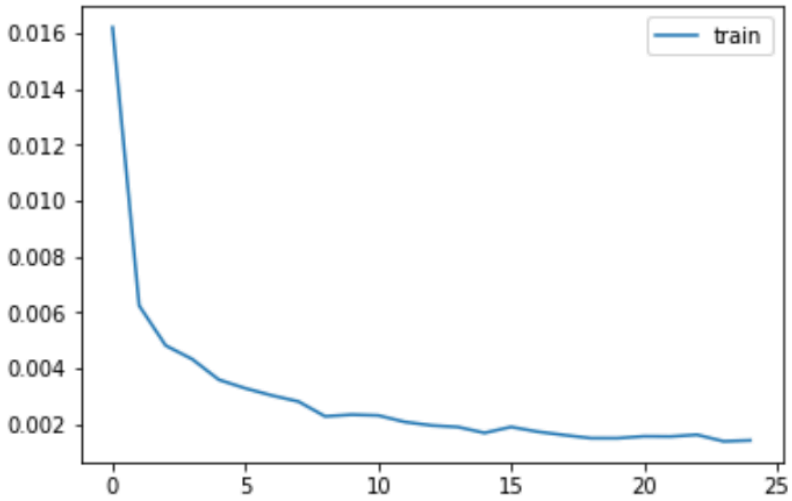
Create and fit the LSTM network

[14]:
```
lstm = Sequential()
lstm.add(LSTM(units=50, return_sequences=True, input_shape=(x_train.shape[1],1)))
lstm.add(LSTM(units=50))
lstm.add(Dense(1))

lstm.compile(loss='mean_squared_error', optimizer='adam')
lstm.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 60, 50)            10400
_________________________________________________________________
lstm_2 (LSTM)                (None, 50)                20200
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 51
=================================================================
Total params: 30,651
Trainable params: 30,651
Non-trainable params: 0
_________________________________________________________________
```

Let’s visualize the model loss:

[17]:
```
plt.plot(history_lstm.history['loss'], label='train')
#plt.plot(history_lstm.history['val_loss'], label='test')
plt.legend()
plt.show()
```



As we can see the lagged LSTM works really well when we compare the predicted vs actual prices.

[28]:
```
Y_test
```

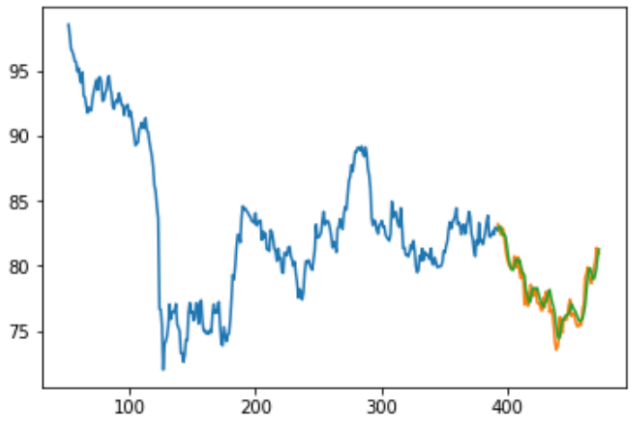
[28]:

	Close	predict
392	83.23	82.741882
393	82.90	83.027023
394	82.46	83.022690
395	82.39	82.694038
396	82.89	82.377693
...	...	...
468	79.88	79.000824
469	80.32	79.350121
470	81.40	79.875824
471	81.22	80.744591
472	80.97	81.276100

81 rows × 2 columns

[22]:
```
#for plotting
test_result=Y_test
test_result['predict']=closing_price
plt.plot(train['Close'])
plt.plot(test_result[['Close','predict']])
```

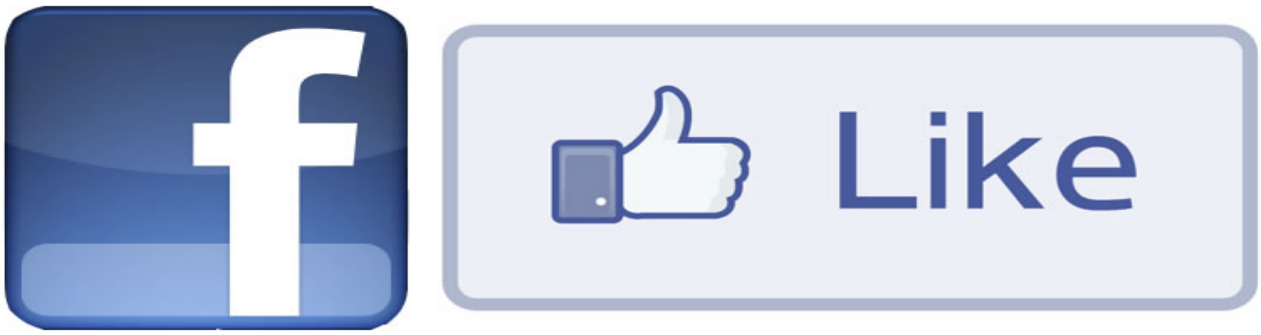
[22]:
```
[<matplotlib.lines.Line2D at 0x2479bbd2d68>,
 <matplotlib.lines.Line2D at 0x2479bbd2eb8>]
```



We got Amazing results by using a 60-day lookup window for Previous Closing Prices, Index funds and by considering the difference between the High low prices.

- R2 - 0.87
- MAPE - 3.24
- RMSE - 0.84
- MAE - 0.67

Stock Prediction using fb 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.

It forecasts time series data based on an additive model in which non-linear trends are fit with yearly, weekly, or 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. Prophet is open source software released by Facebook's core Data Science team.

[7]:
```
Stock.columns = ['ds', 'y']

prophet_model = Prophet(yearly_seasonality=True, daily_seasonality=True)
prophet_model.add_country_holidays(country_name='US')
prophet_model.add_seasonality(name='monthly', period=30.5, fourier_order=5)
```

[7]:
```
<fbprophet.forecaster.Prophet at 0x26c0926dfd0>
```

[8]:
```
prophet_model.fit(Stock)
```

[8]:
```
<fbprophet.forecaster.Prophet at 0x26c0926dfd0>
```

[9]:
```
future = prophet_model.make_future_dataframe(periods=30)
future.tail()
```

[9]:

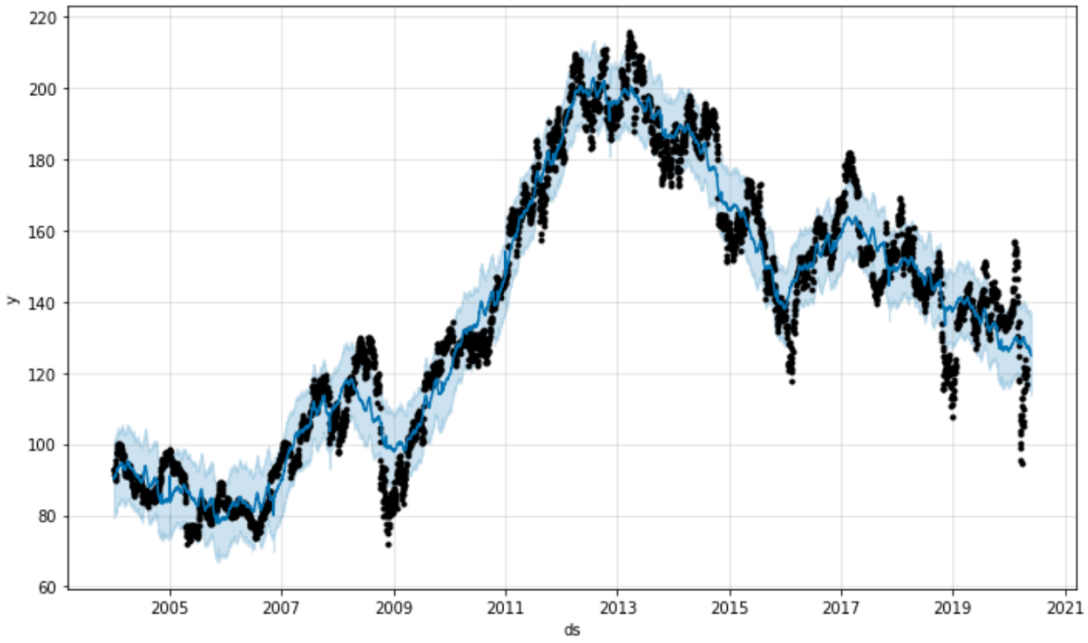
	ds
4131	2020-05-20
4132	2020-05-21
4133	2020-05-22
4134	2020-05-23
4135	2020-05-24

[10]:
```
forcast = prophet_model.predict(future)
forcast.tail()
```

[10]:

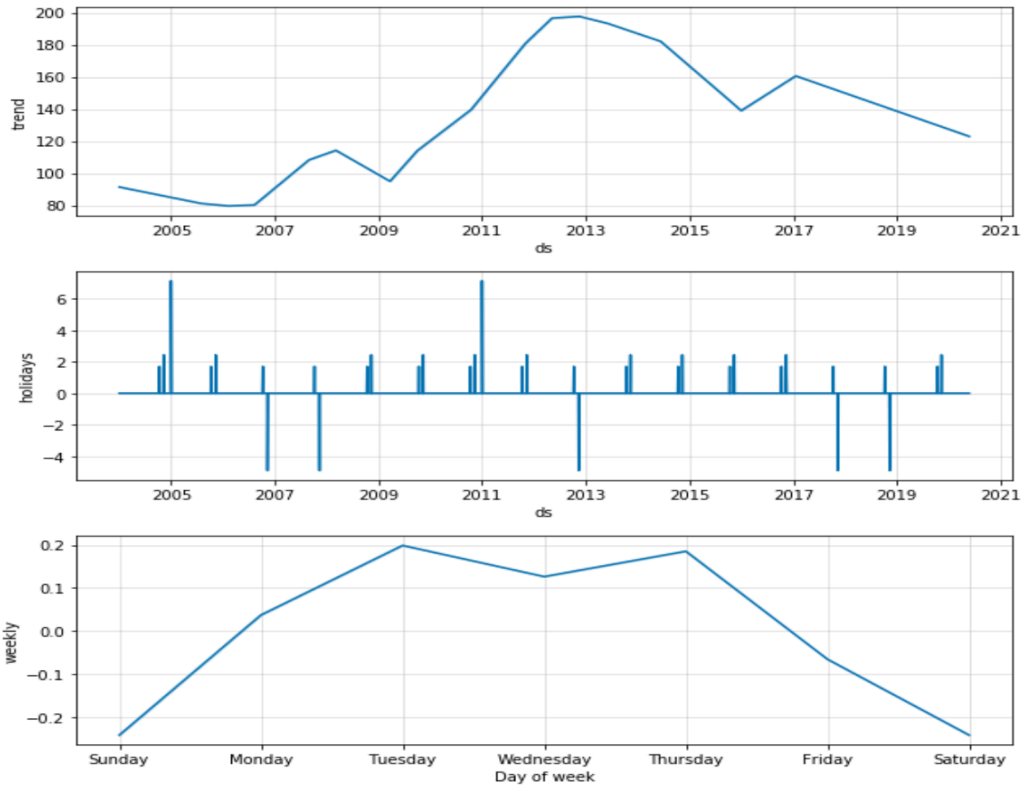
	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	Christmas Day	Christmas Day_lower	Christmas Day_upper	Christmas Day (Observed)	...	weekly	weekly_lower	weekly_upper	yearly	y
4131	2020-05-20	123.030223	116.150340	137.469394	123.030223	123.030223	0.0	0.0	0.0	0.0	...	0.126468	0.126468	0.126468	0.882137	

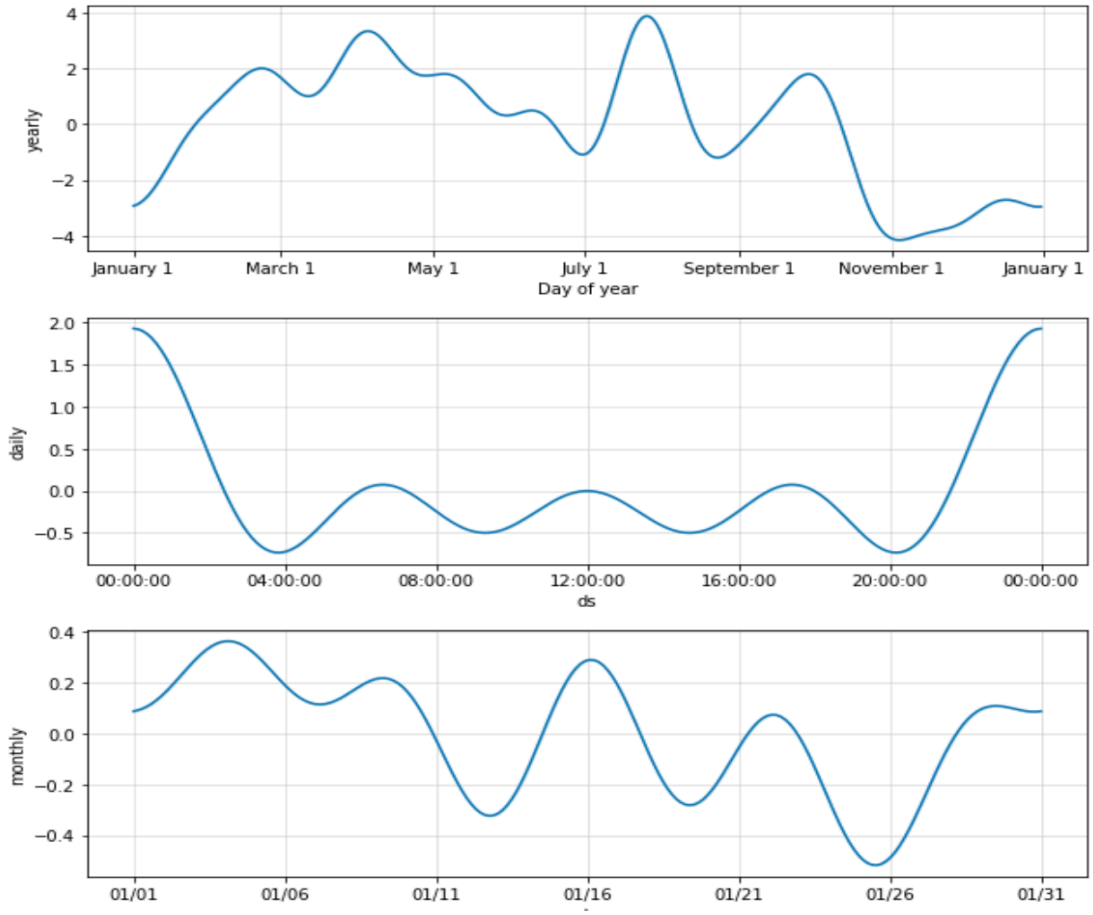
```
[11]: prophet_model.plot(forcast);
```



If you want to visualize the individual forecast components, we can use Prophet's built-in plot components method like below

```
[12]: prophet_model.plot_components(forcast);
```





Prediction Performance

The performance metrics utility can be used to compute some useful statistics of the prediction performance (yhat, yhat_lower, and yhat_upper compared to y), as a function of the distance from the cutoff (how far into the future the prediction was). The statistics computed are mean squared error (MSE), root mean squared error (RMSE), mean absolute error (MAE), mean absolute percent error (MAPE), and coverage of the yhat_lower and yhat_upper estimates.

[15]:
```
from fbprophet.diagnostics import cross_validation, performance_metrics
df_cv = cross_validation(prophet_model, horizon='180 days')
df_cv.head()
```

```
INFO:fbprophet:Making 59 forecasts with cutoffs between 2005-07-12 00:00:00 and 2019-10-27 00
```

[15]:

	ds	yhat	yhat_lower	yhat_upper	y	cutoff
0	2005-07-13	73.682183	71.495715	75.790542	81.45	2005-07-12
1	2005-07-14	73.410412	71.437530	75.639369	82.42	2005-07-12
2	2005-07-15	73.021985	70.781670	75.132137	82.38	2005-07-12
3	2005-07-18	73.289362	71.045425	75.499744	81.81	2005-07-12
4	2005-07-19	73.335579	71.141799	75.455836	83.70	2005-07-12

[16]:
```
df_cv
```

[16]:

	ds	yhat	yhat_lower	yhat_upper	y	cutoff
0	2005-07-13	73.682183	71.495715	75.790542	81.45	2005-07-12
1	2005-07-14	73.410412	71.437530	75.639369	82.42	2005-07-12
2	2005-07-15	73.021985	70.781670	75.132137	82.38	2005-07-12
3	2005-07-18	73.289362	71.045425	75.499744	81.81	2005-07-12
4	2005-07-19	73.335579	71.141799	75.455836	83.70	2005-07-12
...	...	...	...	...	...	...
7313	2020-04-20	129.598327	118.836802	139.811361	120.41	2019-10-27
7314	2020-04-21	129.456418	118.360249	140.835334	116.76	2019-10-27
7315	2020-04-22	129.094986	118.233972	140.602861	119.31	2019-10-27
7316	2020-04-23	128.969940	118.486456	139.772499	121.35	2019-10-27
7317	2020-04-24	128.739500	117.884714	139.546177	121.40	2019-10-27

7318 rows × 6 columns

NEXT STEP IS TO CREATE A PIPELINE AS RUNNING EVERY STEP ONE-By-ONE WOULD BE CUMBERSOME

END RESULT:

```
[5]: Metrics = pd.DataFrame.from_dict({(i,j): eval_metrics[i][j]
                                       for i in eval_metrics.keys()
                                       for j in eval_metrics[i].keys()},
                                      orient='index')
     Metrics
```

[5]:

		Train_MAE	Train_RSq	Train_MAPE	Train_RMSE	Test_MAE	Test_RSq	Test_MAPE	Test_RMSE
NKE	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.22	1.00	0.30	0.57	0.68	0.97	0.85	0.93
	XGBoost	0.29	1.00	0.40	0.36	0.67	0.97	0.84	0.93
	LSTM	16.56	-0.08	21.40	20.75	15.18	-4.85	17.50	16.69
IBM	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.32	1.00	0.21	0.48	1.22	0.97	0.94	1.75
	XGBoost	0.31	1.00	0.20	0.40	1.44	0.96	1.09	1.90
	LSTM	24.06	0.00	16.77	29.73	22.87	-6.14	17.35	24.63
KO	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.09	1.00	0.19	0.30	0.33	0.98	0.66	0.46
	XGBoost	0.21	1.00	0.45	0.28	0.36	0.97	0.72	0.50
	LSTM	7.75	-0.11	13.87	11.38	5.18	-2.38	10.00	6.18
GS	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.49	1.00	0.32	0.69	2.33	0.96	1.17	3.08
	XGBoost	0.32	1.00	0.21	0.41	2.80	0.94	1.42	3.86
	LSTM	46.42	-0.54	35.62	54.62	14.98	-0.06	7.49	17.98
JNJ	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.15	1.00	0.17	0.24	1.24	0.90	0.91	1.71
	XGBoost	0.26	1.00	0.31	0.34	0.93	0.94	0.69	1.27
	LSTM	29.13	-0.72	42.05	34.40	15.22	-8.05	11.07	16.10
NVDA	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.25	1.00	0.48	0.61	16.75	0.72	10.68	20.00
	XGBoost	0.20	1.00	0.88	0.28	14.05	0.81	8.89	16.33
	LSTM	78.87	-0.59	449.12	81.48	53.32	-1.58	25.14	66.59

We will first define the Feature Extraction Pipeline:

Let's create a class with all the required functions, the final output from feature extraction pipeline would be a dataset which we will use in the **Model Building Pipeline.**

Pipeline to Scrape the Historical Stocks data and Extract features from it.

```
]:
class Stocks:
    def __init__(self, ticker, start_date, forcast_horz):
        self.Ticker = ticker
        self.Start_Date = start_date
        self.forcast_horz = forcast_horz


    def get_stock_data(self, Ticker):

        ALPHA_VANTAGE_API_KEY = 'XAGC5LBB1SI9RDLW'
        self.ts = TimeSeries(key= ALPHA_VANTAGE_API_KEY, output_format='pandas')
        print('Loading Historical Price data for ' + self.Ticker + '....')
        self.df_Stock, self.Stock_info = self.ts.get_daily(self.Ticker, outputsize='full')
        print(self.Stock_info)
        self.df_Stock = self.df_Stock.rename(columns={'1. open' : 'Open', '2. high': 'High', '3. low':'Low', '4. close': 'Close',
        self.df_Stock = self.df_Stock.rename_axis(['Date'])
        #sorting index
        self.Stock = self.df_Stock.sort_index(ascending=True, axis=0)
        #slicing the data for 15 years from '2004-01-02' to today
        self.Stock = self.Stock.loc[self.Start_Date:]

        self.Stock['Close'].plot(figsize=(10, 7))
        plt.title("Stock Price", fontsize=17)
        plt.ylabel('Price', fontsize=14)
        plt.xlabel('Time', fontsize=14)
        plt.grid(which="major", color='k', linestyle='-.', linewidth=0.5)
        plt.show()
```

```
def extract_Technical_Indicators(self, Ticker):

    print(' ')
    print('Feature extraction of technical Indicators....')
    #get Boolinger Bands
    self.Stock['MA_20'] = self.Stock.Close.rolling(window=20).mean()
    self.Stock['SD20'] = self.Stock.Close.rolling(window=20).std()
    self.Stock['Upper_Band'] = self.Stock.Close.rolling(window=20).mean() + (self.Stock['SD20']*2)
    self.Stock['Lower_Band'] = self.Stock.Close.rolling(window=20).mean() - (self.Stock['SD20']*2)
    print('Boolinger bands..')

    #shifting for lagged data
    self.Stock['S_Close(t-1)'] = self.Stock.Close.shift(periods=1)
    self.Stock['S_Close(t-2)'] = self.Stock.Close.shift(periods=2)
    self.Stock['S_Close(t-3)'] = self.Stock.Close.shift(periods=3)
    self.Stock['S_Close(t-5)'] = self.Stock.Close.shift(periods=5)
    self.Stock['S_Open(t-1)'] = self.Stock.Open.shift(periods=1)
    print('Lagged Price data for previous days....')

    #simple moving average
    self.Stock['MA5'] = self.Stock.Close.rolling(window=5).mean()
    self.Stock['MA10'] = self.Stock.Close.rolling(window=10).mean()
    self.Stock['MA20'] = self.Stock.Close.rolling(window=20).mean()
    self.Stock['MA50'] = self.Stock.Close.rolling(window=50).mean()
    self.Stock['MA200'] = self.Stock.Close.rolling(window=200).mean()
    print('Simple Moving Average....')

    #Exponential Moving Averages
    self.Stock['EMA10'] = self.Stock.Close.ewm(span=5, adjust=False).mean().fillna(0)
    self.Stock['EMA20'] = self.Stock.Close.ewm(span=5, adjust=False).mean().fillna(0)
    self.Stock['EMA50'] = self.Stock.Close.ewm(span=5, adjust=False).mean().fillna(0)
    self.Stock['EMA100'] = self.Stock.Close.ewm(span=5, adjust=False).mean().fillna(0)
    self.Stock['EMA200'] = self.Stock.Close.ewm(span=5, adjust=False).mean().fillna(0)
    print('Exponential Moving Average....')

    #Moving Average Convergance Divergances
    self.Stock['EMA_12'] = self.Stock.Close.ewm(span=12, adjust=False).mean()
    self.Stock['EMA_26'] = self.Stock.Close.ewm(span=26, adjust=False).mean()
    self.Stock['MACD'] = self.Stock['EMA_12'] - self.Stock['EMA_26']

    self.Stock['MACD_EMA'] = self.Stock.MACD.ewm(span=9, adjust=False).mean()
```

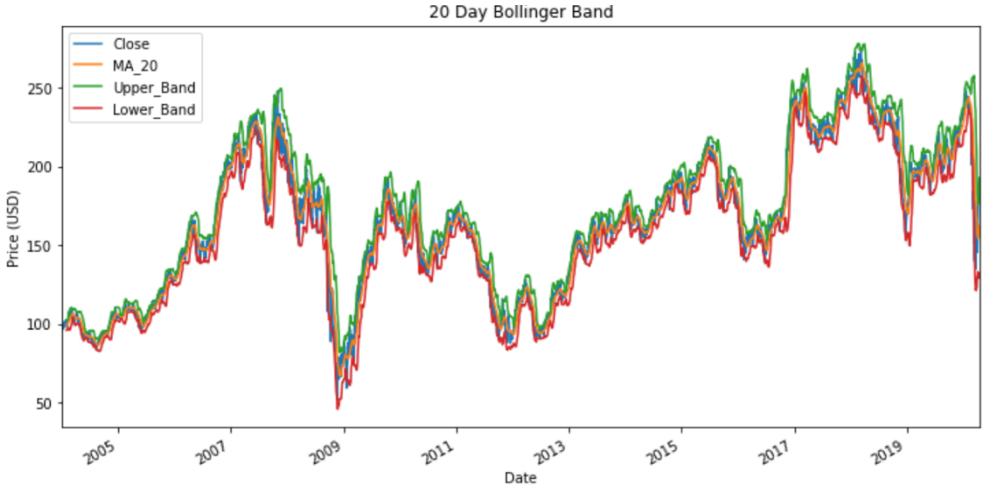
```
  #Average True Range
  self.Stock['ATR'] = talib.ATR(self.Stock['High'].values, self.Stock['Low'].values, self.Stock['Close'].values, timeperiod=14)

  #Average Directional Index
  self.Stock['ADX'] = talib.ADX(self.Stock['High'], self.Stock['Low'], self.Stock['Close'], timeperiod=14)

  #Commodity Channel index
  tp = (self.Stock['High'] + self.Stock['Low'] + self.Stock['Close']) /3
  ma = tp/20
  md = (tp-ma)/20
  self.Stock['CCI'] = (tp-ma)/(0.015 * md)
  print('Commodity Channel Index....')

  #Rate of Change
  self.Stock['ROC'] = ((self.Stock['Close'] - self.Stock['Close'].shift(10)) / (self.Stock['Close'].shift(10)))*100

  #Relative Strength Index
  self.Stock['RSI'] = talib.RSI(self.Stock.Close.values, timeperiod=14)

  #William's %R
  self.Stock['William%R'] = talib.WILLR(self.Stock.High.values, self.Stock.Low.values, self.Stock.Close.values, 14)

  #Stocastic K
  self.Stock['SO%K'] = ((self.Stock.Close - self.Stock.Low.rolling(window=14).min()) / (self.Stock.High.rolling(window=14).max(
  print('Stocastic %K ....')
  #Standard Deviation of last 5 day returns
  self.Stock['per_change'] = self.Stock.Close.pct_change()
  self.Stock['STD5'] = self.Stock.per_change.rolling(window=5).std()

  #Force Index
  self.Stock['ForceIndex1'] = self.Stock.Close.diff(1) * self.Stock.Volume
  self.Stock['ForceIndex20'] = self.Stock.Close.diff(20) * self.Stock.Volume
  print('Force index....')

  #print('Stock Data ', self.Stock)

  self.Stock[['Close', 'MA_20', 'Upper_Band', 'Lower_Band']].plot(figsize=(12,6))
  plt.title('20 Day Bollinger Band')
  plt.ylabel('Price (USD)')
  plt.show();

  self.Stock[['Close', 'MA20', 'MA200', 'MA50']].plot()
  plt.show();

  self.Stock[['MACD', 'MACD_EMA']].plot()
```

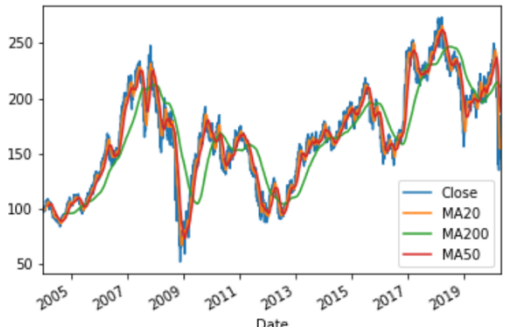
```
def get_IDXFunds_features(self, Ticker):
    print(' ')
    print('Fetching data for NASDAQ-100 Index Fund ETF QQQ & S&P 500 index ......')
    # Nasdaq-100 Index Fund ETF QQQ
    QQQ, QQQ_info = self.ts.get_daily('QQQ', outputsize='full')
    QQQ = QQQ.rename(columns={'1. open' : 'Open', '2. high': 'High', '3. low':'Low', '4. close': 'QQQ_Close', '5. volume': 'V
    QQQ = QQQ.rename_axis(['Date'])
    QQQ = QQQ.drop(columns=['Open', 'High', 'Low', 'Volume'])

    #sorting index
    QQQ = QQQ.sort_index(ascending=True, axis=0)
    #slicing the data for 15 years from '2004-01-02' to today
    QQQ = QQQ.loc[self.Start_Date:]
    QQQ['QQQ(t-1)'] = QQQ.QQQ_Close.shift(periods=1)
    QQQ['QQQ(t-2)'] =  QQQ.QQQ_Close.shift(periods=2)
    QQQ['QQQ(t-5)'] =  QQQ.QQQ_Close.shift(periods=5)

    QQQ['QQQ_MA10'] = QQQ.QQQ_Close.rolling(window=10).mean()
    #QQQ['QQQ_MA10_t'] = QQQ.QQQ_ClosePrev1.rolling(window=10).mean()
    QQQ['QQQ_MA20'] = QQQ.QQQ_Close.rolling(window=20).mean()
    QQQ['QQQ_MA50'] = QQQ.QQQ_Close.rolling(window=50).mean()


    #S&P 500 Index Fund
    SnP, SnP_info = self.ts.get_daily('INX', outputsize='full')
    SnP = SnP.rename(columns={'1. open' : 'Open', '2. high': 'High', '3. low':'Low', '4. close': 'SnP_Close', '5. volume': 'V
    SnP = SnP.rename_axis(['Date'])
    SnP = SnP.drop(columns=['Open', 'High', 'Low', 'Volume'])

    #sorting index
    SnP = SnP.sort_index(ascending=True, axis=0)
    #slicing the data for 15 years from '2004-01-02' to today
    SnP = SnP.loc[self.Start_Date:]
    SnP
    SnP['SnP(t-1))'] = SnP.SnP_Close.shift(periods=1)
    SnP['SnP(t-5)'] =  SnP.SnP_Close.shift(periods=5)


    #S&P 500 Index Fund
    DJIA, DJIA_info = self.ts.get_daily('DJI', outputsize='full')
    DJIA = DJIA.rename(columns={'1. open' : 'Open', '2. high': 'High', '3. low':'Low', '4. close': 'DJIA_Close', '5. volume':
    DJIA = DJIA.rename_axis(['Date'])
    DJIA = DJIA.drop(columns=['Open', 'High', 'Low', 'Volume'])
```

```
def forcast_Horizon(self, Ticker):

    print(' ')
    print('Adding the future day close price as a target column for Forcast Horizon of ' + str(self.forcast_horz))
    #Adding the future day close price as a target column which needs to be predicted using Supervised Machine learning models
    self.Stock['Close_forcast'] = self.Stock.Close.shift(-self.forcast_horz)
    self.Stock = self.Stock.rename(columns={'Close': 'Close(t)'})
    self.Stock = self.Stock.dropna()


def save_features(self, Ticker):
    print('Saving extracted features data in S3 Bucket....')
    self.Stock.to_csv(self.Ticker + '.csv')
    print('Extracted features shape - '+ str(self.Stock.shape))
    print(' ')
    print('Extracted features dataframe - ')
    print(self.Stock)
    return self.Stock


T = TypeVar('T')

def pipeline(self,
    value: T,
    function_pipeline: Sequence[Callable[[T], T]],
    ) -> T:

    return reduce(lambda v, f: f(v), function_pipeline, value)

def pipeline_sequence(self):

    print('Initiating Pipeline....')
    z = self.pipeline(
        value=self.Ticker,
        function_pipeline=(
            self.get_stock_data,
            self.extract_Technical_Indicators,
            self.extract_date_features,
            self.get_IDXFunds_features,
            self.forcast_Horizon,
            self.save_features
                )
            )

    print(f'z={z}')
```

```
Initiating Pipeline....
Loading Historical Price data for GS....
{'1. Information': 'Daily Prices (open, high, low, close) and Volumes', '2. Symbol': 'GS', '3. Last Refreshed': '2020-04-17', '4.
Output Size': 'Full size', '5. Time Zone': 'US/Eastern'}
```


```
Feature extraction of technical Indicators....
Boolinger bands..
Lagged Price data for previous days....
Simple Moving Average....
Exponential Moving Average....
Commodity Channel Index....
Stocastic %K ....
Force index....
```





In previous step we made a feature extraction pipeline. Next step is to make a machine learning model pipeline which will train every model in a single go.

This is a snapshot of different models getting trained with all the stocks data we prepared. A detailed python notebook is added with this report which will help a user to understand the pipeline and model training.

[3]:
```
Stocks = ['NKE', 'IBM', 'KO', 'GS', 'JNJ', 'NVDA']
models = ['Linear Regression', 'Random Forest', 'XGBoost', 'LSTM']
training = Stock_Prediction_Modeling(Stocks, models)
training.pipeline_sequence()
```

```
Initiating Pipeline for Stock Ticker ----  NKE
Loading Historical Price data for NKE....
Index(['Close', 'Diff', 'High-low', 'QQQ_Close', 'SnP_Close', 'DJIA_Close',
       'ATR', 'RSI', 'MA50', 'EMA200', 'Upper_Band'],
      dtype='object')
Preparing Lagged Features for Stock, Index Funds.....
Removing NAN rows -  26
Index(['Close', 'ATR', 'RSI', 'MA50', 'EMA200', 'Upper_Band', 'Close(t-1)',
       'Close(t-2)', 'Close(t-3)', 'Close(t-4)', 'Close(t-5)', 'Close(t-6)',
       'Close(t-7)', 'Close(t-8)', 'Close(t-9)', 'Close(t-10)', 'Close(t-11)',
       'Close(t-12)', 'Close(t-13)', 'Close(t-14)', 'Close(t-15)',
       'Close(t-16)', 'Close(t-17)', 'Close(t-18)', 'Close(t-19)',
       'Close(t-20)', 'Close(t-21)', 'Close(t-22)', 'Close(t-23)',
       'Close(t-24)', 'Close(t-25)', 'QQQ_Close(t-1)', 'SnP_Close(t-1)',
       'DJIA_Close(t-1)', 'QQQ_Close(t-2)', 'SnP_Close(t-2)',
       'DJIA_Close(t-2)', 'QQQ_Close(t-3)', 'SnP_Close(t-3)',
       'DJIA_Close(t-3)', 'QQQ_Close(t-4)', 'SnP_Close(t-4)',
       'DJIA_Close(t-4)', 'QQQ_Close(t-5)', 'SnP_Close(t-5)',
       'DJIA_Close(t-5)', 'QQQ_Close(t-6)', 'SnP_Close(t-6)',
       'DJIA_Close(t-6)', 'QQQ_Close(t-7)', 'SnP_Close(t-7)',
       'DJIA_Close(t-7)', 'QQQ_Close(t-8)', 'SnP_Close(t-8)',
       'DJIA_Close(t-8)', 'QQQ_Close(t-9)', 'SnP_Close(t-9)',
       'DJIA_Close(t-9)', 'QQQ_Close(t-10)', 'SnP_Close(t-10)',
       'DJIA_Close(t-10)', 'Diff(t-1)', 'High-low(t-1)', 'Diff(t-2)',
       'High-low(t-2)', 'Diff(t-3)', 'High-low(t-3)', 'Diff(t-4)',
       'High-low(t-4)', 'Diff(t-5)', 'High-low(t-5)'],
      dtype='object')
(2805, 71)
```

[5]:
```
Metrics = pd.DataFrame.from_dict({(i,j): eval_metrics[i][j]
                                  for i in eval_metrics.keys()
                                  for j in eval_metrics[i].keys()},
                                 orient='index')
Metrics
```

[5]:

		Train_MAE	Train_RSq	Train_MAPE	Train_RMSE	Test_MAE	Test_RSq	Test_MAPE	Test_RMSE
NKE	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.22	1.00	0.30	0.57	0.68	0.97	0.85	0.93
	XGBoost	0.29	1.00	0.40	0.36	0.67	0.97	0.84	0.93
	LSTM	16.56	-0.08	21.40	20.75	15.18	-4.85	17.50	16.69
IBM	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.32	1.00	0.21	0.48	1.22	0.97	0.94	1.75
	XGBoost	0.31	1.00	0.20	0.40	1.44	0.96	1.09	1.90
	LSTM	24.06	0.00	16.77	29.73	22.87	-6.14	17.35	24.63
KO	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.09	1.00	0.19	0.30	0.33	0.98	0.66	0.46
	XGBoost	0.21	1.00	0.45	0.28	0.36	0.97	0.72	0.50
	LSTM	7.75	-0.11	13.87	11.38	5.18	-2.38	10.00	6.18
GS	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.49	1.00	0.32	0.69	2.33	0.96	1.17	3.08
	XGBoost	0.32	1.00	0.21	0.41	2.80	0.94	1.42	3.86
	LSTM	46.42	-0.54	35.62	54.62	14.98	-0.06	7.49	17.98
JNJ	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.15	1.00	0.17	0.24	1.24	0.90	0.91	1.71
	XGBoost	0.26	1.00	0.31	0.34	0.93	0.94	0.69	1.27
	LSTM	29.13	-0.72	42.05	34.40	15.22	-8.05	11.07	16.10
NVDA	Linear Regression	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00
	Random Forest	0.25	1.00	0.48	0.61	16.75	0.72	10.68	20.00
	XGBoost	0.20	1.00	0.88	0.28	14.05	0.81	8.89	16.33
	LSTM	78.87	-0.59	449.12	81.48	53.32	-1.58	25.14	66.59

Portfolio Allocation and Optimization: Monte Carlo Simulation and Optimization Algorithm:

Until now we were focused more on analyzing and forecasting Stock prices for Individual Stocks. A trader or a Hedge fund manager can optimize their Trading strategies based on the forecast Buy/Sell signal.
We will now act as a Portfolio Manager and try to build a Trading Strategy using Sharpe Ratio and Optimization to demonstrate how a Portfolio can be optimized for higher gains.

Sharpe Ratio

The Sharpe ratio measures the performance of an investment compared to a risk-free asset, after adjusting for its risk. It is defined as the difference between the returns of the investment and the risk-free return, divided by the standard deviation of the investment.

Let take the starting Portfolio Allocations consisting of these 4 stocks as -

- 25% in MCD
- 10% in KO
- 30% in JNJ
- 35% in GS

[32]:
```
for stock_df, allo in zip((mcd, ko, jnj, gs),[.25,.1,.3,.35]):
    stock_df['Allocation'] = stock_df['Normed Return']*allo
```

[33]:
```
mcd.head()
```

[33]:

Date	Close	Normed Return	Allocation
2018-09-12	164.74	1.000000	0.250000
2018-09-13	162.40	0.985796	0.246449
2018-09-14	160.84	0.976326	0.244082
2018-09-17	158.14	0.959937	0.239984
2018-09-18	157.77	0.957691	0.239423

Assuming the starting portfolio Position of 1 million $, lets look at how the value of our position changes in each stock

[34]:
```
# value of each position
for stock_df in (mcd, ko, jnj, gs):
    stock_df['Position Value'] = stock_df['Allocation']*1000000
```

[36]:
```
gs.head(10)
```

[36]:

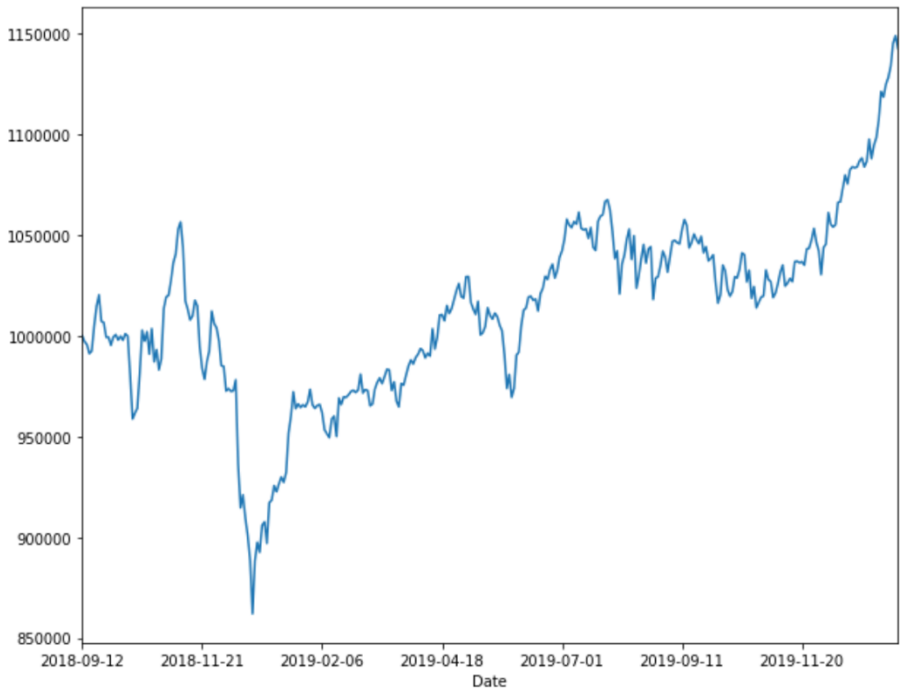
Date	Close	Normed Return	Allocation	Position Value
2018-09-12	228.15	1.000000	0.350000	350000.000000
2018-09-13	228.33	1.000789	0.350276	350276.134122
2018-09-14	229.24	1.004778	0.351672	351672.145518
2018-09-17	227.89	0.998860	0.349601	349601.139601
2018-09-18	228.89	1.003243	0.351135	351135.218058
2018-09-19	235.58	1.032566	0.361398	361398.202937
2018-09-20	237.40	1.040544	0.364190	364190.225729
2018-09-21	235.34	1.031514	0.361030	361030.024107
2018-09-24	232.90	1.020820	0.357287	357286.872671
2018-09-25	232.50	1.019066	0.356673	356673.241289

You can see how the value in increasing. Lets create a single dataframe for all of the 4 stocks

we can see day-by-day how our positions and portfolio value is changing.

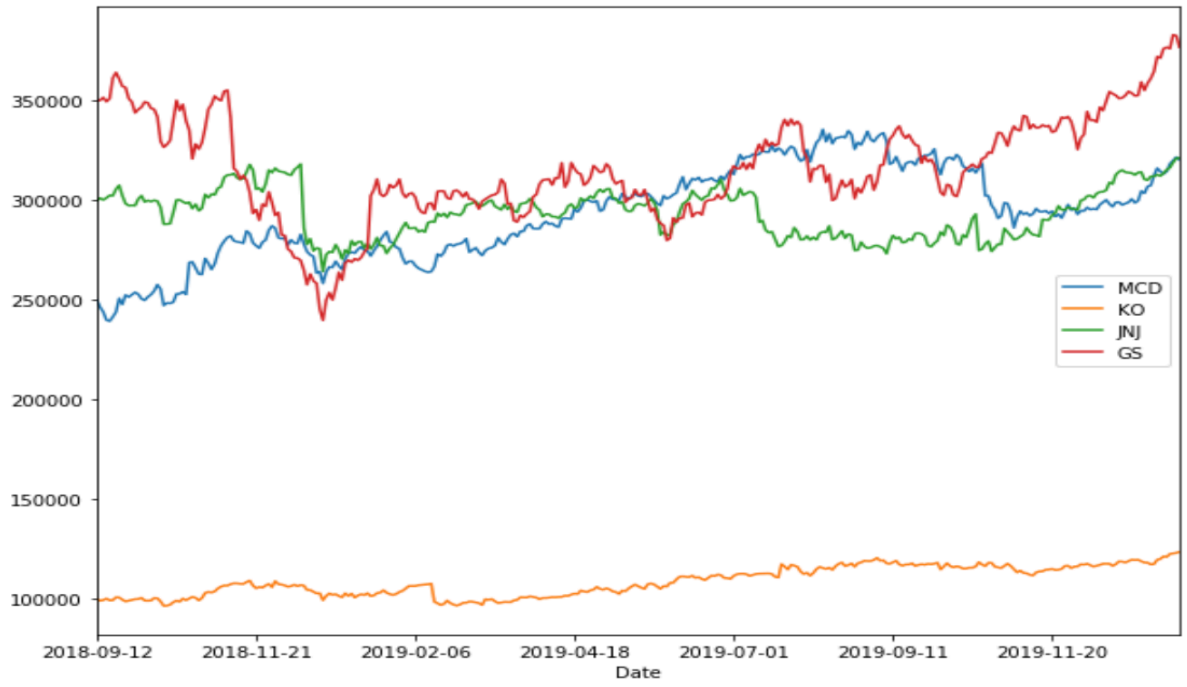
[39]:
```
# plot our portfolio
import matplotlib.pyplot as plt
%matplotlib inline
portfolio_val['Total'].plot(figsize=(10,8))
```

[39]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x23ef7f184e0>
```



We have made 150k for the year

Let's also look at individual contributions of each stock in our portfolio

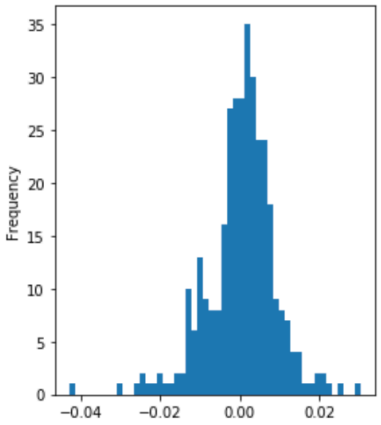


Let's move towards implementing MCMC and Optimization:

[41]:
```
# Daily Return
portfolio_val['Daily Return'] = portfolio_val['Total'].pct_change(1)
```

[43]:
```
# average daily return
portfolio_val['Daily Return'].mean()

# standard deviation
portfolio_val['Daily Return'].std()

# plot histogram of daily returns
portfolio_val['Daily Return'].plot(kind='hist', bins=50, figsize=(4,5))
```

[43]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x23ef83fbc50>
```



Calculate Sharpe Ratio

The Sharpe Ratio is the mean (portfolio return - the risk free rate) % standard deviation.

[47]:
```
sharpe_ratio = portfolio_val['Daily Return'].mean() / portfolio_val['Daily Return'].std()
ASR = (252**0.5) * sharpe_ratio
ASR
```

[47]:
```
0.7923843343475722
```

Annualized Share Ratio is 0.79

Optimization using Monte Carlo Simulation:

We will check a bunch of random allocations and analyze which one has the best Sharpe Ratio.
This process of randomly guessing is known as a Monte Carlo Simulation.

We will randomly assign weights to our stocks in the portfolio using mcmc and then calculate the average daily return & SD (Standard deviation) of return. Then we can calculate the Sharpe Ratio for many randomly selected allocations.

We will further use Optimization Algorithm to minimize for this.
Minimization is a similar concept to optimization - let's say we have a simple equation y = x2 - the idea is we're trying to figure out what value of x will minimize y, in this example 0.
This idea of a minimizer will allow us to build an optimizer.

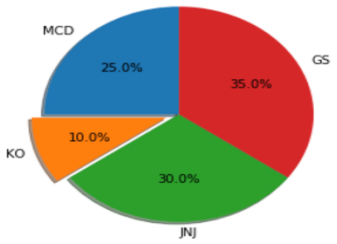
[63]:
```
num_ports = 8000
all_weights = np.zeros((num_ports, len(stocks.columns)))
ret_arr = np.zeros(num_ports)
vol_arr = np.zeros(num_ports)
sharpe_arr = np.zeros(num_ports)

for ind in range(num_ports):
    # weights
    weights = np.array(np.random.random(4))
    weights = weights/np.sum(weights)
    # save the weights
    all_weights[ind,:] = weights
    # expected return
    ret_arr[ind] = np.sum((log_return.mean()*weights)*252)

    # expected volatility
    vol_arr[ind] = np.sqrt(np.dot(weights.T,np.dot(log_return.cov()*252, weights)))

    # Sharpe Ratio
    sharpe_arr[ind] = ret_arr[ind]/vol_arr[ind]
```

[64]:
```
sharpe_arr.max()
```

[64]:
```
1.232583481117841
```

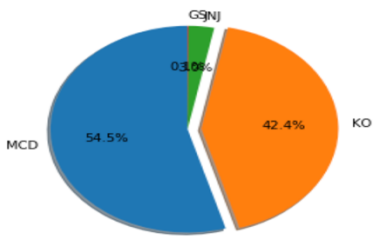
Initial Allocation

[68]:
```
import matplotlib.pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = 'MCD', 'KO', 'JNJ', 'GS'
sizes = [25, 10, 30, 35]
explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()
```

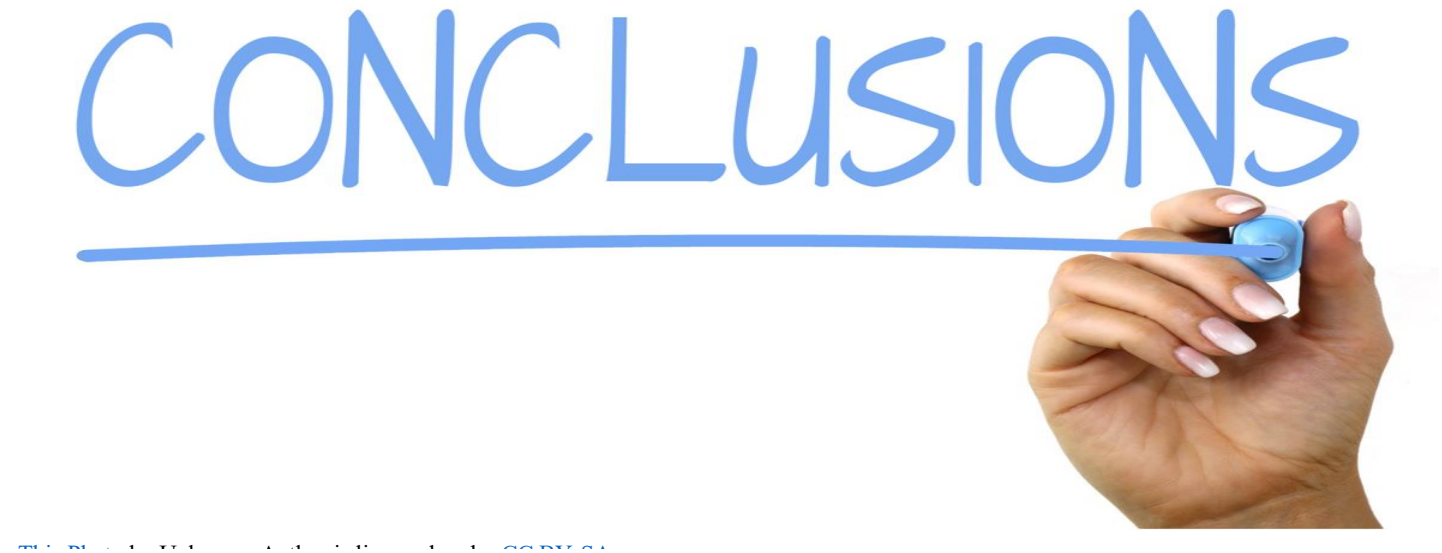


After MCMC

[69]:
```
import matplotlib.pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = 'MCD', 'KO', 'JNJ', 'GS'
sizes = [54, 42, 3, 0.1]
explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()
```



CONCLUSION:



Random Forest, XG Boost and Linear regression seems to work very well on such a long data as well. We have taken around 12-15 years of historical data which was divided into Training, Validation and Testing.

These algorithms are able to catch the trend and move close to the Actual closing price in the right direction with a very low Mean Absolute error (MAE). The Evaluation Metrics is above to see how each algorithm performed on each stock. They worked so well because of the Feature Extraction we did to extract around 60 features including lagged Index funds prices, Technical Indicators like Exponential Moving Average, RSI, ADR, Willam's R, bollinger bands and many more.

LSTMs did not perform well on this as expected, as the data is not using any sequence of previous many time steps. and LSTM will work better on time series data once we provide it with previous 30-60 days of lookback data for every prediction.

We then checked on the LSTM Lagged 60 – Day and the model performed really well there.
We have implemented Monte carlo Simulation technique to randomly take a sample and simulate it to find the best sharpe ratio for a given portfolio.
Then we moved to a better appraoch using Optimization Algorithm by Minimize function in Spicy library.

We get around same Sharpe ratio for both which improves on the initial portfolio allocation we started off with.

We can form more trading strategies based on other methods too like Pairs trading, Butterfly spread, Bull-bear spread etc. But we just wanted to implement a Trading strategy and see how we can optimize a portfolio.

REFERENCES:

- https://www.kaggle.com/c/two-sigma-financial-news
- https://en.wikipedia.org/wiki/Stock_market_prediction
- https://towardsdatascience.com/simple-linear-regression-in-four-lines-of-code-d690fe4dba84
- https://towardsdatascience.com/https-medium-com-vishalmorde-xgboost-algorithm-long-she-may-rein-edd9f99be63d
- https://towardsdatascience.com/understanding-random-forest-58381e0602d2
- https://towardsdatascience.com/random-forest-and-its-implementation-71824ced454f
- https://towardsdatascience.com/long-short-term-memory-and-gated-recurrent-units-explained-eli5-way-eff3d44f50dd
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://www.mlq.ai/python-for-finance-portfolio-optimization/
- https://www.streamlit.io/gallery
-

MIT License