Application of Machine Learning Techniques for Stock Market Prediction

Introduction

Predicting how the stock market will perform is one of the most difficult things to do. There are so many factors involved in the prediction – physical factors vs. psychological, rational and irrational behaviour, etc. All these aspects combine to make share prices volatile and very difficult to predict with a high degree of accuracy.

Can we use machine learning as a game changer in this domain? Using features like the latest announcements about an organization, their quarterly revenue results, etc., machine learning techniques have the potential to unearth patterns and insights we didn't see before, and these can be used to make unerringly accurate predictions.

We will work with historical data about the stock prices of a publicly listed company. We will implement a mix of machine learning algorithms to predict the future stock price of this company, starting with simple algorithms like averaging and linear regression, and then moving on to advanced techniques like Auto ARIMA.

The core idea is to showcase how these algorithms are implemented, and briefly describing the underlying techniques

Table of Contents

Understanding the Problem Statement

We'll dive into the implementation part of this article soon, but first it's important to establish what we're aiming to solve. Broadly, stock market analysis is divided into two parts – Fundamental Analysis and Technical Analysis.

Fundamental Analysis involves analysing the company's future profitability on the basis of its current business environment and financial performance.

Technical Analysis, on the other hand, includes reading the charts and using statistical figures to identify the trends in the stock market.

Our focus will be on the technical analysis part. We'll be using datasets from Quandl (can find historical data for all public stocks) and for this particular project, we've used the data for 'AMAZON'.

We will first load the dataset and define the target variable for the problem:

```
#import packages
import pandas as pd
import numpy as np

#to plot within notebook
import matplotlib.pyplot as plt
%matplotlib inline

#setting figure size
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 20,10

#for normalizing data
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))

#read the file
df = pd.read_csv('NASDAQ-AMAZON.csv')

#print the head
df.head()
```

	Date	Open	High	Low	Close	Volume
2	30-11-2018	1679.5	1696	1666.5	1690.17	5761800
3	03-12-2018	1769.46	1778.34	1730	1772.36	6862300
4	04-12-2018	1756	1770.34	1665	1668.4	8694500
5	06-12-2018	1614.87	1701.05	1609.85	1699.19	8789400
6	07-12-2018	1705.07	1718.93	1625.46	1629.13	7576100
7	10-12-2018	1623.84	1657.99	1590.87	1641.03	7487500

There are multiple variables in the dataset – date, open, high, low, last close and total trade quantity in volume. The columns *Open* and *Close* represent the starting and final price at which the stock is traded on a particular day. *High*, *Low* and *Last* represent the maximum, minimum, and last price of the share for the day. *Volume* is the number of shares bought or sold in the day.

The profit or loss calculation is usually determined by the closing price of a stock for the day, hence we will consider the closing price as the target variable. Let's plot the target variable to understand how it's shaping up in our data:

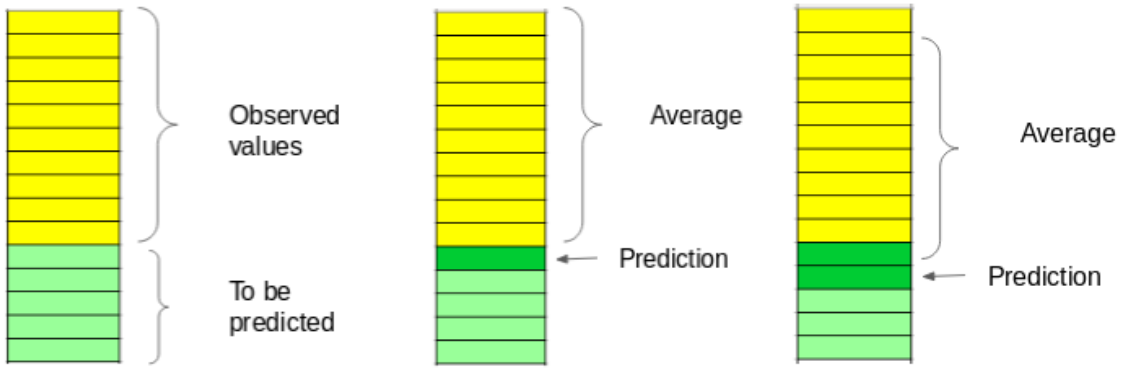
```
#setting index as date
df['Date'] = pd.to_datetime(df.Date,format='%Y-%m-%d')
df.index = df['Date']

#plot
plt.figure(figsize=(16,8))
plt.plot(df['Close'], label='Close Price history')
```

We will explore these variables and use different techniques to predict the daily closing price of the stock.

Moving Average

Introduction

'Average' is easily one of the most common things we use in our day-to-day lives. For instance, calculating the average marks to determine overall performance, or finding the average temperature of the past few days to get an idea about today's temperature – these all are routine tasks we do on a regular basis. So this is a good starting point to use on our dataset for making predictions.
The predicted closing price for each day will be the average of a set of previously observed values. Instead of using the simple average, we will be using the moving average technique which uses the latest set of values for each prediction. In other words, for each subsequent step, the predicted values are taken into consideration while removing the oldest observed value from the set. Here is a simple figure that will help you understand this with more clarity.



We will implement this technique on our dataset. The first step is to create a dataframe that contains only the *Date* and *Close* price columns, then split it into train and validation sets to verify our predictions.

Implementation

```
#creating dataframe with date and the target variable
data = df.sort_index(ascending=True, axis=0)
new_data = pd.DataFrame(index=range(0,len(df)),columns=['Date', 'Close'])

for i in range(0,len(data)):
    new_data['Date'][i] = data['Date'][i]
    new_data['Close'][i] = data['Close'][i]
```

While splitting the data into train and validation, we cannot use random splitting since that will destroy the time component. So here I have set the last year's data into validation and the 4 years' data before that into train.

```
#splitting into train and validation
train = new_data[:987]
valid = new_data[987:]
new_data.shape, train.shape, valid.shape
((1235, 2), (987, 2), (248, 2))
train['Date'].min(), train['Date'].max(), valid['Date'].min(), valid['Date'].max()

(Timestamp('2014-12-05 00:00:00'),
Timestamp('2017-10-06 00:00:00'),
Timestamp('2017-12-08 00:00:00'),
Timestamp('2018-12-07 00:00:00'))
```

The next step is to create predictions for the validation set and check the RMSE using the actual values.

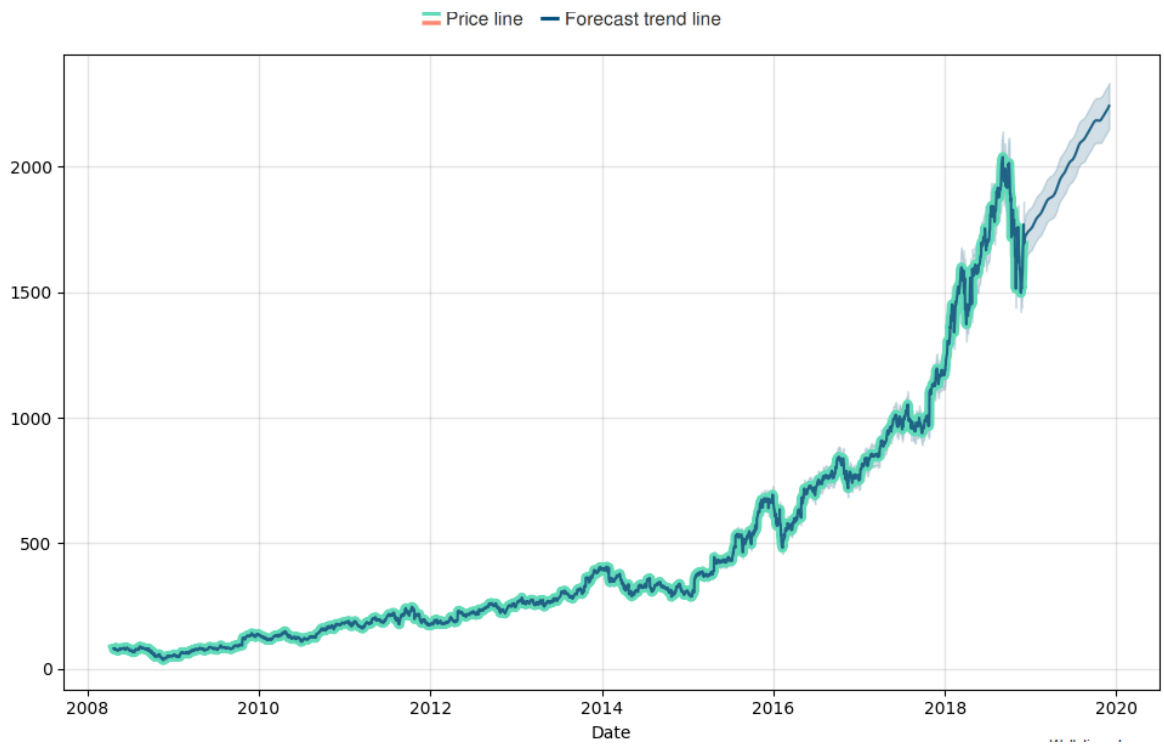
```
#make predictions
preds = []
for i in range(0,248):
   a = train['Close'][len(train)-248+i:].sum() + sum(preds)
   b = a/248
   preds.append(b)
```

Results

```
#calculate rmse
rms = np.sqrt(np.mean(np.power((np.array(valid['Close'])-preds),2)))
rms = 104.51415465984348
```

Just checking the RMSE does not help us in understanding how the model performed. Let's visualize this to get a more intuitive understanding. So here is a plot of the predicted values along with the actual values.

```
#plot
valid['Predictions'] = 0
valid['Predictions'] = preds
plt.plot(train['Close'])
plt.plot(valid[['Close', 'Predictions']])
```



Inference

The RMSE value is close to 105 but the results are not very promising (as you can gather from the plot). The predicted values are of the same range as the observed values in the train set (there is an increasing trend initially and then a slow decrease).

In the next section, we will look at two commonly used machine learning techniques – Linear Regression and kNN, and see how they perform on our stock market data.

Linear Regression

Introduction

The most basic machine learning algorithm that can be implemented on this data is linear regression. The linear regression model returns an equation that determines the relationship between the independent variables and the dependent variable.
The equation for linear regression can be written as:

$$Y = \theta_1 X_1 + \theta_2 X_2 + \dots \theta_n X_n$$

Here, x_1, x_2,….x_n represent the independent variables while the coefficients θ_1, θ_2, …. θ_n represent the weights. You can refer to the following article to study linear regression in more detail:

For our problem statement, we do not have a set of independent variables. We have only the dates instead. Let us use the date column to extract features like – day, month, year, mon/fri etc. and then fit a linear regression model.

Implementation

We will first sort the dataset in ascending order and then create a separate dataset so that any new feature created does not affect the original data.

```
#setting index as date values
df['Date'] = pd.to_datetime(df.Date,format='%Y-%m-%d')
df.index = df['Date']

#sorting
data = df.sort_index(ascending=True, axis=0)

#creating a separate dataset
new_data = pd.DataFrame(index=range(0,len(df)),columns=['Date', 'Close'])

for i in range(0,len(data)):
    new_data['Date'][i] = data['Date'][i]
    new_data['Close'][i] = data['Close'][i]

#create features
from fastai.structured import  add_datepart
add_datepart(new_data, 'Date')
new_data.drop('Elapsed', axis=1, inplace=True)  #elapsed will be the time stamp
```

This creates features such as:
‘Year’(1.5), ‘Month’(1.5), ‘Week’(0.5), ‘Day’(1.5), ‘Is_month_end’(1), ‘Is_month_start’(1), ‘Is_quarter_end’(2), ‘Is_quarter_start(2)’. We assigned the weights based on intuition

because weren't able to calculate them because of the lack of time. We chose numbers between 0 and 2. No negative number because none of them exactly negatively impact the data.
We gave quarter end and quarter start the highest possible weight of 2. We decided to do so because of how important those periods of time are. We gave week only a 0.5 because that doesn't really help because we have other features such as month start or end which give us better information and so on.

Note: I have used add_datepart from fastai library. Otherwise, this feature can be implemented using simple for loops in python.

Apart from this, we can add our own set of features that we believe would be relevant for the predictions. For instance, my hypothesis is that the first and last days of the week could potentially affect the closing price of the stock far more than the other days. So I have created a feature that identifies whether a given day is Monday / Friday or Tuesday / Wednesday / Thursday. This can be done using the following lines of code:

```
new_data['mon_fri'] = 0

for i in range(0,len(new_data)):
    if (new_data['Dayofweek'][i] == 0 or new_data['Dayofweek'][i] == 4):
        new_data['mon_fri'][i] = 1
    else:
        new_data['mon_fri'][i] = 0
```

If the day of week is equal to 0 or 4, the column value will be 1, otherwise 0. Similarly, we can create multiple features.

We will now split the data into train and validation sets to check the performance of the model.

```
#split into train and validation
train = new_data[:987]
valid = new_data[987:]

x_train = train.drop('Close', axis=1)
y_train = train['Close']
x_valid = valid.drop('Close', axis=1)
y_valid = valid['Close']

#implement linear regression
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(x_train,y_train)
```

Results

```
#make predictions and find the rmse
preds = model.predict(x_valid)
rms=np.sqrt(np.mean(np.power((np.array(y_valid)-np.array(preds)),2)))
rms = 121.16291596523156
```

The RMSE value is higher than the previous technique, which clearly shows that linear regression has performed poorly. Let's look at the plot and understand why linear regression has not done well:

```
#plot
valid['Predictions'] = 0
valid['Predictions'] = preds

valid.index = new_data[987:].index
train.index = new_data[:987].index

plt.plot(train['Close'])
plt.plot(valid[['Close', 'Predictions']])
```

Inference

Linear regression is a simple technique and quite easy to interpret, but there are a few obvious disadvantages. One problem in using regression algorithms is that the model overfits to the date and month column. Instead of taking into account the previous values from the point of

prediction, the model will consider the value from the same *date* a month ago, or the same *date/month* a year ago.

As seen from the plot above, for January 2017 and January 2018, there was an increase in the stock price. The model has predicted the same for January 2019. A linear regression technique can perform well for such problems where the independent features are useful for determining the target value.

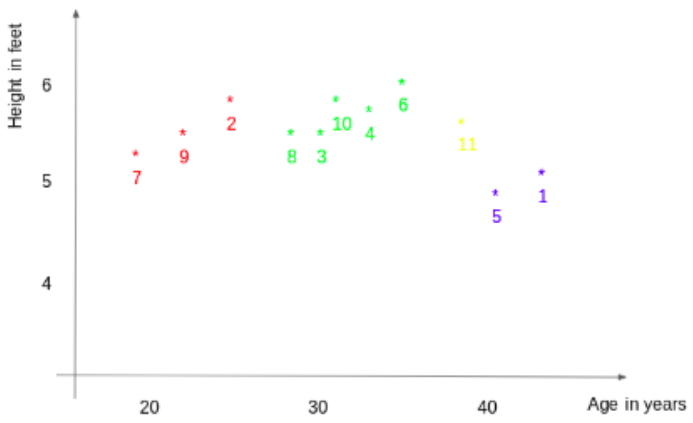
k-Nearest Neighbours

Introduction

Another interesting ML algorithm that one can use here is kNN (k nearest neighbours). Based on the independent variables, kNN finds the similarity between new data points and old data points. This can be better explained with a simple example.

Consider the height and age for 11 people. On the basis of given features ('Age' and 'Height'), the table can be represented in a graphical format as shown below:

ID	Age	Height	Weight
1	45	5	77
2	26	5.11	47
3	30	5.6	55
4	34	5.9	59
5	40	4.8	72
6	36	5.8	60
7	19	5.3	40
8	28	5.8	60
9	23	5.5	45
10	32	5.6	58
11	38	5.5	?



To determine the weight for ID #11, kNN considers the weight of the nearest neighbours of this ID. The weight of ID #11 is predicted to be the average of its neighbours. If we consider three neighbours (k=3) for now, the weight for ID #11 would be = (77+72+60)/3 = 69.66 kg.

ID	Height	Age	Weight
1	5	45	77
5	4.8	40	72
6	5.8	36	60

Implementation

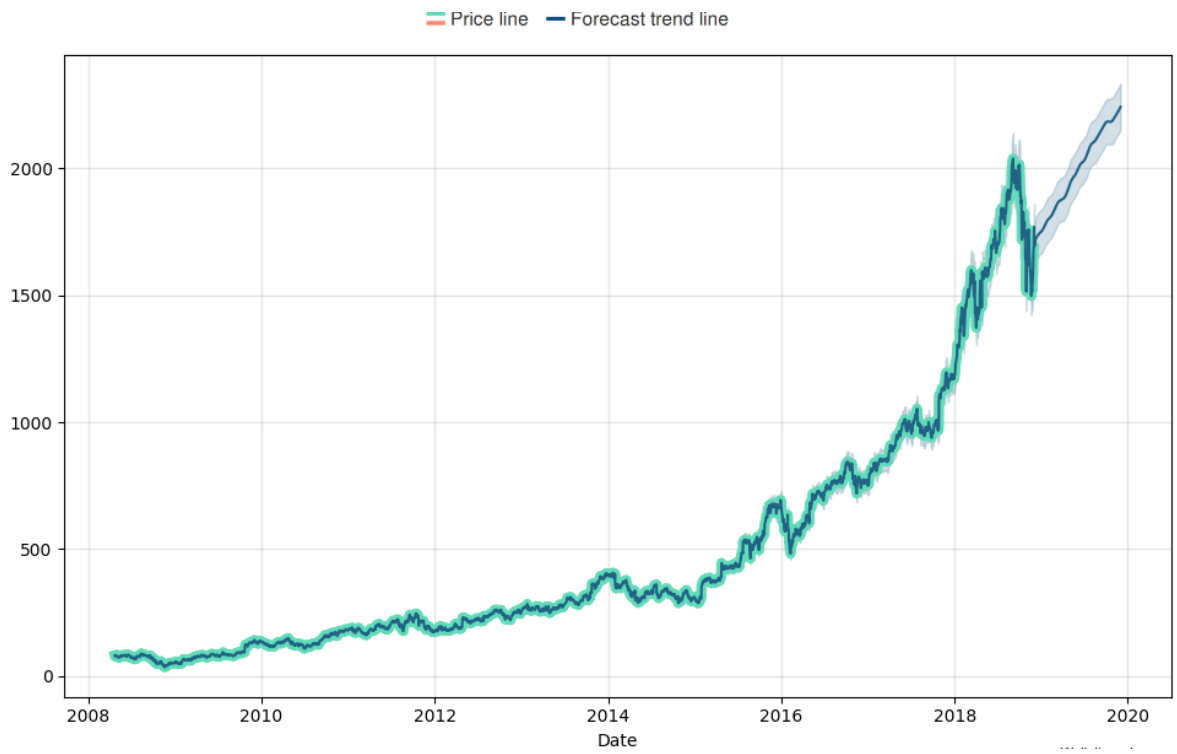
```
#importing libraries
from sklearn import neighbors
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
```
Using the same train and validation set from the last section:

```
#scaling data

x_train_scaled = scaler.fit_transform(x_train)
x_train = pd.DataFrame(x_train_scaled)
x_valid_scaled = scaler.fit_transform(x_valid)
x_valid = pd.DataFrame(x_valid_scaled)

#using gridsearch to find the best parameter
params = {'n_neighbors':[2,3,4,5,6,7,8,9]}
knn = neighbors.KNeighborsRegressor()
model = GridSearchCV(knn, params, cv=5)

#fit the model and make predictions
model.fit(x_train,y_train)

preds = model.predict(x_valid)
```
Results

```
#rmse
rms=np.sqrt(np.mean(np.power((np.array(y_valid)-np.array(preds)),2)))
rms = 115.17086550026721
```

There is not a huge difference in the RMSE value, but a plot for the predicted and actual values should provide a more clear understanding.

```
#plot
valid['Predictions'] = 0
valid['Predictions'] = preds
plt.plot(valid[['Close', 'Predictions']])
plt.plot(train['Close'])
```



Inference

The RMSE value is almost similar to the linear regression model and the plot shows the same pattern. Like linear regression, kNN also identified a price increase in January 2019 since that has been the pattern for the past years. We can safely say that regression algorithms have not performed well on this dataset.

Let's go ahead and look at some time series forecasting techniques to find out how they perform when faced with this stock prices prediction challenge.

ARIMA (Auto Regressive Integrated Moving Average)

Introduction

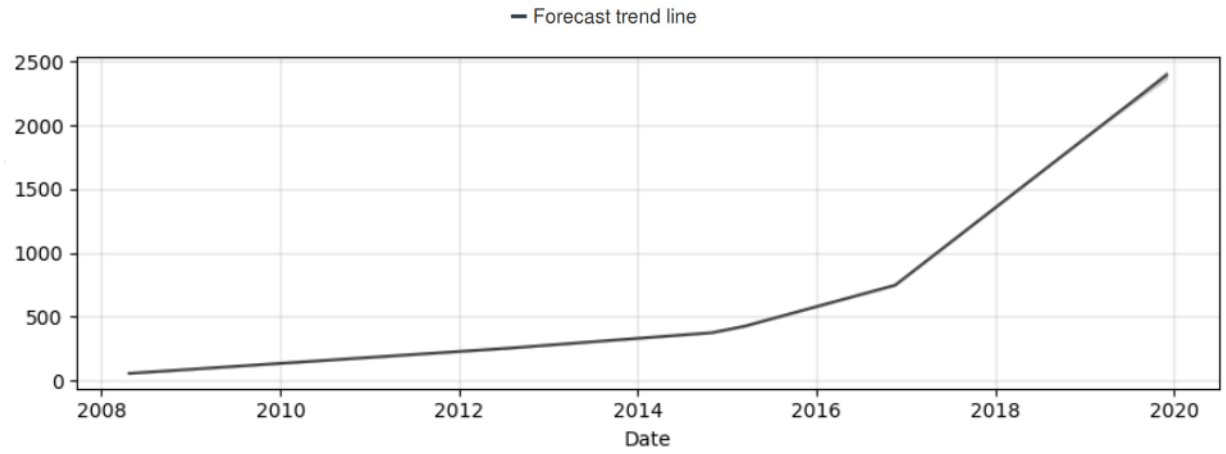
ARIMA is a very popular statistical method for time series forecasting and is a generalization of an autoregressive moving average model. ARIMA models take into account the past values to predict the future values. There are three important parameters in ARIMA:
p (past values used for forecasting the next value)
q (past forecast errors used to predict the future values)
d (order of differencing)
Parameter tuning for ARIMA consumes a lot of time. So we will use auto ARIMA which automatically selects the best combination of (p,q,d) that provides the least error.

Implementation

```
from pyramid.arima import auto_arima

data = df.sort_index(ascending=True, axis=0)

train = data[:987]
valid = data[987:]

training = train['Close']
validation = valid['Close']

model = auto_arima(training, start_p=1, start_q=1,max_p=3, max_q=3, m=12,start_P=0, seasonal=True,d=1, D=1, trace=True,error_action='ignore',suppress_warnings=True)
model.fit(training)

forecast = model.predict(n_periods=248)
forecast = pd.DataFrame(forecast,index = valid.index,columns=['Prediction'])
```

Results

```
rms=np.sqrt(np.mean(np.power((np.array(valid['Close'])-np.array(forecast['Prediction'])),2)))
rms = 44.954584993246954

#plot
plt.plot(train['Close'])
plt.plot(valid['Close'])
plt.plot(forecast['Prediction'])
```



Inference

As we saw earlier, an auto ARIMA model uses past data to understand the pattern in the time series. Using these values, the model captured an increasing trend in the series. Although the predictions using this technique are far better than that of the previously implemented machine learning models, these predictions are still not close to the real values.

As its evident from the plot, the model has captured a trend in the series, but does not focus on the seasonal part. In the next section, we will implement a time series model that takes both trend and seasonality of a series into account.

Prophet

Introduction

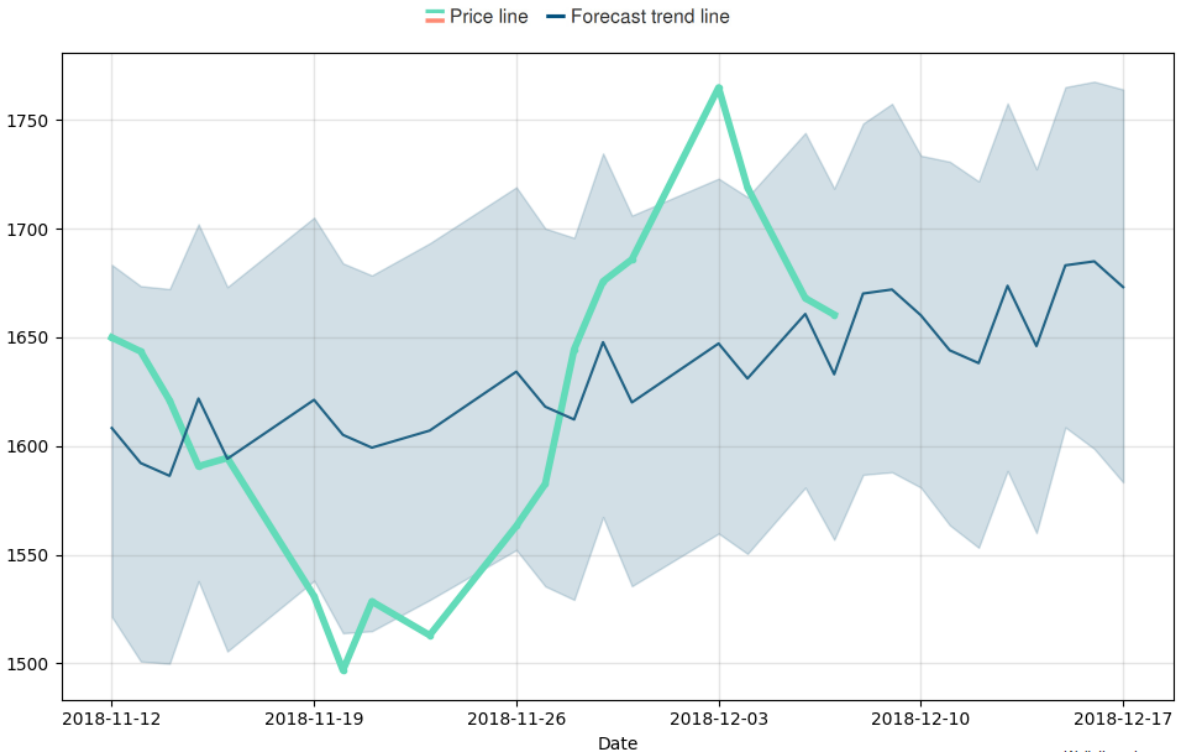
There are a number of time series techniques that can be implemented on the stock prediction dataset, but most of these techniques require a lot of data pre-processing before fitting the model. Prophet, designed and pioneered by Facebook, is a time series forecasting library that requires no data pre-processing and is extremely simple to implement. The input for Prophet is a dataframe with two columns: date and target (ds and y).

Prophet tries to capture the seasonality in the past data and works well with large datasets.

Implementation

```
#importing prophet
from fbprophet import Prophet

#creating dataframe
new_data = pd.DataFrame(index=range(0,len(df)),columns=['Date', 'Close'])

for i in range(0,len(data)):
   new_data['Date'][i] = data['Date'][i]
   new_data['Close'][i] = data['Close'][i]

new_data['Date'] = pd.to_datetime(new_data.Date,format='%Y-%m-%d')
new_data.index = new_data['Date']

#preparing data
new_data.rename(columns={'Close': 'y', 'Date': 'ds'}, inplace=True)

#train and validation
train = new_data[:987]
valid = new_data[987:]

#fit the model
model = Prophet()
model.fit(train)

#predictions
close_prices = model.make_future_dataframe(periods=len(valid))
forecast = model.predict(close_prices)
Results

#rmse
forecast_valid = forecast['yhat'][987:]
```

```
rms=np.sqrt(np.mean(np.power((np.array(valid['y'])-np.array(forecast_valid)),2)))
rms = 57.494461930575149

#plot
valid['Predictions'] = 0
valid['Predictions'] = forecast_valid.values

plt.plot(train['y'])
plt.plot(valid[['y', 'Predictions']])
```



Inference

Prophet (like most time series forecasting techniques) tries to capture the trend and seasonality from past data. This model usually performs well on time series datasets, but fails to live up to its reputation in this case.

As it turns out, stock prices do not have a particular trend or seasonality. It highly depends on what is currently going on in the market and thus the prices rise and fall. Hence forecasting techniques like ARIMA, SARIMA (Seasonal ARIMA) and Prophet would not show good results for this particular problem.

Conclusion

Time series forecasting is a very intriguing area with a general perception that it's a complex field, and while there is a grain of truth in there, it's not so difficult once we get the hang of the basic techniques.

The takeaway from these steps is that each of these models can outperform others on a particular dataset. Therefore it doesn't mean that one model which performs best on one type of dataset will perform the same for all others too.

We may also explore Double seasonality models from Python's forecast package that may generate an even better model resulting in a better score.

MODEL FUSION

We decided that averaging the predictions of all the models would be a good idea because every one of the classifiers has its own advantages and disadvantages and just because some of them didn't work well with this dataset doesn't mean they perform poorly on this task.

Average RMS = 88.656

We planned on dropping some of the models like KNN and moving averages because they didn't produce good results, but we ended up deciding to keep them because they were a part of the whole process.