**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:https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/10/stock11.png

Here, x1, x2,….xn represent the independent variables while the coefficients θ1, θ2, …. θn  represent the weights. You can refer to the following article to study linear regression in more detail:

* [A comprehensive beginners guide for Linear, Ridge and Lasso Regression](https://www.analyticsvidhya.com/blog/2017/06/a-comprehensive-guide-for-linear-ridge-and-lasso-regression/).

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’, ‘Month’, ‘Week’, ‘Day’, ‘Dayofweek’, ‘Dayofyear’, ‘Is\_month\_end’, ‘Is\_month\_start’, ‘Is\_quarter\_end’, ‘Is\_quarter\_start’,  ‘Is\_year\_end’, and  ‘Is\_year\_start’.

*Note: I have used add\_datepart from fastai library. If you do not have it installed, you can simply use the command****pip install fastai****. Otherwise, you can create these feature using simple for loops in python. I have shown an example below.*

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, you can create multiple features. *If you have some ideas for features that can be helpful in predicting stock price, please share in the comment section.*

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

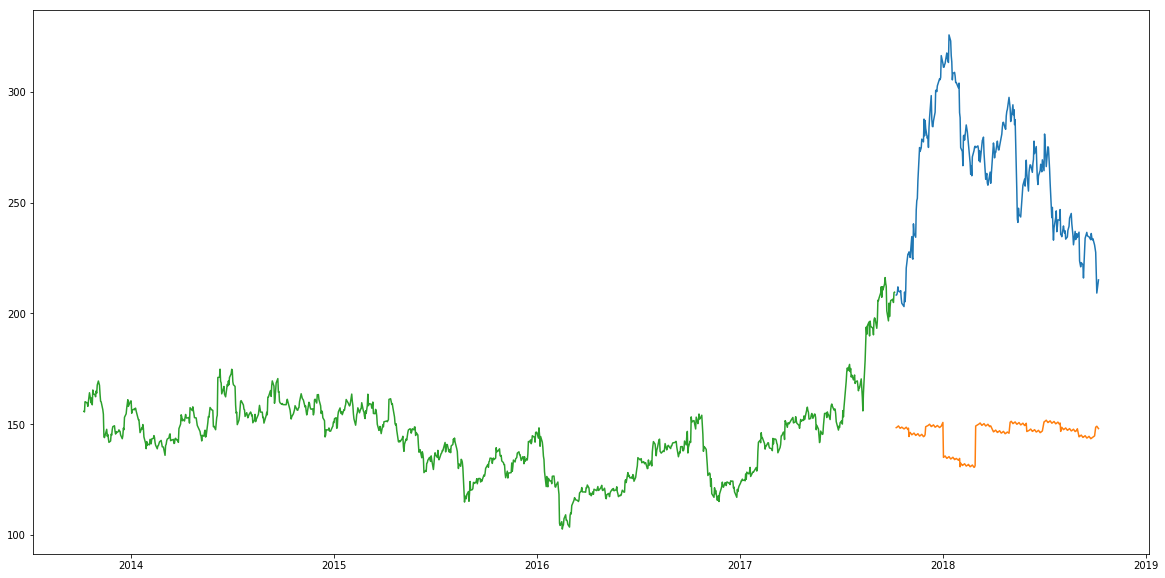
valid['Predictions'] = preds

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.

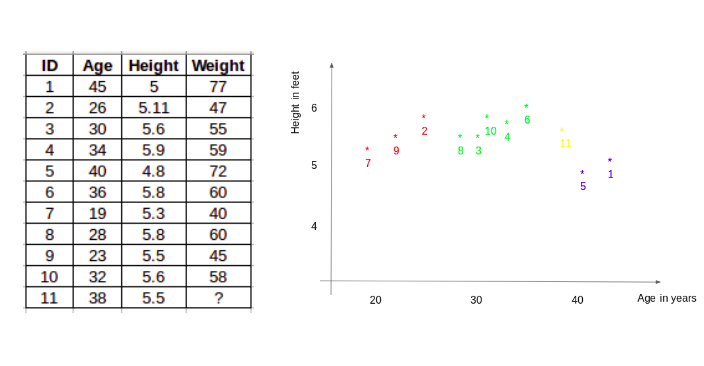
As seen from the plot above, for January 2016 and January 2017, there was a drop in the stock price. The model has predicted the same for January 2018. A linear regression technique can perform well for problems such as [Big Mart sales](https://datahack.analyticsvidhya.com/contest/practice-problem-big-mart-sales-iii/) where the independent features are useful for determining the target value.

**k-Nearest Neighbours**

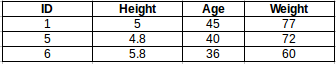
**Introduction**

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. Let me explain this 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:



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



For a detailed understanding of kNN, you can refer to the following articles:

* [Introduction to k-Nearest Neighbors: Simplified](https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/)
* [A Practical Introduction to K-Nearest Neighbors Algorithm for Regression](https://www.analyticsvidhya.com/blog/2018/08/k-nearest-neighbor-introduction-regression-python/)

**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.

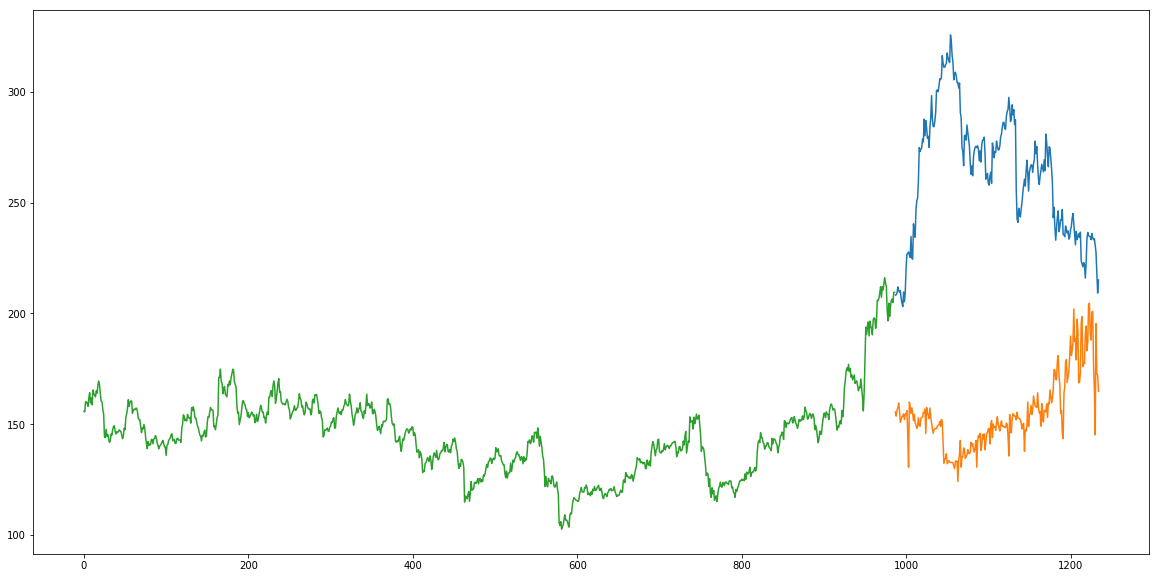
#plot

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 drop in January 2018 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.

**Auto ARIMA**

**Introduction**

ARIMA is a very popular statistical method for time series forecasting. 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. To read more about how auto ARIMA works, refer to this article:

* [Build High Performance Time Series Models using Auto ARIMA](https://www.analyticsvidhya.com/blog/2018/08/auto-arima-time-series-modeling-python-r/)

**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

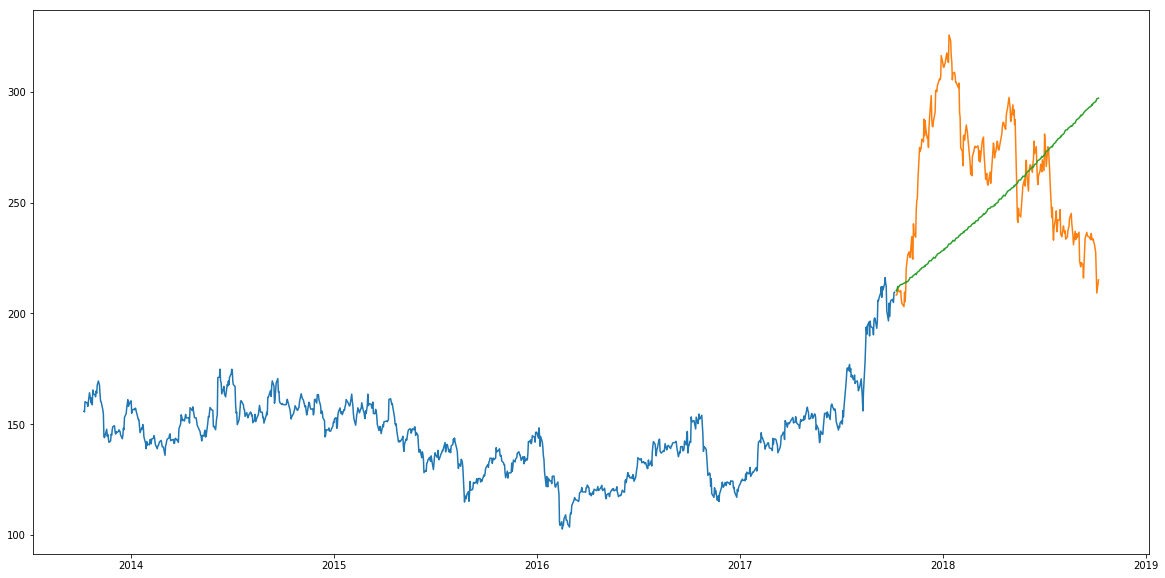
44.954584993246954

#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 preprocessing before fitting the model. Prophet, designed and pioneered by Facebook, is a time series forecasting library that requires no data preprocessing 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 when the dataset is large. Here is an interesting article that explains Prophet in a simple and intuitive manner:

* [Generate Quick and Accurate Time Series Forecasts using Facebook’s Prophet](https://www.analyticsvidhya.com/blog/2018/05/generate-accurate-forecasts-facebook-prophet-python-r/).

**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

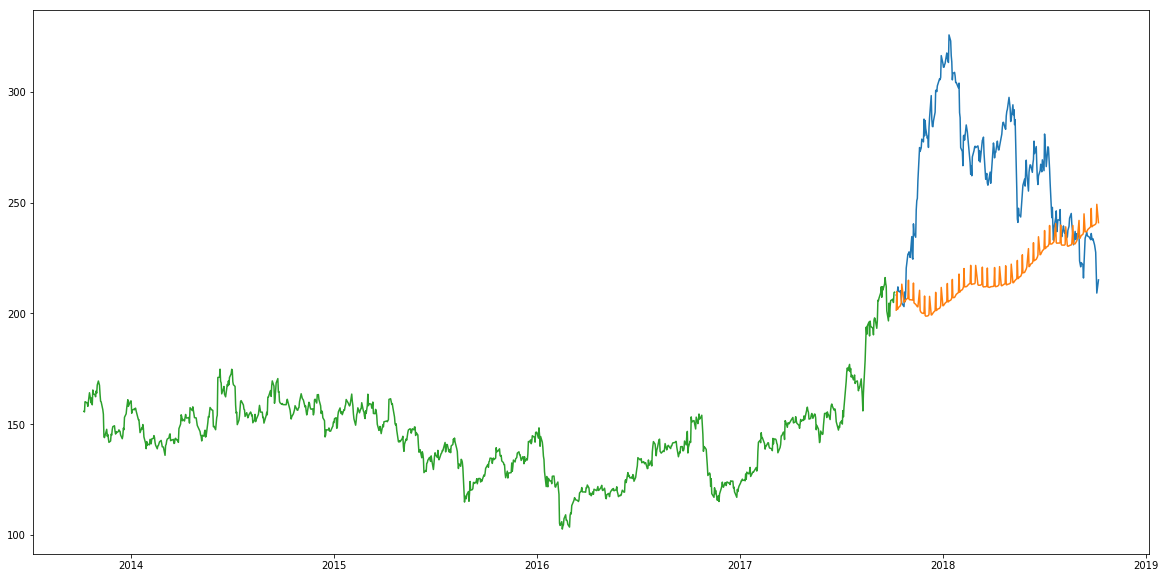
#plot

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 it’s 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 and Prophet would not show good results for this particular problem.

Let us go ahead and try another advanced technique – Long Short Term Memory (LSTM).

**Long Short Term Memory (LSTM)**

**Introduction**

LSTMs are widely used for sequence prediction problems and have proven to be extremely effective. The reason they work so well is because LSTM is able to store past information that is important, and forget the information that is not. LSTM has three gates:

* **The input gate:** The input gate adds information to the cell state
* **The forget gate:** It removes the information that is no longer required by the model
* **The output gate:**Output Gate at LSTM selects the information to be shown as output

For a more detailed understanding of LSTM and its architecture, you can go through the below article:

* [Introduction to Long Short Term Memory](https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/)

For now, let us implement LSTM as a black box and check it’s performance on our particular data.

**Implementation**

#importing required libraries

from sklearn.preprocessing import MinMaxScaler

from keras.models import Sequential

from keras.layers import Dense, Dropout, LSTM

#creating dataframe

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]

#setting index

new\_data.index = new\_data.Date

new\_data.drop('Date', axis=1, inplace=True)

#creating train and test sets

dataset = new\_data.values

train = dataset[0:987,:]

valid = dataset[987:,:]

#converting dataset into x\_train and y\_train

scaler = MinMaxScaler(feature\_range=(0, 1))

scaled\_data = scaler.fit\_transform(dataset)

x\_train, y\_train = [], []

for i in range(60,len(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

model = Sequential()

model.add(LSTM(units=50, return\_sequences=True, input\_shape=(x\_train.shape[1],1)))

model.add(LSTM(units=50))

model.add(Dense(1))

model.compile(loss='mean\_squared\_error', optimizer='adam')

model.fit(x\_train, y\_train, epochs=1, batch\_size=1, verbose=2)

#predicting 246 values, using past 60 from the train data

inputs = new\_data[len(new\_data) - len(valid) - 60:].values

inputs = inputs.reshape(-1,1)

inputs = scaler.transform(inputs)

X\_test = []

for i in range(60,inputs.shape[0]):

X\_test.append(inputs[i-60:i,0])

X\_test = np.array(X\_test)

X\_test = np.reshape(X\_test, (X\_test.shape[0],X\_test.shape[1],1))

closing\_price = model.predict(X\_test)

closing\_price = scaler.inverse\_transform(closing\_price)

**Results**

rms=np.sqrt(np.mean(np.power((valid-closing\_price),2)))

rms

11.772259608962642

#for plotting

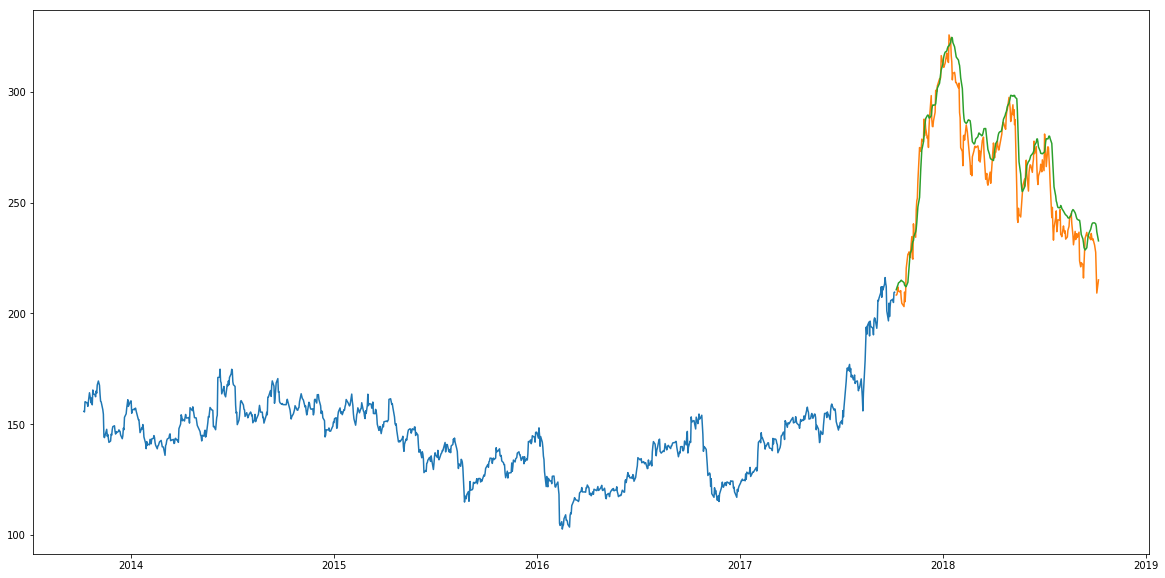
train = new\_data[:987]

valid = new\_data[987:]

valid['Predictions'] = closing\_price

plt.plot(train['Close'])

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



**Inference**

Wow! The LSTM model can be tuned for various parameters such as changing the number of LSTM layers, adding dropout value or increasing the number of epochs. But are the predictions from LSTM enough to identify whether the stock price will increase or decrease? Certainly not!

As I mentioned at the start of the article, stock price is affected by the news about the company and other factors like demonetization or merger/demerger of the companies. There are certain intangible factors as well which can often be impossible to predict beforehand.