MIE1628 Big Data Science

Final Project Report

Time Series AAPL Stock Value Prediction

By
Han Xu
#1000296583

From
Group 2

Dec 20, 2019

Table of Content

1.0 Introduction

The concept of the time-series is that the instructed data points are in a time-sequential order with an equal interval or time spaces. This kind of data can be forecasted by training the models to predict the data's future performance. This project is to perform a time-series stock value prediction for Apple stock prices (AAPL). The data is in the daily interval, beginning from 1980 to 2018, which has been obtained from Yahoo Finance. The overall price trend is shown in Figure 1. As can be seen, there was not a dramatic fluctuation that happened between 1980 to 1999 until the early 21st century. The purpose of this project is to use the AAPL dataset to conduct several forecasting methods or train some machine learning models by using Pyspark in order to predict the stock prices in recent times, or even for the future.

Figure 1: AAPL close price from 1980 to 2018

There are in total of five models and forecasting methods that are chosen to complete the prediction. The selected forecasting methods are Simple Moving Average and Autoregressive Integrated Moving Average Model, as well as the three machine learning models, are including Linear Regression, Random Forest (Regression Tree), and Extreme Gradient Boost. Each model will be evaluated by using the Rooted Mean Square Error (RMSE) and Symmetric Mean Absolute Percentage Error (sMAPE), and their performances will also be further compared and analyzed.

2.0 Features Extraction and Collection

In the purpose of predicting the time series stock price, there are in general two kinds of features that can be used. They can be classified as the factors that can be generated from the original dataset, as well as the indexes from the external resources. Note that due to the external indexes will also contain a lot of missing values or other fraudulent noises, therefore the data cleaning procedure will take place after all features are extracted and collected. In a nutshell, there are in total of 34 features that will be used for model training and testing. Please refer to Appendix A to see the detailed feature extraction process.

2.1 Internal Factors

The internal factors are those features that can be created or calculated from the original dataset. For example, in time-series prediction, lag is one of the most sufficient features that can boost the performance of the regression models. Besides, several financial factors such as returns, log returns, losses (delta), can also be calculated out from the AAPL historical close prices. A detailed table of the internal factors is shown as the following:

Internal Factors	**Description**
Rate of Return	The daily return that can be calculated as *Pi+1/Pi - 1*
Sign of Return	The + or - sign of the trend of the daily rate of return. If the rate increases, it returns a + sign.
Absolutely Return	The absolute value of the rate of return.
Log Return	The daily rate of return with logarithm, calculated as *ln(Pi+1/Pi - 1)*
Absolutely Log Return	The absolute value of the Log return
Lag for 1 Day	The pushed stock price for 1 day forward
Lag for 1 Week	The pushed stock price for 5 days forward
Lag for 2 Weeks	The pushed stock price for 10 days forward
Lag for 1 Month	The pushed stock price for 20 days forward
Lag for 4 Months	The pushed stock price for 80 days forward
Losses (Delta)	The losses or earns (delta) of the stock prices, can be calculated as *P1 - P0*
Rolling Average 1 Day	The average price of the recent 2 days can be calculated as *(Pi+1 + Pi)/2*
Rolling Average 1 Week	The average price of the recent 5 days can be calculated as $\sum_{i=1}^{n=5} (Pi) / n$
Rolling Average 2 Weeks	The average price of the recent 10 days can be calculated as $\sum_{i=1}^{n=10} (Pi) / n$
Rolling Average 1 Month	The average price of the recent 20 days can be calculated as $\sum_{i=1}^{n=20} (Pi) / n$
Rolling Average 1 Months	The average price of the recent 80 days can be calculated as $\sum_{i=1}^{n=80} (Pi) / n$

Table 1: Internal Factors

2.2 External Indexes

The external indexes are the stock prices from other relative or competitive companies, or the price from the stock market indexes. They are extracted from numerous external datasets which are also found on yahoo finance. They include:

External Indexes	**Categories**
Best Buy	Relative Company Stock Price
EU100	European Market Index
France Index	Franch Market Index
Germany Index	Germany Market Index
HSI	Hongkong (China) Market Index
Japan N255	Japanese Market Index
NASDAQ	American Market Index
IBM	Competitive Company Stock Price
Intel	Relative Company Stock Price
Oracle	Relative Company Stock Price
Qualcomm	Relative Company Stock Price
Samsung	Competitive Company Stock Price
Shanghai Index	Chinese Stock Market Index
S&P 500	American Stock Market Index
AMD	Relative Company Stock Price
Amazon	Relative Company Stock Price
Oil Price	Natural Resource Market Index
Walmart	Relative Company Stock Price

Table 2: External Indexes

3.0 Data Engineering

The major fraudulence from the existing dataset is the missing values. For example, several indexes, as well as the AAPL price itself are containing periods of missing values. The manner to deal with this problem is to implement linear interpolation. Referring to Figure 2, the missing values are existing between day 6 to day 8, therefore a straight line will be fitted between the existing values at day 5 and day 9 and assign the values along with the fitted line to replace the missing values. Although this method is better than simply assigning the previous existing numbers or giving random values because, in certain degrees, it restored the historical tendency of the stock development, it still contains some disadvantages that the developing tendency of the stock price is assumed to be developed in one-way direction constantly without any fluctuations. However, the missing values within the features will only exist in a short period instead of for a long time, therefore the linear interpolation is reasonable to be implemented in this case. Additionally, by observation of the original dataset, it can be realized that before 1999, the AAPL stock price had little fluctuations without significant raising or deduction. For the purpose of predicting the recent stock price, the recent data is more appropriate to be used for the training of the model. Therefore the data after 1999 has been selected for further prediction. After the data cleaning procedure, the *dropna()* function is used to further eliminate the non-existing data. For a detailed operation process and code, please refers to Appendix A.

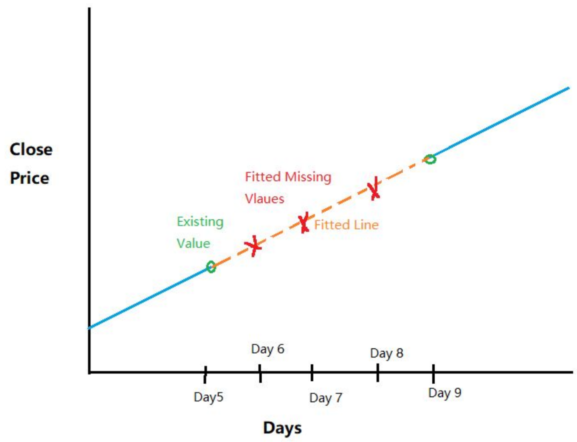

Figure 2: Linear Interpolation Example

4.0 Features Correlation

The features correlation matrix is shown in the file Feature.html (or the code link within Appendix A) (it is not showing here since it is too large). As can be seen from the matrix, the red and dark blue cubes are the features that have certain relativity to the target close price. The target price is in the 6th column, and it can be concluded that the features above the sign of the rate of return are certainly correlated to the label. Besides, a pair plot graph is provided below the correlation matrix, which is showing the linear relationship with the target price. However, to implement certain models, it is better to use the relative feature importance and feature coefficient in order to optimize the model performance.

5.0 Model Implementation

There are in total of five models that have been implemented to this AAPL time series prediction. Two of them are price forecasting methods, and three of them are machine learning models. The forecasting methods include Simple Moving Average and Autoregressive Integrated Moving Average, on the other hand, the machine learning models such as Linear Regression, Random Forest and Extreme Gradient Boosting Regression are conducted.

5.1 Simple Moving Average Forecasting Method (SMA)

The SMA is one of the moving average arithmetics. It can be determined by adding up all of the most recent closing stock prices within a time period and then dividing them by the number of days within that period. The SMA will be calculated in a time horizon of 1 Day, 1 Week, 2 Weeks, 1 Month and 4 Months. Each moving average will be used as the prediction in order to predict the future stock price. The performance of this forecasting method is evaluated by using the symmetric mean absolute percentage error (sMAPE) and shown in Table 3. As can be seen, the sMAPE score is increasing with time horizons, with the smallest error for 1-day prediction and the largest score for 4 months result. This indicates that this method will be more accurate if the prediction is conducting for recent prices. Also, please refer to Appendix B to see the detailed code and implementation (including performance graph).

5.2 Autoregressive Integrated Moving Average Method (ARIMA)

The ARIMA is one of the statistics and econometric model which in particular to conduct the time series prediction. It can provide a decent understanding of the data as well as the prediction of future forecasting. Besides, the high-performance will be realized when dealing with the evidential and non-stationary dataset which the initial integrated step is able to be implemented multiple times to minimize the non-stationarity.

For implementation, the data need to be initially checked for seasonality, and then make the data become stationary such as log and shift. The training and testing set in this case are set to be 0.75/0.25 since it will make the prediction more accurate than 0.7/0.3. The key parameters such as p, d and q are decided by checking the standard deviation and autocorrelations (ACF and PACF) within a for loop. Afterwards, the model can be trained with the training set and then tested with the test set. The evaluation matrices for this model will be based on sMAPE, and the values are shown in Table 3. Also, please refer to Appendix C to see the detailed code and implementation, as well as the performance graph.

5.3 Linear Regression Machine Learning Model

Linear regression is one of the supervised machine learning models that are preferable to be used in time series analysis. The basic idea of the linear regression is to fit a line or curve in between the massive and fluctuate data to predict its general future performance. Due to the existing dataset is containing a large number of features, some of them may not play a significant role in the training of the model to cause underfitting. Therefore, the coefficient of the model is determined and analyzed at the start and those with the lower coefficient were dropped to improve future performance. The dataset has been split into training and testing sets with a ratio of 0.7/0.3, in which the first 70% of the dataset becomes the training set and the last 30% for testing. Besides, in order to accomplish the time horizon prediction, the model is trained for five times with different labels such as closing price, lag 1 week, lag 2 weeks, lag 1 month and lag 4 months. For detailed code and implementation please refer to Appendix D (including the table of feature coefficients and performance graph). And each prediction result is evaluated by both RMSE and sMAPE which are shown in Table 3.

5.4 Random Forest Machine Learning Model

The random forest is also classified as a supervised learning regression model, also known as the regression tree. The regression tree is built through a binary recursive partitioning process, from the root node to the decision nodes and then finally to the terminal nodes. The root node represents the entire population or sample, and it will be further split the data into two partitions based on the evaluation matrices or threshold. The algorithm chooses the predictor and cutpoints that reduce the sum of squad error, and the splitting is made by deciding to group the variables into the homogeneous classes. Finally, it will reach the Leaf, known as the terminal node to reach the endpoint. On the other hand, the random forest is an ensemble version of the decision trees. It is a learning algorithm that selects, at each candidate split in the learning process a random subset of the features. It trains each tree independently, using a random sample of data, and this randomness helps to make the model more robust than a single decision tree, with less chance of overfitting.

For implementation, the delta (losses) is calculated as the target for the prediction instead of the close price itself. Besides, the features are evaluated by using the features coefficients, and with those coefficients greater than 0.01 is selected to be the significant features for training this model. The data is split into 0.7/0.3 for training and testing set and the baseline model is trained with the training set. Afterwards, the hyperparameter is tuned in order to select the best parameter for the secondary model training. The prediction result is evaluated by using RMSE and sMAPE, which are shown in Table 3. Last but not least, please refer to Appendix E (including the table of feature coefficients) to see the detailed code and implementation.

5.5 Extreme Gradient Boosting Regression Machine Learning Model (XGBoost)

The XGBoost model is an advanced version of the Gradient Boosted Machines (GBM). In GBM, one tree will be trained at a time and the new tree will help to correct the errors from the previous tree. The error is measured by using the MSE, and each new tree will try to minimize this error until it finds the optimal solution. The XGB is operating in the same way as the regression trees with several advanced factors such as regularizations, handling spare data, weighted quantile sketch, block structure for parallel learning, out-of-core computing and etc. The most important factor behind the XGBoost is its scalability in all scenarios. The system runs more than ten times faster than the existing popular solution on a single machine and scales to billions of examples in distributed or memory-limited settings.

The operation for the XGBoost model is in a similar way to the Random Forest Model. Firstly the feature is selected by using the feature importance of the baseline model. The features with the proportion which is shown as the bar chart, and with those greater than 0.01 are selected to be used for this model. The data set splitting is in the ratio of 0.7/0.3, and the initial training is completed by using the 70% training set. Afterwards, the hyperparameter tuning process is conducted to select the most appropriate hyperparameters for the secondary training to optimize the model performance. As a result, the prediction is evaluated by RMSE and sMAPE which are shown in Table 3. Also, please refer to Appendix F (including the table of feature importances and the performance graph). to see the detailed code and implementation.

6.0 Model Comparison

The performances of all of the models mentioned above are listed in Table 3. As can be seen, the overall performance is quite similar that the prediction with a short time horizon will provide the most precise results and the accuracy will be getting worse with the interval of the time horizon increases. By looking at the sMAPE, the Linear Regression model will provide the best performance to compare with other machine learning models within all time horizons, which can be indicated that this model is extremely useful for conducting the time-series analysis. Besides, even though the SMA will be giving a better performance over the long time horizon intervals (greater than 1 month), the linear regression model is still preferred because the calculation of the SMA is too simple that it is containing a lot of biases. Also, the ARIMA forecasting method will provide the best score for 4 months prediction, which represents that this model is suitable for long-time analysis. For advanced machine learning models, due to XGBoost is an improved version of other regression trees, its performance is better than the Random Forest. In conclusion, in time-series analysis, the Linear Regression model will provide the most accurate score for a short term prediction which is within a month; however, the long-term analysis is preferred to be conducted based on ARIMA.

Time	RSME			sMAPE				
	Linear Regression	Random Forest	XGBoost	SMA	ARIMA	Linear Regression	Random Forest	XGBoost
1 Day	1.997	3.416	2.303	1.984%	1.516%	1.120%	1.398%	1.329%
1 Week	4.190	6.978	4.863	2.697%	3.114%	2.497%	2.967%	2.989%
2 Weeks	6.023	9.750	6.718	3.687%	4.311%	3.632%	4.084%	4.121%
1 Month	9.756	16.782	11.108	5.127%	6.082%	5.803%	6.907%	6.511%
4 Months	18.60	33.834	22.738	11.335%	11.241%	13.726%	15.02%	14.209%

Table 3: *Result Evaluation for All Models and Forecasting Methods in 5 Different Time Horizons*

7.0 Conclusion and Business Insight

This project has a great business insight for the customers to be used for their future stock investment simulation. Initially, this project will provide a clear numerical and graphical understanding to the customers, making them easy to acknowledge the tendency and the accuracy of the models. Besides, the models are conducted based on Spark, therefore it is able to obtain the newest daily data to predict the future prices of the stocks in a certain time horizon. This will provide a useful investment strategy for the customers to modify their weight of the portfolio to maximize their return, as well as minimize the variances. In the industry of portfolio, bound or options optimizations, the investment strategies such as maximum return, minimum variances, robust optimization, Sharpe ratio optimization as well as the interior-point optimization will be analyzed based on the historical data only. Therefore, if the strategy analysis could be involved with the prediction of the future stock prices, the selection of the investment strategy will be more appropriate and efficient for the customers to not only acquire the maximized return from their portfolio investment, but also in some cases to minimize their risks (such as financial crisis) to keep competitive on the capital market industry. In a nutshell, as mentioned above, the team encourages the customer to use the Linear Regression model for conducting a short term prediction, as well as a long term analysis should be completed by using ARIMA.

8.0 Appendices

Appendix A: Features Selection and Data Cleaning

Features Selection and Data Cleaning Code Link:
https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/694794198277716/4206302335939029/4949361027789338/latest.html

```
#Library Import
import math
import matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
import time
import datetime as dt
from matplotlib import pyplot as plt

from pyspark.sql import SQLContext, Window
from pyspark.sql.functions import *
from pyspark.ml.regression import LinearRegression
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import RegressionEvaluator
from  pyspark.sql.functions import abs, sqrt

# Open AAPL as spark dataframe
import pyspark.sql.functions as func

sqlContext = SQLContext(sc)
df = sqlContext.sql('SELECT * FROM apple')
df.show(5)

# INDEXES
df_sp = sqlContext.sql("SELECT Date, Close FROM sp500")
df_nsdq = sqlContext.sql("SELECT Date, Close FROM nsdq")
pd_sp = df_sp.toPandas()
pd_nsdq = df_nsdq.toPandas()

#Original Data Plot
pd1 = df.toPandas()
ax = pd1.plot(x='Date', y='Close', style='b-', grid=True, linewidth = 2)
ax.set_xlabel("Date")
ax.set_ylabel("USD")
ax.set_title("Close Price of AAPL")
display()
```

```
#Data cleaning
aapl = pd1
# Fill Missing Values for the Original Dataset
i = 0
test = []
for column in range(1,len(aapl.columns)):
  for items in aapl.iloc[:,column]:
    if i>0 and i < len(aapl)-1:
      if np.isnan(items):
        lag = aapl.iloc[i-1,column]
        roof = aapl.iloc[i+1,column]
        A = (roof-lag)/2
        nrplc = A+lag
        aapl.iloc[i,column] = nrplc
    i+=1
  i=0

# Additional features from external recources
sqlContext = SQLContext(sc)
df_apple = sqlContext.sql("SELECT Date, Close as Apple_Close FROM apple")
df_sp500 = sqlContext.sql("SELECT Date, Close as SP500 FROM sp500")
df_nsdq = sqlContext.sql("SELECT Date, Close as NSDQ FROM nsdq")
df_amazon = sqlContext.sql("SELECT Date, Close as Amazon FROM amazon")
df_amd = sqlContext.sql("SELECT Date, Close as AMD FROM amd_csv")
df_eu100 = sqlContext.sql("SELECT Date, Close as EU100 FROM eu_n100")
df_fr = sqlContext.sql("SELECT Date, Close as France_Index FROM france")
df_ger = sqlContext.sql("SELECT Date, Close as Germany_Index FROM germany")
df_hk = sqlContext.sql("SELECT Date, Close as HSI FROM hongkong_hsi")
df_jp = sqlContext.sql("SELECT Date, Close as Japan_N225 FROM japan_n225")
df_sh = sqlContext.sql("SELECT Date, Close as Shanghai FROM shanghai")
df_ibm = sqlContext.sql("SELECT Date, Close as IBM FROM ibm_csv")
df_intel = sqlContext.sql("SELECT Date, Close as Intel FROM intel")
df_ms = sqlContext.sql("SELECT Date, Close as Microsoft FROM microsoft")
df_orcl = sqlContext.sql("SELECT Date, Close as Orcl FROM orcl_csv")
df_qual = sqlContext.sql("SELECT Date, Close as Qualcomm FROM qualcomm")
df_sam = sqlContext.sql("SELECT Date, Close as Samsung FROM samsung")
df_wal = sqlContext.sql("SELECT Date, Close as Walmart FROM wmt_csv")
df_bb = sqlContext.sql("SELECT Date, Close as BestBuy FROM bby_bestbuy")
df_gold = sqlContext.sql("SELECT Date, 'Gold price' as Gold FROM gold")
df_oil = sqlContext.sql("SELECT Date, DCOILWTICO as Oil_price FROM oilwtico_csv")

df_feature_index = df_apple.join(df_bb, "Date", how = 'left').join(df_eu100, "Date", how = 'left').join(df_fr, "Date",
how = 'left').join(df_ger, "Date", how = 'left').join(df_hk, "Date", how = 'left').join(df_jp, "Date", how =
'left').join(df_ms, "Date", how = 'left').join(df_nsdq, "Date", how = 'left').join(df_ibm, "Date", how =
'left').join(df_intel, "Date", how = 'left').join(df_orcl, "Date", how = 'left').join(df_qual, "Date", how =
'left').join(df_sam, "Date", how = 'left').join(df_sh, "Date", how = 'left').join(df_sp500, "Date", how =
```

```
'left').join(df_amd, "Date", how = 'left').join(df_amazon, "Date", how = 'left').join(df_gold, "Date", how =
'left').join(df_oil, "Date", how = 'left').join(df_wal, "Date", how = 'left')

#Covert all strings to floats
df_I = df_feature_index.toPandas()
df_I.replace(to_replace=[None], value=np.nan, inplace=True)
i = 0
for column in range(len(df_I.columns)):
  for items in df_I.iloc[:,column]:
    if items == 'null' or items == '.':
      df_I.iloc[i,column] = np.nan
    i+=1
  i=0
for j in range(1,len(df_I.columns)):
  df_I.iloc[:,j]=df_I.iloc[:,j].astype(float)

for column in range(1,len(df_I.columns)):
  for items in df_I.iloc[:,column]:
    if items < 0:
      df_I.iloc[i,column] = np.nan
    i+=1
  i=0

#fill the missing values, exluding the time when the index is not on the market yet.
for col in range(1,len(df_I.columns)):
  i=-1
  for a in df_I.iloc[:,col]:
    i+=1
    if i>0 and i < len(df_I)-1:
      if np.isnan(a):
        lagI = df_I.iloc[i-1,col]
        if np.isnan(lagI):
          continue
        else:
          n=1
          for j in range(i,len(df_I)):
            if np.isnan(df_I.iloc[j,col]):
              n+=1
            else:
              roofI = df_I.iloc[j,col]
              break
          Ai = (roofI-lagI)/n
          nrplcI = Ai+lagI
          df_I.iloc[i,col] = nrplcI

#fill rest of the NaN to 0
df_I = df_I.loc[:, df_I.isnull().mean() < .7]
df_I = df_I.fillna(0)

#Constructing Features
# Indexes
df_I.drop(labels=['Date'], axis=1,inplace = True)
```

```
df_I.drop(labels=['Apple_Close'], axis=1,inplace = True)
# calculate pct change of the Adj. Close between records
aapl['return'] = aapl['Close']/aapl['Close'].shift(1) - 1
aapl['return'] = aapl['return'].shift(1)
aapl['+_-_ret'] = aapl['return'].apply(np.sign)
aapl['abs_ret'] = aapl['return'].abs()

# calculate the log return on Adj. Close between records
aapl['log_return'] = np.log(aapl['Close']) - np.log(aapl['Close'].shift(1))
aapl['log_return'] = aapl['log_return'].shift(1)
aapl['abs_log_ret'] = aapl['log_return'].abs()

aapl['lag_1D'] = aapl['Close'].shift(1)
aapl['lag_1W'] = aapl['Close'].shift(5)
aapl['lag_2W'] = aapl['Close'].shift(10)
aapl['lag_1M'] = aapl['Close'].shift(26)
aapl['lag_4M'] = aapl['Close'].shift(104)

aapl['loss'] = aapl['Close'] - aapl['Close'].shift(-1)

#rolling average
ma_c = aapl["Close"]
df_matitle = ['Mov_Avg_1D','Mov_Avg_1W','Mov_Avg_2W','Mov_Avg_1M','Mov_Avg_4M']
column = 0
for days in (2,5,10,20,80):
  mov_a = []
  i = 0
  sum_a = 0
  for rows in ma_c:
    if i < days:
      mov_a.append(ma_c[i])
      i=i+1
    else:
      for j in range(days):
        sum_a = sum_a + ma_c[i-j-1]
      avg = sum_a/days
      mov_a.append(avg)
      sum_a = 0
      i=i+1
  aapl[df_matitle[column]] = pd.Series(mov_a)
  column = column + 1

#Move Adj close to the front
clo = aapl['Close']
aapl.drop(labels=['Adj Close'], axis=1,inplace = True)
aapl.drop(labels=['Close'], axis=1,inplace = True)
aapl.drop(labels=['High'], axis=1,inplace = True)
aapl.drop(labels=['Low'], axis=1,inplace = True)
aapl.drop(labels=['Open'], axis=1,inplace = True)
aapl.drop(labels=['Volume'], axis=1,inplace = True)
aapl.insert(1, 'Close', clo)

#Inset Indexes
aapl = pd.concat([aapl, df_I], axis=1)
```

```
aapl.head()

# remove the columns that contains over 70% of NaN
# fill the miss values for the features
aapl = aapl.loc[:, aapl.isnull().mean() < .3]
i = 0
for col in range(1,len(aapl.columns)):
  i=-1
  for a in aapl.iloc[:,col]:
    i+=1
    if i>0 and i < len(aapl)-1:
      if np.isnan(a):
        lag = aapl.iloc[i-1,col]
        if np.isnan(lag):
          continue
        else:
          n=1
          for j in range(i,len(aapl)):
            if np.isnan(aapl.iloc[j,col]):
              n+=1
            else:
              roof = aapl.iloc[j,col]
              break
          A = (roof-lag)/n
          nrplc = A+lag
          aapl.iloc[i,col] = nrplc
aapl = aapl.dropna()
aapl.head()

#Correlation Matrix Plot
from sklearn.preprocessing import StandardScaler,Normalizer
from scipy.cluster import hierarchy
from scipy.spatial import distance
corr_matrix = aapl.drop(['Date'],axis=1).corr()
corr_matrix = aapl.corr()
correlations_array = np.asarray(corr_matrix)
linkage = hierarchy.linkage(distance.pdist(correlations_array), \
                            method='average')

g = sns.clustermap(corr_matrix,row_linkage=linkage,col_linkage=linkage,\
row_cluster=True,col_cluster=True,annot=True,fmt=".2f",annot_kws={'size':8},figsize=(20,20),cmap='coolwarm')
plt.setp(g.ax_heatmap.yaxis.get_majorticklabels(), rotation=0)
display()
label_order = corr_matrix.iloc[:,g.dendrogram_row.reordered_ind].columns

#Pairplot Graph Plotting
import seaborn as sns; sns.set(style="ticks", color_codes=True)
g = sns.pairplot(aapl)
display()

#Decision Tree Featrues Importance
data = aapl.drop(['Date'], axis=1)
X = data.iloc[:,1:50]  #independent columns
y = data.iloc[:,0]    #target column i.e price range
```

```
from sklearn.tree import DecisionTreeRegressor
# model = ExtraTreesClassifier()
model = DecisionTreeRegressor()
model.fit(X,y)
importances = model.feature_importances_
labels = list(aapl.columns)
labels.remove('Date')
labels.remove('Close')
table_FI = {'Importances':importances,'Labels':labels}
df_FI = pd.DataFrame(table_FI)
df_FI.sort_values(by='Importances',ascending=False)
# df_FI['Importances'].sum()

#XGBoost Features Importance
!pip install xgboost
import xgboost as xgb
from xgboost import XGBRegressor
model_xgb = XGBRegressor()
model_xgb.fit(X,y)
importances_xgb = model_xgb.feature_importances_
table_FI_xgb = {'Importances':importances_xgb,'Labels':labels}
df_FI_xgb = pd.DataFrame(table_FI_xgb)
df_FI_xgb.sort_values(by='Importances',ascending=False)

#Linear Regression Features Coefficient
from sklearn.linear_model import LinearRegression
model3 = LinearRegression()
model3.fit(X,y)
coef = model3.coef_
table_FI_lm = {'Coefficient':coef,'Labels':labels}
df_FI_lm = pd.DataFrame(table_FI_lm)
df_FI_lm.sort_values(by='Coefficient',ascending=False)

#Save the Features table to dbsf for future use
aapl.to_csv("/dbfs/FileStore/tables/features_final.csv", sep=',')
```

Appendix B: SMA Model Implementation

SMA Code Link:
https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/694794198277716/4206302335939040/4949361027789338/latest.html

```
#Libary Import
import math
import matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
import time
import datetime as dt
from matplotlib import pyplot as plt

from pyspark.sql import SQLContext, Window
from pyspark.sql.functions import *
from pyspark.ml.regression import LinearRegression
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import RegressionEvaluator
from  pyspark.sql.functions import abs, sqrt

# Open AAPL as spark dataframe from 1999 to 2018
import pyspark.sql.functions as func
sqlContext = SQLContext(sc)
df = sqlContext.sql("SELECT * FROM apple WHERE YEAR(apple.Date) BETWEEN 1999 AND 2018")
df.show()

# Fill Missing Values
df_ma = df.select("Date","Adj Close").toPandas()
i = 0
for column in range(1,len(df_ma.columns)):
  for items in df_ma.iloc[:,column]:
    if i>0 and i < len(df_ma)-1:
      if np.isnan(items):
        lag = df_ma.iloc[i-1,column]
        roof = df_ma.iloc[i+1,column]
        A = (roof-lag)/2
        nrplc = A+lag
        df_ma.iloc[i,column] = nrplc
    i+=1
  i=0
ma_c = df_ma["Adj Close"]

# Moving Average Calculation
df_matitle = ['Mov_Avg_1D','Mov_Avg_1W','Mov_Avg_2W','Mov_Avg_1M','Mov_Avg_4M']
column = 0
sMAPE = []
for days in (2,5,10,20,80):
```

```
  mov_a = []
  i = 0
  sum_a = 0
  for rows in ma_c:
    if i < days:
      mov_a.append(ma_c[i])
      i=i+1
    else:
      for j in range(days):
        sum_a = sum_a + ma_c[i-j-1]
      avg = sum_a/days
      mov_a.append(avg)
      sum_a = 0
      i=i+1
  df_ma[df_matitle[column]] = pd.Series(mov_a)
  column = column + 1
df_ma.head(6)

# sMAPE Evaluation
def SMAPE(A, F):
  score = 100/len(A) * np.sum(2 * np.abs(F - A) / (np.abs(A) + np.abs(F)))
  return score

target=df_ma.iloc[:,1].values
d1=df_ma.iloc[:,2].values
w1=df_ma.iloc[:,3].values
w2=df_ma.iloc[:,4].values
m1=df_ma.iloc[:,5].values
m4=df_ma.iloc[:,6].values

print(SMAPE(target, d1))
print(SMAPE(target, w1))
print(SMAPE(target, w2))
print(SMAPE(target, m1))
print(SMAPE(target, m4))

# Evaluation Plot
split_ratio = 0.7
ratio = math.ceil(len(df_ma)*split_ratio)

df_ma_plot = df_ma[ratio:]

ax = df_ma_plot.plot(x='Date',grid=True,figsize=(15,8))
ax.set_xlabel("date")
ax.set_ylabel("USD")
display()
```

Figure B: Prediction Tendency for 5 Time Horizons from the SMA method

Appendix C: ARIMA Model Implementation

ARIMA Code Link:
https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/694794198277716/1440830649323514/4949361027789338/latest.html

```
#Library Import
import math
import matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
import time
import datetime as dt
import pyspark.sql.functions as func
from pandas import datetime
from pandas import DataFrame
from matplotlib import pyplot as plt
from pyspark.sql import SQLContext, Window
from pyspark.sql.functions import *
from pyspark.ml.regression import LinearRegression
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import RegressionEvaluator
from  pyspark.sql.functions import abs, sqrt
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
import numpy as np, pandas as pd
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from pandas.plotting import autocorrelation_plot
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from pyspark.sql.types import DateType
from pyspark.sql.functions import lit
from statsmodels.tsa.stattools import adfuller, kpss, acf
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose
import statsmodels.api as sm
import scipy.stats as scs
import statsmodels.tsa.api as smt
from pylab import rcParams

# Open AAPL as df--------------
import pyspark.sql.functions as func
sqlContext = SQLContext(sc)
df = sqlContext.sql("SELECT Date, Close FROM apple WHERE YEAR(apple.Date) BETWEEN 2013 AND
2018")
df = df.toPandas()
df['Date'] = df['Date'].apply(lambda x:
```

```
                              dt.datetime.strptime(x,'%Y-%m-%d'))
df.set_index('Date',inplace = True)
df.head()

## Fill Missing Values-----------------------
df_c = df
i = 0
for column in range(1,len(df_c.columns)):
  for items in df_c.iloc[:,column]:
    if i>0 and i < len(df_c)-1:
      if np.isnan(items):
        lag = df_c.iloc[i-1,column]
        roof = df_c.iloc[i+1,column]
        A = (roof-lag)/2
        nrplc = A+lag
        df_c.iloc[i,column] = nrplc
    i+=1
  i=0
df = df_c
df_c.head()

plt1= df.plot( style='b-', grid=True, linewidth = 1, figsize = (20,5))
plt1.set_xlabel("Date")
plt1.set_ylabel("Close price (USD)")
plt1.set_title("Close Price of AAPL")
display(plt1)

def get_stationarity(timeseries):

    # rolling statistics
    rolling_mean = timeseries.rolling(10).mean()
    rolling_std = timeseries.rolling(10).std()

    # Dickey–Fuller test:
    result = adfuller(timeseries)
    print('ADF Statistic: {}'.format(result[0]))
    print('p-value: {}'.format(result[1]))
    print('Critical Values:')
    for key, value in result[4].items():
        print('\t{}: {}'.format(key, value))

# transforming the data to stationary with log and shift----------------------------------
df_log = np.log(df.Close)
df_log_shift = (df_log-df_log.shift(1))
rev_close = np.exp(df_log)
diff = (rev_close-rev_close.shift(1)).dropna()
diff
get_stationarity(diff)
diff = pd.DataFrame(diff)
plt0= diff.plot( style='b-', grid=True, linewidth = 1, figsize = (20,5),title ="Delta transformed close price")
plt0.set_xlabel("Date")
plt0.set_ylabel("Close price (USD)")
display()
```

```
# check log stationarity
get_stationarity(df_log)
df_log = pd.DataFrame(df_log)
plt2= df_log.plot( style='b-', grid=True, linewidth = 1, figsize = (20,5),title ="Log transformed close price")
plt2.set_xlabel("Date")
plt2.set_ylabel("Close price (USD)")
display()
df_log.head()

# check log shift stationarity----------------------------------
df_log_shift = df_log_shift.dropna()
get_stationarity(df_log_shift)
df_log_shift = pd.DataFrame(df_log_shift)
plt3= df_log_shift.plot( style='b-', grid=True, linewidth = 1, figsize = (20,5),title ="Log Shift transformed close
price")
plt3.set_xlabel("Date")
plt3.set_ylabel("Close price (USD)")
display()
df_log_shift.head()

def acf_pacf_plot_lg(y, lags=None, figsize=(15, 10), style='bmh'):
    fig = plt.figure(figsize=figsize)
    #mpl.rcParams['font.family'] = 'Ubuntu Mono'
    layout = (2, 2)
    acf_ax = plt.subplot2grid(layout, (0, 0), colspan=2)
    plt.plot(y = 0.5)
    pacf_ax = plt.subplot2grid(layout, (1, 0), colspan=2)

    plot_acf(y, lags=lags, ax=acf_ax)
    acf_ax.axhline(y=0.5, color='r', linestyle='--')
    plot_pacf(y, lags=lags, ax=pacf_ax)
    pacf_ax.axhline(y=-0.5, color='r', linestyle='--')

    plt.tight_layout()
    return

def std_check(data, p,d,q):
  diff_model = ARIMA(training, order = (p,d,q))
  result = diff_model.fit(disp=0)
  std = np.std(result.resid)
  acf_pacf_plot_lg(pd.Series(result.resid), lags = 10)
  print('(%d,%d,%d)Standard Deviation: %f' %(p,d,q,std))

#split into training and testing set
df_use = diff
choice = 2
sp = math.ceil(len(df_use)*0.75)
training, testing = df_use.iloc[:sp].values, df_use.iloc[sp:].values
testlog = df_log.iloc[sp:].values
print(len(training))
print(len(testing))
sub_arr = testlog[:-1].copy()
print(len(sub_arr))
```

```
index = df.index.to_frame(index=False)
test= pd.DataFrame(testing)
test.index =  [x for x in range(len(training), len(df_use))]
final_df =
pd.concat([index,pd.DataFrame(df.Close.values),pd.DataFrame(df_use.Close.values),pd.DataFrame(training),test],a
xis = 1)
final_df.columns = ['date','close_price','transformed','train','test']
final_df.head()

#determine parameters
for p in range (0,5):
  for d in range (0,2):
    for q in range (0,5):
      try:
        arima1_original = std_check(training,p,d,q)
      except:
        pass

arima1_original = std_check(training,0,0,0)
display()
arima1_original = std_check(training,2,1,2)
display()

#fit model
i_train = training
i_test = testing
history = [x for x in i_train]
predictions = list()
predictions1w = list()
predictions2w = list()
predictions1m = list()
predictions4m = list()

for t in range(len(i_train),len(i_train)+len(i_test)):

  model = ARIMA(history, order=(1,1,0))
  model_fit = model.fit(disp=0)
  for v in [1,5,10,20,80]:
    output = model_fit.forecast(v)[0][-1]
    if v == 1:
      predictions.append(output)
    if v == 5:
      predictions1w.append(output)
    if v == 10:
      predictions2w.append(output)
    if v == 20:
      predictions1m.append(output)
    if v == 80:
      predictions4m.append(output)

  history.append(i_test[t-len(i_train)])
  print('%f' % (t))

test_list = [i[0] for i in i_test.tolist()]
```

```
df_pred =pd.DataFrame(
    {'Expected':test_list,
     'predictions': predictions ,
     'predictions1w': predictions1w,
     'predictions2w': predictions2w,
     'predictions1m': predictions1m,
     'predictions4m': predictions4m
    })
df_pred.head()


lags = [1,5,10,20,80]
lags[0]
len(df_use)
len(final_df.close_price.iloc[sp:-lags[1]])
a = df_pred.iloc[:,2][:-lags[1]].values
b = df_pred.iloc[:,1][:-lags[1-1]].values
print(b)

def inverse_difference(history, yhat, interval=1):
            return pd.DataFrame(yhat + history)

def reverseLogShift (history_log,yhatLogShift, interval=1):
  his = history_log[:-interval].copy()
  predict_log=inverse_difference(his,yhatLogShift,interval=1)
  predict_price = np.exp(predict_log)
  return predict_price[:-interval].copy()
dfAll = pd.concat([final_df.date,final_df.close_price],axis = 1)
for i in range(1,6):
  lags = [1,5,10,20,80]
  if (choice == 1):
    pred_log_shift = pd.DataFrame(df_pred.iloc[:,i])
    pred = reverseLogShift (testlog, pred_log_shift, interval=lags[i-1])
  if (choice == 2):
    origin = final_df.close_price.iloc[sp:-lags[i-1]].values
    if(i == 1):
      pred_diff = df_pred.iloc[:,i][:].values
    else:
      pred_diff = df_pred.iloc[:,i][:-lags[i-1]+1].values
    pred = inverse_difference(origin, pred_diff, interval=lags[i-1])
  pred.index =  [x for x in range((len(training)+lags[i-1]), (len(df_use)+1))]
  dfAll.insert(i+1,"prediction"+str(i),pred)
dfAll.set_index(['date'],inplace = True)

plotdf=dfAll
plotdf.plot(lineWidth=0.8, figsize = (20,5),title ="Actual vs Prediction")
display()
#Performance Printing________________________
pred_res = plotdf.dropna()
def SMAPE(A, F):
  score = 100/len(A) * np.sum( np.abs(F - A) / ((np.abs(A) + np.abs(F))/2))
  return score
#SMAPE Evaluation—————————————————
for i in range (-5,0):
```

```
target=pred_res.iloc[:,0].values
d1=pred_res.iloc[:,i].values
print(SMAPE(target, d1))
```

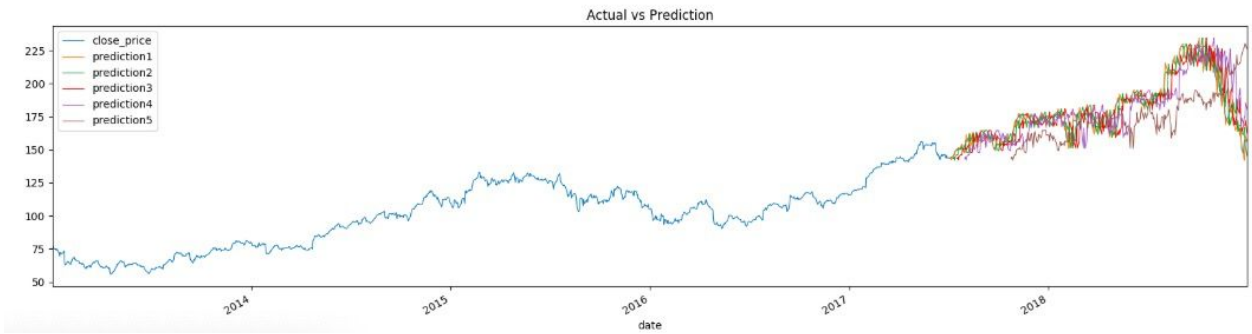

Figure C: Prediction Tendency for 5 Time Horizons from the ARIMA model

Appendix D: Linear Regression Model Implementation

Linear Regression Code Link:
https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/694794198277716/1616028577216904/4949361027789338/latest.html

```
#Library Import
import math
import matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
import time
import datetime as dt
from matplotlib import pyplot as plt

from pyspark.sql import SQLContext, Window
from pyspark.sql.functions import *
from pyspark.ml.regression import LinearRegression
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import RegressionEvaluator
from  pyspark.sql.functions import abs, sqrt

from pyspark.sql import SQLContext, Window
from pyspark.sql.functions import *
from pyspark.ml.regression import LinearRegression
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import RegressionEvaluator
from  pyspark.sql.functions import abs, sqrt

# Open AAPL as Spark Datafram, convert to Panda DF by using pyarrow, the convert all strings to float
# Selected Data from 1999 to 2018
import pyspark.sql.functions as func
sqlContext = SQLContext(sc)
df = sqlContext.sql('SELECT * FROM features WHERE YEAR(features.Date) BETWEEN 1999 AND 2018')
df = df.toPandas()
df.drop(labels=['_c0'], axis=1,inplace = True)

for i in range(1,len(df.columns)):
  df.iloc[:,i]=df.iloc[:,i].astype(float)

df['lag_2D'] = df['Close'].shift(2)
# df['lag_7D'] = df['Close'].shift(7)
df['lag_20D'] = df['Close'].shift(20)
df['lag_102D'] = df['Close'].shift(102)
df = df.dropna()
df.head()

# Drop the features with lower coefficient
```

```
df.drop(labels=['Japan_N225'], axis=1,inplace = True)
df.drop(labels=['HSI'], axis=1,inplace = True)
df.drop(labels=['France_Index'], axis=1,inplace = True)
df.drop(labels=['EU100'], axis=1,inplace = True)
df.drop(labels=['AMD'], axis=1,inplace = True)
df.drop(labels=['Walmart'], axis=1,inplace = True)
df.drop(labels=['Intel'], axis=1,inplace = True)
df.drop(labels=['loss'], axis=1,inplace = True)
df.drop(labels=['log_return'], axis=1,inplace = True)
df.drop(labels=['abs_log_ret'], axis=1,inplace = True)
df.drop(labels=['Mov_Avg_2W'], axis=1,inplace = True)
df.drop(labels=['Mov_Avg_1D'], axis=1,inplace = True)
sdf = sqlContext.createDataFrame(df)
sdf.show()

#Remove the potential targets
head_lis = sdf.schema.names
head_lis.remove('Date')
head_lis.remove('Close')
head_lis.remove('lag_1W')
head_lis.remove('lag_2W')
head_lis.remove('lag_1M')
head_lis.remove('lag_4M')

# Split the set by 0.7/0.3 and train the model 5 times with different labels in time horizon
features = head_lis
vectorAssembler = VectorAssembler(inputCols=features, outputCol="features")
df1 = vectorAssembler.transform(sdf)

# Split the data with first 0.7 for traning and last 0.3 for testing
df1 = df1.toPandas()
split_ratio = 0.7
ratio = math.ceil(len(df1)*split_ratio)
train_datapd = df1[:ratio]
test_datapd = df1[ratio:]

train_data = sqlContext.createDataFrame(train_datapd)
test_data = sqlContext.createDataFrame(test_datapd)

lr = LinearRegression(featuresCol = "features", labelCol="Close")
model = lr.fit(train_data)
pred_results = model.transform(test_data)

lr = LinearRegression(featuresCol = "features", labelCol="lag_1W")
model1w = lr.fit(train_data)
pred_results1w = model1w.transform(test_data)

lr = LinearRegression(featuresCol = "features", labelCol="lag_2W")
model2w = lr.fit(train_data)
pred_results2w = model2w.transform(test_data)

lr = LinearRegression(featuresCol = "features", labelCol="lag_1M")
model1m = lr.fit(train_data)
pred_results1m = model1m.transform(test_data)
```

```
lr = LinearRegression(featuresCol = "features", labelCol="lag_4M")
model4m = lr.fit(train_data)
pred_results4m = model4m.transform(test_data)

# Valuation Matrices of RMSE
evaluator= RegressionEvaluator(labelCol="Close", predictionCol="prediction", metricName="rmse")
print(evaluator.evaluate(pred_results))
print(evaluator.evaluate(pred_results1w))
print(evaluator.evaluate(pred_results2w))
print(evaluator.evaluate(pred_results1m))
print(evaluator.evaluate(pred_results4m))

#Insert the predictions into the original table
pd_predr = pred_results.toPandas()
pd_predr1w = pred_results1w.toPandas()
pd_predr2w = pred_results2w.toPandas()
pd_predr1m = pred_results1m.toPandas()
pd_predr4m = pred_results4m.toPandas()

pd_predr['prediction_1W'] = pd_predr1w['prediction']
pd_predr['prediction_2W'] = pd_predr2w['prediction']
pd_predr['prediction_1M'] = pd_predr1m['prediction']
pd_predr['prediction_4M'] = pd_predr4m['prediction']
pd_predr.head()

#Evaluation Matrices of sMAPE
def SMAPE(A, F):
  score = 100/len(A) * np.sum(2 * np.abs(F - A) / (np.abs(A) + np.abs(F)))
  return score
for i in range(-5,0):
  target=pd_predr.iloc[:,1].values
  pred=pd_predr.iloc[:,i].values
  print(SMAPE(target, pred))

# Evaluation Plotting
pred_plot = pd_predr[['Date','Close','prediction','prediction_1W','prediction_2W','prediction_1M','prediction_4M']]
ax = pred_plot.plot(x='Date',grid=True,figsize=(15,8))
ax.set_xlabel("date")
ax.set_ylabel("USD")
display()
```

	Coefficient	Labels
2	6.907995	abs_ret
0	2.875846	return
5	1.087043	lag_1D
12	0.220958	Mov_Avg_1W
7	0.053211	lag_2W
1	0.022729	+_-_ret
14	0.008344	Mov_Avg_1M
25	0.004403	Orcl
23	0.001071	IBM
26	0.000751	Qualcomm
32	0.000574	Oil_price
15	0.000406	Mov_Avg_4M
31	0.000354	Amazon
16	0.000309	BestBuy
22	0.000173	NSDQ
29	0.000061	SP500
19	0.000037	Germany_Index
28	0.000016	Shanghai
27	0.000012	Samsung
21	-0.000002	Japan_N225
20	-0.000009	HSI
18	-0.000051	France_Index
17	-0.000127	EU100
9	-0.001315	lag_4M

Table D: Features Coefficient for Linear Regression Baseline Model

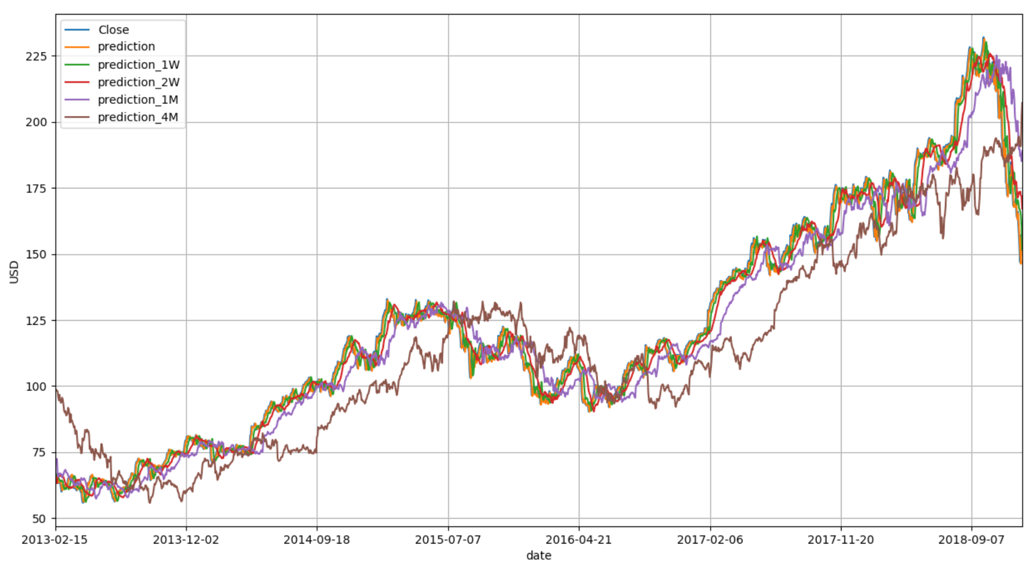

Figure D: Prediction Tendency for 5 Time Horizons from the Linear Regression model

Appendix E: Random Forest Model Implementation

Random Forest Code Link:
https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/694794198277716/3585680806055484/4949361027789338/latest.html

```
import math
import matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
import time
import datetime as dt
from matplotlib import pyplot as plt

from pyspark.sql import SQLContext, Window
from pyspark.sql.functions import *
from pyspark.ml.regression import LinearRegression
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import RegressionEvaluator
from  pyspark.sql.functions import abs, sqrt

from pyspark.ml.tuning import CrossValidator
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import OneHotEncoderEstimator
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml import Pipeline
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import GridSearchCV

from sklearn.tree import DecisionTreeRegressor

from sklearn.model_selection import train_test_split
from sklearn import metrics
from math import sqrt

# additional features
sqlContext = SQLContext(sc)
df_apple = sqlContext.sql("SELECT Date, Close as Apple_Close FROM apple")
df_sp500 = sqlContext.sql("SELECT Date, Close as SP500 FROM sp500")
df_nsdq = sqlContext.sql("SELECT Date, Close as NSDQ FROM nsdq")
df_amazon = sqlContext.sql("SELECT Date, Close as Amazon FROM amazon")
df_amd = sqlContext.sql("SELECT Date, Close as AMD FROM amd_csv")
df_eu100 = sqlContext.sql("SELECT Date, Close as EU100 FROM eu_n100")
df_fr = sqlContext.sql("SELECT Date, Close as France_Index FROM france")
df_ger = sqlContext.sql("SELECT Date, Close as Germany_Index FROM germany")
df_hk = sqlContext.sql("SELECT Date, Close as HSI FROM hongkong_hsi")
df_jp = sqlContext.sql("SELECT Date, Close as Japan_N225 FROM japan_n225")
```

```
df_sh = sqlContext.sql("SELECT Date, Close as Shanghai FROM shanghai")
df_ibm = sqlContext.sql("SELECT Date, Close as IBM FROM ibm_csv")
df_intel = sqlContext.sql("SELECT Date, Close as Intel FROM intel")
df_ms = sqlContext.sql("SELECT Date, Close as Microsoft FROM microsoft")
df_orcl = sqlContext.sql("SELECT Date, Close as Orcl FROM orcl_csv")
df_qual = sqlContext.sql("SELECT Date, Close as Qualcomm FROM qualcomm")
df_sam = sqlContext.sql("SELECT Date, Close as Samsung FROM samsung")
df_wal = sqlContext.sql("SELECT Date, Close as Walmart FROM wmt_csv")
df_bb = sqlContext.sql("SELECT Date, Close as BestBuy FROM bby_bestbuy")
df_oil = sqlContext.sql("SELECT Date, DCOILWTICO as Oil_price FROM oilwtico_csv")

df_feature_index = df_apple.join(df_bb, "Date", how = 'left').join(df_eu100, "Date", how = 'left').join(df_fr, "Date",
how = 'left').join(df_ger, "Date", how = 'left').join(df_hk, "Date", how = 'left').join(df_jp, "Date", how =
'left').join(df_ms, "Date", how = 'left').join(df_nsdq, "Date", how = 'left').join(df_ibm, "Date", how =
'left').join(df_intel, "Date", how = 'left').join(df_orcl, "Date", how = 'left').join(df_qual, "Date", how =
'left').join(df_sam, "Date", how = 'left').join(df_sh, "Date", how = 'left').join(df_sp500, "Date", how =
'left').join(df_amd, "Date", how = 'left').join(df_amazon, "Date", how = 'left').join(df_oil, "Date", how =
'left').join(df_wal, "Date", how = 'left')

df_feature_pd = df_feature_index.toPandas()
df_feature_pd.replace(to_replace=[None], value=np.nan, inplace=True)

df_feature_pd['return'] = df_feature_pd['Apple_Close']/df_feature_pd['Apple_Close'].shift(1) - 1
df_feature_pd['return'] = df_feature_pd['return'].shift(1)
#df_feature_pd['+_-_ret'] = df_feature_pd['return'].apply(np.sign)
df_feature_pd['abs_ret'] = df_feature_pd['return'].abs()

# calculate the log return on Adj. Close between records
df_feature_pd['log_return'] = np.log(df_feature_pd['Apple_Close']) - np.log(df_feature_pd['Apple_Close'].shift(1))
df_feature_pd['log_return'] = df_feature_pd['log_return'].shift(1)
df_feature_pd['abs_log_ret'] = df_feature_pd['log_return'].abs()

df_feature_pd['lag_1D'] = df_feature_pd['Apple_Close'].shift(1)
df_feature_pd['lag_1W'] = df_feature_pd['Apple_Close'].shift(5)
df_feature_pd['lag_2W'] = df_feature_pd['Apple_Close'].shift(10)
df_feature_pd['lag_1M'] = df_feature_pd['Apple_Close'].shift(26)
df_feature_pd['lag_4M'] = df_feature_pd['Apple_Close'].shift(104)

df_feature_pd['loss'] = df_feature_pd['Apple_Close'] - df_feature_pd['Apple_Close'].shift(-1)

ma_c = df_feature_pd["Apple_Close"]
df_matitle = ['Mov_Avg_1D','Mov_Avg_1W','Mov_Avg_2W','Mov_Avg_1M','Mov_Avg_4M']
column = 0
for days in (2,5,10,20,80):
  mov_a = []
  i = 0
  sum_a = 0
  for rows in ma_c:
    if i < days:
      mov_a.append(ma_c[i])
      i=i+1
    else:
      for j in range(days):
        sum_a = sum_a + ma_c[i-j-1]
```

```
    avg = sum_a/days
    mov_a.append(avg)
    sum_a = 0
    i=i+1
  df_feature_pd[df_matitle[column]] = pd.Series(mov_a)
  column = column + 1

df_feature_pd['prediction_1D'] = df_feature_pd['Apple_Close'].shift(-1)
df_feature_pd['prediction_1W'] = df_feature_pd['Apple_Close'].shift(-5)
df_feature_pd['prediction_2W'] = df_feature_pd['Apple_Close'].shift(-10)
df_feature_pd['prediction_1M'] = df_feature_pd['Apple_Close'].shift(-26)
df_feature_pd['prediction_4M'] = df_feature_pd['Apple_Close'].shift(-104)
df_feature_pd.head()

#Fill missing data using linear regression
#If data cant be filled using interpolate, fill it with 0
#Turn all column in dataframe into numeric values except the first column

for col in df_feature_pd.columns[1:]:
  df_feature_pd[col] = pd.to_numeric(df_feature_pd[col], errors='coerce')

#Use linear interpolate the fill NaN data between two existing data
df_feature_pd.interpolate(method='linear',  inplace=True)

#Fill 0 to all other NaN data
df_feature_pd.fillna(0, inplace=True)
df_feature_pd['Date'] = pd.to_datetime(df_feature_pd['Date'], format = '%Y-%m-%d')
df_feature_pd = df_feature_pd[df_feature_pd["Date"]>'2008-01-01']
df_feature_pd.set_index('Date', inplace=True)
df_feature_pd.head()

features = df_feature_pd.drop(['Apple_Close'], axis = 1).drop(['prediction_1D'], axis = 1).drop(['prediction_1W'],
axis = 1).drop(['prediction_2W'], axis = 1).drop(['prediction_1M'], axis = 1).drop(['prediction_4M'], axis = 1)
features.info()

from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import r2_score, median_absolute_error, mean_absolute_error
from sklearn.metrics import median_absolute_error, mean_squared_error, mean_squared_log_error

#Perform train-test split with respect to time series structure
def timeseries_train_test_split(X, y, test_size):

  # get the index after which test set starts
  test_index = int(len(X)*(1-test_size))

  X_train = X.iloc[:test_index]
  y_train = y.iloc[:test_index]
  X_test = X.iloc[test_index:]
  y_test = y.iloc[test_index:]

  return X_train, X_test, y_train, y_test
```

```
X = features
y = df_feature_pd['Apple_Close']

X_train, X_test, y_train, y_test = timeseries_train_test_split(X, y, test_size=0.15)

rfr = RandomForestRegressor(n_estimators = 1000, random_state = 123)
rfr.fit(X_train, y_train)
prediction = rfr.predict(X_test)

df_1 = pd.DataFrame({'Actual':y_test, 'Predicted':prediction})
display(df_1.plot(grid=True, figsize = (15,8)))


#Create New DataFrame with cotaining delta of all features
df2 = pd.DataFrame(index=df_feature_pd.index.copy())

#Calculating the delta
def delta(featuer_name):
  featuer_name = str(featuer_name)
  out_put_name = str(featuer_name) + '_delta'
  df2[out_put_name] = df_feature_pd[featuer_name].diff().shift(-1)
  return

delta('Apple_Close')
delta('BestBuy')
delta('EU100')
delta('France_Index')
delta('Germany_Index')
delta('HSI')
delta('Japan_N225')
delta('Microsoft')
delta('NSDQ')
delta('IBM')
delta('Intel')
delta('Orcl')
delta('Qualcomm')
delta('Samsung')
delta('Shanghai')
delta('SP500')
delta('AMD')
delta('Amazon')
delta('Oil_price')
delta('Walmart')
delta('return')
delta('log_return')
delta('lag_1D')
delta('lag_1W')
delta('lag_2W')
delta('lag_1M')
delta('lag_4M')
delta('loss')
delta('Mov_Avg_1D')
delta('Mov_Avg_1W')
```

```
delta('Mov_Avg_2W')
delta('Mov_Avg_1M')
delta('Mov_Avg_4M')
delta('prediction_1D')
delta('prediction_1W')
delta('prediction_2W')
delta('prediction_1M')
delta('prediction_4M')
df2.head()

#Drop all labels but keep all features
features_delta = df2.drop(['Apple_Close_delta'], axis = 1).drop(['prediction_1D_delta'], axis = 1).drop(['prediction_1W_delta'], axis = 1).drop(['prediction_2W_delta'], axis = 1).drop(['prediction_1M_delta'], axis = 1).drop(['prediction_4M_delta'], axis = 1)

#Feture Selection
from sklearn.ensemble import RandomForestRegressor
from sklearn.feature_selection import SelectFromModel

X_delta_ms = features_delta
y_delta_ms = df2['Apple_Close_delta']

#Get feature labels
feat_labels = list(features_delta.columns.values)

#Train test split with 90% train and 10% test
X_train_delta_ms, X_test_delta_ms, y_train_delta_ms, y_test_delta_ms = timeseries_train_test_split(X_delta_ms, y_delta_ms, test_size=0.1)
X_test_delta_ms = X_test_delta_ms[:-1]
y_test_delta_ms = y_test_delta_ms[:-1]

rfr_fs = RandomForestRegressor(n_estimators = 100)
rfr_fs.fit(X_train_delta_ms, y_train_delta_ms)

#Create a dataframe cotaining features importance
importances = rfr_fs.feature_importances_
table_FI = {'Importances':importances,'Labels':feat_labels}
df_FI = pd.DataFrame(table_FI)
df_FI.sort_values(by='Importances',ascending=False)
df_FI.head(5)

fig = plt.figure(figsize=(30,15))
plt.bar(feat_labels, rfr_fs.feature_importances_)
display(fig)

#select features with feature importance > 0.01
features_selected = df_FI.loc[df_FI['Importances'] > 0.01]['Labels'].tolist()
features_selected

#Drop features with low feature importance
features_delta_updated = features_delta[features_selected]
features_delta_updated.head(5)

#X as Features and y as target
```

```
X_delta = features_delta_updated
y_delta = df2['Apple_Close_delta']

from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score

#Train test split with 85% train and 15% test
X_train_delta, X_test_delta, y_train_delta, y_test_delta = timeseries_train_test_split(X_delta,y_delta,
test_size=0.15)
X_test_delta = X_test_delta[:-1]
y_test_delta = y_test_delta[:-1]
y_train_delta.head()

rfr_initial = RandomForestRegressor()
rfr_initial.fit(X_train_delta, y_train_delta)
prediction_delta_initial = rfr_initial.predict(X_test_delta)

df_delta_initial = pd.DataFrame({'Actual':y_test_delta, 'Predicted_delta':prediction_delta_initial})
df_delta_initial = pd.merge(df_delta_initial, df_feature_pd['lag_1D'], left_index = True, right_index=True)
df_delta_initial['Predicted_close'] = df_delta_initial['Predicted_delta'] + df_delta_initial['lag_1D']
df_delta_initial = pd.merge(df_delta_initial, df_feature_pd['Apple_Close'], left_index = True, right_index=True)

#sMAPE Evaluation
def smape(A, P):
    return 100/len(A) * np.sum(2 * np.abs(P - A) / (np.abs(A) + np.abs(P)))

print('Root Mean Squared Error:', sqrt(metrics.mean_squared_error(df_delta_initial['Apple_Close'],
df_delta_initial['Predicted_close'])))
print('SMAPE :', smape(df_delta_initial["Apple_Close"], df_delta_initial['Predicted_close']))


#Random Hyperparameter Grid
from sklearn.model_selection import RandomizedSearchCV
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}
print(random_grid)
```

```
# Use the random grid to search for best hyperparameters
rf = RandomForestRegressor()

# Random search of parameters, using 3 fold cross validation
rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid, n_iter = 50, cv = 3,
verbose=2, random_state=123, n_jobs = -1)
# Fit the random search model
rf_random.fit(X_train_delta, y_train_delta)

#Showing best parameters from the random grid search
rf_random.best_params_

#Grid with Cross Validation
from sklearn.model_selection import TimeSeriesSplit
from sklearn.model_selection import GridSearchCV

tscv = TimeSeriesSplit(n_splits=2)
# parameter grid based on the results of random search
param_grid = {
    'bootstrap': [True],
    'max_depth': [20, 60, 100, None],
    'max_features': ['auto'],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [3, 5, 8],
    'n_estimators': [800, 1000, 1200]
}
# Create a based model
rf = RandomForestRegressor()
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                          cv = tscv, n_jobs = -1, verbose = 2)

grid_search.fit(X_train_delta, y_train_delta)
grid_search.best_params_

# Random Forest Regression
#training a random forest regressor using the best parameter from grid search
rfr = RandomForestRegressor(max_depth = 100,
 max_features = 'auto', min_samples_leaf = 3,min_samples_split = 8, n_estimators = 800, random_state = 123)
rfr.fit(X_train_delta, y_train_delta)
prediction_delta = rfr.predict(X_test_delta)

#Creating dataframe containing teh actual delta of close price and predicted delta of close price
df_delta = pd.DataFrame({'Actual':y_test_delta, 'Predicted_delta':prediction_delta})
df_delta.head(5)

#Calculating the predicting close price using delta
df_delta = pd.merge(df_delta, df_feature_pd['lag_1D'], left_index = True, right_index=True)
df_delta['Predicted_close'] = df_delta['Predicted_delta'] + df_delta['lag_1D']
df_delta = pd.merge(df_delta, df_feature_pd['Apple_Close'], left_index = True, right_index=True)

#plotting predicted close price against actual close price
display(df_delta.plot(y=['Predicted_close', 'Apple_Close'], figsize=(15,8), grid=True))
```

```
#Print sMAPE
print('1 Day Prediction Errors')
print('Root Mean Squared Error:', sqrt(metrics.mean_squared_error(df_delta['Apple_Close'],
df_delta['Predicted_close'])))
print('SMAPE :', smape(df_delta["Apple_Close"], df_delta['Predicted_close']))

X_delta_1W = features_delta
y_delta_1W = df2['prediction_1W_delta']

X_train_delta, X_test_delta, y_train_delta, y_test_delta = timeseries_train_test_split(X_delta_1W, y_delta_1W,
test_size=0.15)
X_test_delta = X_test_delta[:-1]
y_test_delta = y_test_delta[:-1]

rfr = RandomForestRegressor(max_depth = 100,
 max_features = 'auto', min_samples_leaf = 3,min_samples_split = 8, n_estimators = 800, random_state = 123)
rfr.fit(X_train_delta, y_train_delta)
prediction_delta = rfr.predict(X_test_delta)

df_delta_1W = pd.DataFrame({'Actual':y_test_delta, 'Predicted_delta':prediction_delta})
#display(df_delta.plot(grid=True, figsize = (15,8)))

df_delta_1W = pd.merge(df_delta_1W, df_feature_pd['lag_1D'], left_index = True, right_index=True)
df_delta_1W['Predicted_close_1W'] = df_delta_1W['Predicted_delta'] + df_delta_1W['lag_1D']
df_delta_1W = pd.merge(df_delta_1W, df_feature_pd['prediction_1W'], left_index = True, right_index=True)

#display(df_delta.plot(y=['Predicted_close_1W', 'prediction_1W'], figsize=(15,8), grid=True))

print('1 Week Prediction Errors')
print('Root Mean Squared Error:', sqrt(metrics.mean_squared_error(df_delta_1W['prediction_1W'],
df_delta_1W['Predicted_close_1W'])))
print('SMAPE :', smape(df_delta_1W["prediction_1W"], df_delta_1W['Predicted_close_1W']))

def error(target, df):

  target_delta = str(target)+'_delta'
  X_delta = features_delta
  y_delta = df2[target_delta]

  X_train_delta, X_test_delta, y_train_delta, y_test_delta = timeseries_train_test_split(X_delta,y_delta,
test_size=0.15)
  X_test_delta = X_test_delta[:-1]
  y_test_delta = y_test_delta[:-1]

  rfr = RandomForestRegressor(max_depth = 100,max_features = 'auto', min_samples_leaf = 3,min_samples_split =
8, n_estimators = 800, random_state = 123)
  rfr.fit(X_train_delta, y_train_delta)
  prediction_delta = rfr.predict(X_test_delta)

  df = pd.DataFrame({'Actual':y_test_delta, 'Predicted_delta':prediction_delta})
  df = pd.merge(df, df_feature_pd['lag_1D'], left_index = True, right_index=True)
  df['Predicted_close'] = df['Predicted_delta'] + df['lag_1D']
  df = pd.merge(df, df_feature_pd[target], left_index = True, right_index=True)
```

```
  print(target+' Errors')
  print('Root Mean Squared Error:', sqrt(metrics.mean_squared_error(df[target], df['Predicted_close'])))
  print('SMAPE :', smape(df[target], df['Predicted_close']))

df_1w_pred = pd.DataFrame()
error('prediction_1W', df_1w_pred)

df_2w_pred = pd.DataFrame()
error('prediction_2W', df_2w_pred)

df_1m_pred = pd.DataFrame()
error('prediction_1M', df_1m_pred)

df_4m_pred = pd.DataFrame()
error('prediction_4M', df_4m_pred)
```

	Importances	Labels
0	0.012784	BestBuy_delta
1	0.007210	EU100_delta
2	0.005831	France_Index_delta
3	0.008427	Germany_Index_delta
4	0.009325	HSI_delta

Table E: Features Coefficient for Random Forest Regression Baseline Model

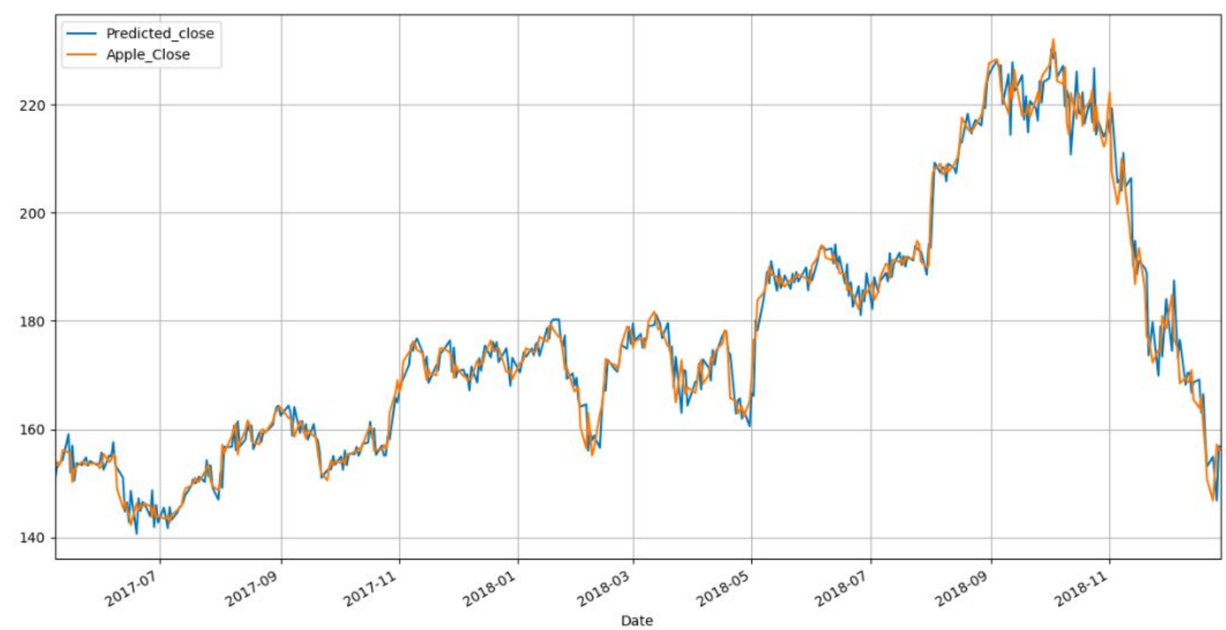

Figure E: Prediction Tendency for the Random Forest Regression model

Appendix F: XGBoost Model Implementation

XGBoost Code Link:
https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/694794198277716/1836567722325543/4949361027789338/latest.html

```
# Library Import
import math
import matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
import time
import datetime as dt
from matplotlib import pyplot as plt

from pyspark.sql import SQLContext, Window
from pyspark.sql.functions import *
from pyspark.ml.regression import LinearRegression
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import RegressionEvaluator
from  pyspark.sql.functions import abs, sqrt

from pyspark.ml.tuning import CrossValidator
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import OneHotEncoderEstimator
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml import Pipeline
from sklearn.model_selection import cross_val_predict

from sklearn.model_selection import train_test_split
from sklearn import metrics
from math import sqrt

from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import TimeSeriesSplit
from sklearn.model_selection import GridSearchCV
from sklearn.feature_selection import SelectFromModel
from sklearn.metrics import r2_score, median_absolute_error, mean_absolute_error
from sklearn.metrics import median_absolute_error, mean_squared_error, mean_squared_log_error

!pip install xgboost
from xgboost.sklearn import XGBRegressor

# additional features
sqlContext = SQLContext(sc)
df_apple = sqlContext.sql("SELECT Date, Close as Apple_Close FROM apple")
```

```
df_sp500 = sqlContext.sql("SELECT Date, Close as SP500 FROM sp500")
df_nsdq = sqlContext.sql("SELECT Date, Close as NSDQ FROM nsdq")
df_amazon = sqlContext.sql("SELECT Date, Close as Amazon FROM amazon")
df_amd = sqlContext.sql("SELECT Date, Close as AMD FROM amd_csv")
df_eu100 = sqlContext.sql("SELECT Date, Close as EU100 FROM eu_n100")
df_fr = sqlContext.sql("SELECT Date, Close as France_Index FROM france")
df_ger = sqlContext.sql("SELECT Date, Close as Germany_Index FROM germany")
df_hk = sqlContext.sql("SELECT Date, Close as HSI FROM hongkong_hsi")
df_jp = sqlContext.sql("SELECT Date, Close as Japan_N225 FROM japan_n225")
df_sh = sqlContext.sql("SELECT Date, Close as Shanghai FROM shanghai")
df_ibm = sqlContext.sql("SELECT Date, Close as IBM FROM ibm_csv")
df_intel = sqlContext.sql("SELECT Date, Close as Intel FROM intel")
df_ms = sqlContext.sql("SELECT Date, Close as Microsoft FROM microsoft")
df_orcl = sqlContext.sql("SELECT Date, Close as Orcl FROM orcl_csv")
df_qual = sqlContext.sql("SELECT Date, Close as Qualcomm FROM qualcomm")
df_sam = sqlContext.sql("SELECT Date, Close as Samsung FROM samsung")
df_wal = sqlContext.sql("SELECT Date, Close as Walmart FROM wmt_csv")
df_bb = sqlContext.sql("SELECT Date, Close as BestBuy FROM bby_bestbuy")
df_oil = sqlContext.sql("SELECT Date, DCOILWTICO as Oil_price FROM oilwtico_csv")

df_feature_index = df_apple.join(df_bb, "Date", how = 'left').join(df_eu100, "Date", how = 'left').join(df_fr, "Date",
how = 'left').join(df_ger, "Date", how = 'left').join(df_hk, "Date", how = 'left').join(df_jp, "Date", how =
'left').join(df_ms, "Date", how = 'left').join(df_nsdq, "Date", how = 'left').join(df_ibm, "Date", how =
'left').join(df_intel, "Date", how = 'left').join(df_orcl, "Date", how = 'left').join(df_qual, "Date", how =
'left').join(df_sam, "Date", how = 'left').join(df_sh, "Date", how = 'left').join(df_sp500, "Date", how =
'left').join(df_amd, "Date", how = 'left').join(df_amazon, "Date", how = 'left').join(df_oil, "Date", how =
'left').join(df_wal, "Date", how = 'left')

df_feature_pd = df_feature_index.toPandas()
df_feature_pd.replace(to_replace=[None], value=np.nan, inplace=True)

df_feature_pd['return'] = df_feature_pd['Apple_Close']/df_feature_pd['Apple_Close'].shift(1) - 1
df_feature_pd['return'] = df_feature_pd['return'].shift(1)
df_feature_pd['abs_ret'] = df_feature_pd['return'].abs()

# calculate the log return on Adj. Close between records
df_feature_pd['log_return'] = np.log(df_feature_pd['Apple_Close']) - np.log(df_feature_pd['Apple_Close'].shift(1))
df_feature_pd['log_return'] = df_feature_pd['log_return'].shift(1)
df_feature_pd['abs_log_ret'] = df_feature_pd['log_return'].abs()

df_feature_pd['lag_1D'] = df_feature_pd['Apple_Close'].shift(1)
df_feature_pd['lag_1W'] = df_feature_pd['Apple_Close'].shift(5)
df_feature_pd['lag_2W'] = df_feature_pd['Apple_Close'].shift(10)
df_feature_pd['lag_1M'] = df_feature_pd['Apple_Close'].shift(26)
df_feature_pd['lag_4M'] = df_feature_pd['Apple_Close'].shift(104)

df_feature_pd['loss'] = df_feature_pd['Apple_Close'] - df_feature_pd['Apple_Close'].shift(-1)

ma_c = df_feature_pd["Apple_Close"]
df_matitle = ['Mov_Avg_1D','Mov_Avg_1W','Mov_Avg_2W','Mov_Avg_1M','Mov_Avg_4M']
column = 0
for days in (2,5,10,20,80):
  mov_a = []
  i = 0
```

```
  sum_a = 0
  for rows in ma_c:
    if i < days:
      mov_a.append(ma_c[i])
      i=i+1
    else:
      for j in range(days):
        sum_a = sum_a + ma_c[i-j-1]
      avg = sum_a/days
      mov_a.append(avg)
      sum_a = 0
      i=i+1
  df_feature_pd[df_matitle[column]] = pd.Series(mov_a)
  column = column + 1

df_feature_pd['prediction_1D'] = df_feature_pd['Apple_Close'].shift(-1)
df_feature_pd['prediction_1W'] = df_feature_pd['Apple_Close'].shift(-5)
df_feature_pd['prediction_2W'] = df_feature_pd['Apple_Close'].shift(-10)
df_feature_pd['prediction_1M'] = df_feature_pd['Apple_Close'].shift(-26)
df_feature_pd['prediction_4M'] = df_feature_pd['Apple_Close'].shift(-104)

df_feature_pd.head()

#Fill missing data using linear regression
#If data cant be filled using interpolate, fill it with 0
#Turn all column in dataframe into numeric values except the first column

for col in df_feature_pd.columns[1:]:
    df_feature_pd[col] = pd.to_numeric(df_feature_pd[col], errors='coerce')

#Use linear interpolate the fill NaN data between two existing data
df_feature_pd.interpolate(method='linear',  inplace=True)

#Fill 0 to all other NaN data
df_feature_pd.fillna(0, inplace=True)
df_feature_pd['Date'] = pd.to_datetime(df_feature_pd['Date'], format = '%Y-%m-%d')
df_feature_pd = df_feature_pd[df_feature_pd["Date"]>'1999-01-01']
df_feature_pd.set_index('Date', inplace=True)

df_feature_pd.head()

features = df_feature_pd.drop(['Apple_Close'], axis = 1).drop(['prediction_1D'], axis = 1).drop(['prediction_1W'],
axis = 1).drop(['prediction_2W'], axis = 1).drop(['prediction_1M'], axis = 1).drop(['prediction_4M'], axis = 1)
features.info()

#Perform train-test split with respect to time series structure
def timeseries_train_test_split(x, y, test_size):
    # get the index after which test set starts
    test_index = int(len(x)*(1-test_size))
    x_train = x.iloc[:test_index]
    y_train = y.iloc[:test_index]
    x_test = x.iloc[test_index:]
    y_test = y.iloc[test_index:]
    return x_train, x_test, y_train, y_test
```

```
x = features
y = df_feature_pd['Apple_Close']

x_train, x_test, y_train, y_test = timeseries_train_test_split(x, y, test_size=0.3)

xgb = XGBRegressor(n_estimators = 1000, random_state = 123)
xgb.fit(x_train, y_train)
prediction = xgb.predict(x_test)

df_orig = pd.DataFrame({'Actual':y_test, 'Predicted':prediction})
display(df_orig.plot(grid=True, figsize = (15,8)))

#Create New DataFrame with cotaining delta of all features
df_all = pd.DataFrame(index=df_feature_pd.index.copy())

#Calculate the delta
def delta(featuer_name):
  featuer_name = str(featuer_name)
  out_put_name = str(featuer_name) + '_delta'
  df_all[out_put_name] = df_feature_pd[featuer_name].diff().shift(-1)
  return

delta('Apple_Close')
delta('BestBuy')
delta('EU100')
delta('France_Index')
delta('Germany_Index')
delta('HSI')
delta('Japan_N225')
delta('Microsoft')
delta('NSDQ')
delta('IBM')
delta('Intel')
delta('Orcl')
delta('Qualcomm')
delta('Samsung')
delta('Shanghai')
delta('SP500')
delta('AMD')
delta('Amazon')
delta('Oil_price')
delta('Walmart')
delta('return')
delta('log_return')
delta('lag_1D')
delta('lag_1W')
delta('lag_2W')
delta('lag_1M')
delta('lag_4M')
delta('loss')
delta('Mov_Avg_1D')
delta('Mov_Avg_1W')
delta('Mov_Avg_2W')
```

```
delta('Mov_Avg_1M')
delta('Mov_Avg_4M')
delta('prediction_1D')
delta('prediction_1W')
delta('prediction_2W')
delta('prediction_1M')
delta('prediction_4M')

df_all.head()

#Drop all labels but keep all features
features_delta = df_all.drop(['Apple_Close_delta'], axis = 1).drop(['prediction_1D_delta'], axis =
1).drop(['prediction_1W_delta'], axis = 1).drop(['prediction_2W_delta'], axis = 1).drop(['prediction_1M_delta'], axis
= 1).drop(['prediction_4M_delta'], axis = 1)

x_delta_ms = features_delta
y_delta_ms = df_all['Apple_Close_delta']

#Get feature labels
feat_labels = list(features_delta.columns.values)

#Train test split with 70% train and 30% test
x_train_delta_ms, x_test_delta_ms, y_train_delta_ms, y_test_delta_ms = timeseries_train_test_split(x_delta_ms,
y_delta_ms, test_size=0.3)
x_test_delta_ms = x_test_delta_ms[:-1]
y_test_delta_ms = y_test_delta_ms[:-1]

xgb_fs = XGBRegressor(n_estimators = 500)
xgb_fs.fit(x_train_delta_ms, y_train_delta_ms)

#Create a dataframe cotaining features importance
importances = xgb_fs.feature_importances_
table_fi = {'Importances':importances,'Labels':feat_labels}
df_fi = pd.DataFrame(table_fi)
df_fi.sort_values(by='Importances',ascending=False)
df_fi.head()

fig = plt.figure(figsize=(15,8))
plt.barh(feat_labels, importances)
display(fig)

#select features with feature importance > 0.01
features_selected = df_fi.loc[df_fi['Importances'] > 0.01]['Labels'].tolist()
features_selected

#Drop features with low feature importance
features_delta_updated = features_delta[features_selected]
features_delta_updated.head()

#Set x as Features, y as target
x_delta = features_delta_updated
y_delta = df_all['Apple_Close_delta']

#Train test split with 70% train and 30% test
```

```
x_train_delta, x_test_delta, y_train_delta, y_test_delta = timeseries_train_test_split(x_delta,y_delta, test_size=0.3)
x_test_delta = x_test_delta[:-1]
y_test_delta = y_test_delta[:-1]
y_train_delta.head()
xgb_initial = XGBRegressor()
xgb_initial.fit(x_train_delta, y_train_delta)
prediction_delta_initial = xgb_initial.predict(x_test_delta)
df_delta_initial = pd.DataFrame({'Actual':y_test_delta, 'Predicted_delta':prediction_delta_initial})
df_delta_initial = pd.merge(df_delta_initial, df_feature_pd['lag_1D'], left_index = True, right_index=True)
df_delta_initial['Predicted_close'] = df_delta_initial['Predicted_delta'] + df_delta_initial['lag_1D']
df_delta_initial = pd.merge(df_delta_initial, df_feature_pd['Apple_Close'], left_index = True, right_index=True)

def smape(A, P):
   return 100/len(A) * np.sum(2 * np.abs(P - A) / (np.abs(A) + np.abs(P)))

print('Root Mean Squared Error:', sqrt(metrics.mean_squared_error(df_delta_initial['Apple_Close'],
df_delta_initial['Predicted_close'])))
print('SMAPE :', smape(df_delta_initial["Apple_Close"], df_delta_initial['Predicted_close']))

#The number of boosted trees to fit
n_estimators = [int(x) for x in np.linspace(start = 300, stop = 700, num = 40)]
#The maximum depth of a tree
max_depth = [int(x) for x in np.linspace(5, 9, num = 5)]
#Boosting learning rate (xgb's “eta”)
learning_rate = [0.005, 0.01, 0.02]
#The minimum sum of weights of all observations required in a child
min_child_weight = [1, 2, 3]
#The fraction of observations to be randomly samples for each tree
subsample = [0.4, 0.5, 0.6]
#The fraction of columns to be randomly samples for each tree
colsample_bytree = [0.6, 0.8, 1]
#The subsample ratio of columns for each split, in each level
colsample_bylevel = [0.8, 1]
#The minimum loss reduction required to make a further partition on a leaf node of the tree
gamma = [0, 1]

# Create the random grid
random_grid = {'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
            }
print(random_grid)

# Use the random grid to search for best hyperparameters
xgbr = XGBRegressor()

# Random search of parameters, using 3 fold cross validation
xgbr_random = RandomizedSearchCV(estimator = xgbr, param_distributions = random_grid, n_iter = 50, cv = 3,
verbose=2, random_state=123, n_jobs = -1)
```

```
# Fit the random search model
xgbr_random.fit(x_train_delta, y_train_delta)

#Showing best parameters from the random grid search
xgbr_random.best_params_

tscv = TimeSeriesSplit(n_splits=2)
# parameter grid based on the results of random search
param_grid = {'n_estimators': [371, 471, 571],
        'max_depth': [6, 7, 8],
        'learning_rate': [0.005, 0.01],
        'min_child_weight': [1, 2, 3],
        'subsample': [0.4, 0.5],
        'colsample_bytree': [0.8, 1],
        'colsample_bylevel': [0.8, 1],
        'gamma': [0, 1]
        }
# Create a based model
xgbr = XGBRegressor()
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = xgbr, param_grid = param_grid, cv = tscv, n_jobs = -1, verbose = 2)

grid_search.fit(x_train_delta, y_train_delta)
grid_search.best_params_

#training a random forest regressor using the best parameter from grid search
xgbRegression = XGBRegressor(n_estimators = 571,
        max_depth = 8,
        learning_rate = 0.01,
        min_child_weight = 1,
        subsample = 0.5,
        colsample_bytree = 1,
        colsample_bylevel = 0.8,
        gamma = 0)
xgbRegression.fit(x_train_delta, y_train_delta)
prediction_delta = xgbRegression.predict(x_test_delta)

#Creating dataframe containing teh actual delta of close price and predicted delta of close price
df_delta = pd.DataFrame({'Actual':y_test_delta, 'Predicted_delta':prediction_delta})
df_delta.head()

#Calculate the predicting close price using delta
df_delta = pd.merge(df_delta, df_feature_pd['lag_1D'], left_index = True, right_index=True)
df_delta['Predicted_close'] = df_delta['Predicted_delta'] + df_delta['lag_1D']
df_delta = pd.merge(df_delta, df_feature_pd['Apple_Close'], left_index = True, right_index=True)

#Plot predicted close price against actual close price
display(df_delta.plot(y=['Predicted_close', 'Apple_Close'], figsize=(15,8), grid=True))

print('1 Day Prediction Errors')
print('Root Mean Squared Error:', sqrt(metrics.mean_squared_error(df_delta['Apple_Close'],
df_delta['Predicted_close'])))
print('SMAPE :', smape(df_delta["Apple_Close"], df_delta['Predicted_close']))
```

```
#Define a function to calculate RMSE and sMAPE
def error(target, df):

  target_delta = str(target)+'_delta'
  X_delta = features_delta
  y_delta = df_all[target_delta]

  X_train_delta, X_test_delta, y_train_delta, y_test_delta = timeseries_train_test_split(X_delta,y_delta,
test_size=0.3)
  X_test_delta = X_test_delta[:-1]
  y_test_delta = y_test_delta[:-1]

  xgbRegression = XGBRegressor(n_estimators = 571,
            max_depth = 8,
            learning_rate = 0.01,
            min_child_weight = 1,
            subsample = 0.5,
            colsample_bytree = 1,
            colsample_bylevel = 0.8,
            gamma = 0)
  xgbRegression.fit(x_train_delta, y_train_delta)
  prediction_delta = xgbRegression.predict(x_test_delta)

  df = pd.DataFrame({'Actual':y_test_delta, 'Predicted_delta':prediction_delta})
  df = pd.merge(df, df_feature_pd['lag_1D'], left_index = True, right_index=True)
  df['Predicted_close'] = df['Predicted_delta'] + df['lag_1D']
  df = pd.merge(df, df_feature_pd[target], left_index = True, right_index=True)

  print(target+' Errors')
  print('Root Mean Squared Error:', sqrt(metrics.mean_squared_error(df[target], df['Predicted_close'])))
  print('sMAPE :', smape(df[target], df['Predicted_close']))

df_1w_pred = pd.DataFrame()
error('prediction_1W', df_1w_pred)

df_2w_pred = pd.DataFrame()
error('prediction_2W', df_2w_pred)

df_1m_pred = pd.DataFrame()
error('prediction_1M', df_1m_pred)

df_4m_pred = pd.DataFrame()
error('prediction_4M', df_4m_pred)
```

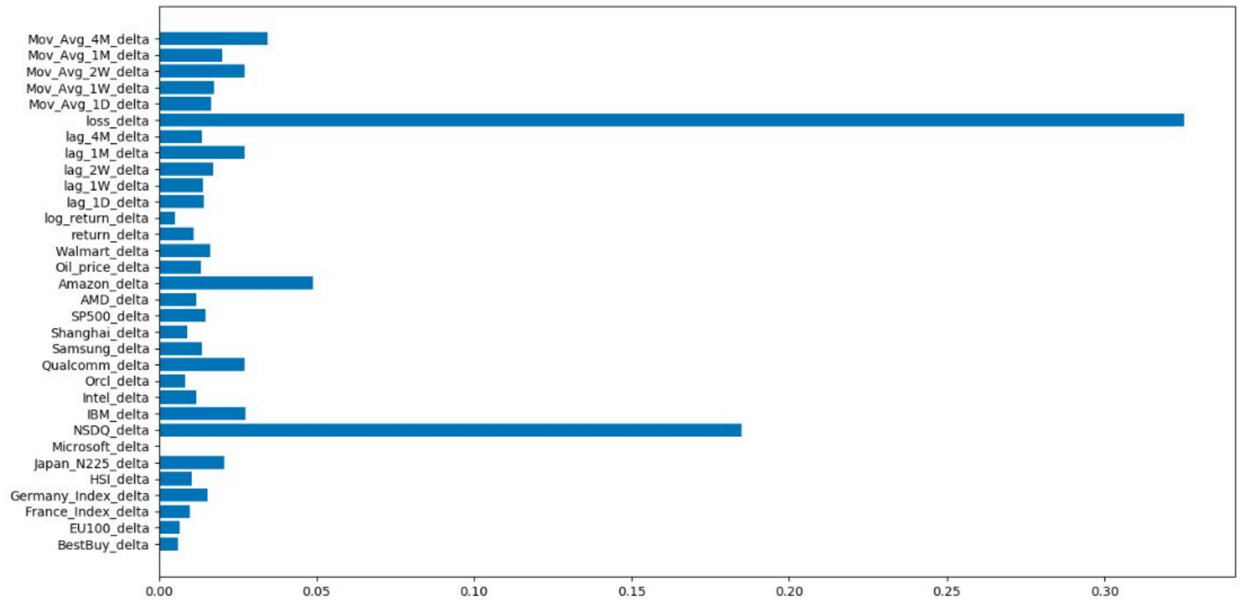

Table F: Features Coefficient for XGBoost Regression Baseline Model

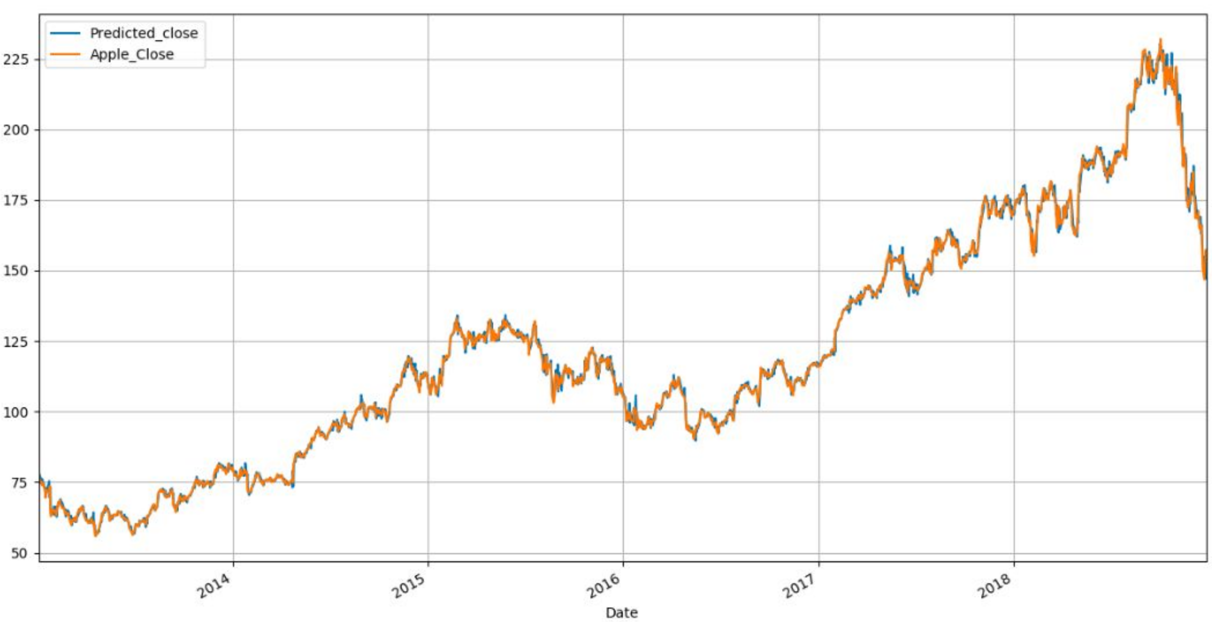

Figure F: Prediction Tendency for the XGBoost Regression model