MIE1628 Final Project Report

Time Series for Stock Price Prediction

Jiaxing Lu

1003564859

June 30, 2019

Contents

[1 Introduction 3](#_Toc12831199)

[2 Methodology 3](#_Toc12831200)

[2.1 Model selection 3](#_Toc12831201)

[2.2 Feature selection 3](#_Toc12831202)

[2.3 Performance verification 4](#_Toc12831203)

[3 Implementation 5](#_Toc12831204)

[3.1 Data preparation 5](#_Toc12831205)

[3.2 Program design 6](#_Toc12831206)

[3.3 Modelling and optimization 6](#_Toc12831207)

[3.3.1 Moving average (SMA) 6](#_Toc12831208)

[3.3.2 Exponential smoothing 6](#_Toc12831209)

[3.3.3 Linear regression 8](#_Toc12831210)

[3.3.4 Decision tree, Random forest and Gradient boosting tree 8](#_Toc12831211)

[4 Result and discussion 9](#_Toc12831212)

[4.1 Result comparison from models 9](#_Toc12831213)

[4.2 Feature and parameter review 10](#_Toc12831214)

[4.3 Implementation challenge and review 11](#_Toc12831215)

[5 Reference for exponential smoothing modelling 12](#_Toc12831216)

[6 Appendix - Code 12](#_Toc12831217)

[6.1 Moving average 12](#_Toc12831218)

[6.2 Exponential smoothing 15](#_Toc12831219)

[6.3 Linear regression 20](#_Toc12831220)

[6.4 Decision tree + Random forest + Gradient boosting tree 25](#_Toc12831221)

# Introduction

The objective of this project is to build and select suitable model to predict the stock price of Apple Inc. As the stock price is a time-paced variable, the problem can be categorized to time series problem. Multiple models are tested and compared for the better performance, and other factors and features will be included in some model to enhance prediction. The author will has implemented the exponential smoothing modelled.

# Methodology

## Model selection

In this project, both statistical models and machine learning models are included showing the development of the time series analysis. There are 5 models are selected with unique reasons.

* **Statistical model**
  + Moving average: One of the classical methods originated in 1920s and widely used until 1950s. It is the foundation of many advanced time series prediction methods.
  + Exponential smoothing: Another conventional method proposed in 1950s and has inspired many successful models. This model distinguished the importance of the past observations along time and can include trend and seasonality into the consideration.
* **Machine learning model**
  + Linear regression: Linear regression is the model to linearize the relationship relate the target variable to time and other features. As linear relation is one of the most basic and common relationship for math models, the multi-variates feature help to distinguish the importance of different features.
  + Decision tree/Random Forest/: Decision tree is another common type of machine learning model and can take continuous values to proceed regression.

## Feature selection

For the machine learning models, other time related data are included as extra features to assist the prediction. There are three categories of features introduced in this project.

* **Lag values of history**

This type of feature includes the historical records of the stock price itself and other related prices. These features are directly related to the target variables. In this project, the Open, Adj Close, Volume, Low, High, Share of AAPL history are considered under this category.

* **Other stock prices**

In this category, stock price of other company and some significant indexes are considered. Microsoft and Google are included as the size and industry can benchmark Apple and can represent the general trend of the industry. The indexes chosen are SP500 and NASDAQ, where both are representative stock market index and indicate the general atmosphere of the stock market and economy.

* **Engineered features**

The engineered features are the processed values from current information. It contains the moving average, log of ratio of lag of Close values, market cap and Z score. In this project, Z score is defined by the difference of the observation and the mean of the observation divided by the standard variance of the dataset. These features are proceeded information or prediction after certain methods that has market or experience proofed, and thus they are reasonable to have correlation with target variable.

## Performance verification

This project used 2 errors between observations and predictions data to justify the performance of the model. Minimizing square mean absolute percentage error will be the key target for the optimization part and model comparison.

* **Root Mean Square Error (RMSE)**

\begin{displaymath}RMS Errors= \sqrt{\frac{\sum_{i=1}^n (\hat{y_i}-y_i)^2}{n}}\end{displaymath}

This error gives the average deviation of the data regardless of the direction of the error, so that the accumulated error will not cancel with each other.

* **Symmetric Mean Absolute Percentage Error (SMAPE)**

A close up of a clock

Description automatically generated

The value gives the sum of every fitted point t and divided again by the number of fitted points n. The range of the error percentage is 0 to 200%. The over-forecasting and under0forcasting will give different SMAPE value.

# Implementation

## Data preparation

A close up of a map

Description automatically generated

Figure 1 - AAPL historical price (1980-2018)

From the historical price plot, it shows that AAPL’s stock price had stayed in very low with little observable variance before 2006. By taking look at the detail price records, it is spotted the price after 2004 are over $2 and start to have larger value and higher variance. Stock prices from 2004 to 2018 are extracted as valid data for this project.

The data should also be verified for null values and other data loss before further process in case of running error. To create data over each horizon, the data points are extracted from the cut data regarding to different horizons. It should be noticed only 5 trading days in a week and 20 for a month, 80 for 4 months for splitting. For the machine learning models, a common 20/80 percentage split is chosen for data training and testing. The random split is avoided to prevent the data leakage.

## Program design

Figure 2 - Program Design Scheme

The chart above illustrates the key flow of the program design for each model build. The structure imports the required libraries and customized input first, and then it prepares the data with horizon splits and training/testing splits. Modelling defined the basic algorithm of the model and analysis part have the RMSE and SMAPE function defined. The optimization will work along with model running until minimized the SMAPE. The textual report can present the required value and the plot can visualize the result for pattern discovery.

## Modelling and optimization

### Moving average (SMA)

The method of Simple Moving Average (SMA) is to select a window (a period) of the data and the next prediction is the mean value of the data over that time window. If the window is set to be 1 day, then the prediction is only one day lag of history data.

### Exponential smoothing

* **Single exponential smoothing (SES)**
* – prediction of next period
* – current observation
* – latest prediction
* – smoothing parameter for the level,

The single exponential smoothing (SES) method indicates that the prediction is the weighted average of the current observation and the latest prediction. As the prediction carries on, the function iterates and contains all past with different orders of . The further the time is from current period, the power is higher and the data has less influence to the prediction.



By varying the smoothing parameter α, the performance of the model can change significantly. The current model can only give the prediction of next period. Without further input, all future value will be the same value, reflecting a flat line on time-price plot.

* **Holt’s linear trend (Double exponential smoothing, DES)**

Forecast equation

Level equation

Trend equation

* – prediction of next period
* – estimate of the level of the series at time t
* - 1 for this case, steps ahead
* – estimate of the trend of the series at time t
* – smoothing parameter for the level,
* - smoothing parameter for the trend,

Like SES, the Holt’s linear trend method has a level function, but it also includes the trend of the function. Trend is the slope of the past level, and it is added to the level function to contribute the one step future. The trend itself is another weighted average of last trend expectation and actual level difference. Though given the step choice h, only 1 period further is reliable. Without further input, the prediction will give a inclined/declined line from the latest data points.

* **Optimization**

The performance of this model is mainly based on the choice of the parameters. As all parameters have a close range between 0 to 1, the performance of the parameter can be justified by changing the value in the range. During the programming, an accuracy/step value is set, and the parameters are tested in a loop by adding steps for parameter and comparing the SMAPE/RMSE. Each new lower SMAPE/RMSE will update the best parameter value.

### Linear regression

A close up of a logo

Description automatically generated

* – variable to be forecasted
* – predictor variables/features
* – coefficient

The multivariate linear regression model is deployed in the model building to seek better prediction performance. By manipulating the coefficients, the equation seeks to find the general minimum difference between prediction and actual with features input. The model building is processed as below:

1. **Linear regression model**: The maximum number of iterations, label and feature columns were defined
2. **Set stages of pipeline:** use PCA to justify the correlation between variate and features. The Vector Indexer used to differentiate between categorical data.
3. **Paramgrid defined:** with regularization to prevent over fitting
4. **Optimization:** K fold cross validation to validate the model in case of the oddity of the training data and reduce the overfitting issue
5. **Validation:** The training data was fit, and predictions were made on the test data

### Decision tree, Random forest and Gradient boosting tree

* **Feature processing**

For tree models, the raw price and other features should be proceeded before being used. The main idea is to take the difference between records and find the change. The reason is that tree model can only reproduced from the learned set, and this can lead to missing increase or decrease. The general trend of the record shows a significant trend of price increase along time. The change in difference however are relatively more stable as the change in price is regulated by the stock market. The selected

* **Model building**

1. Take stock price difference between each period to solve non-stationary issue
2. Splitting the apple stock price in a way to prevent data leakage
3. Choosing appropriate features to train the model
4. Hyperparameter tuning and cross validation
5. Smooth the prediction base on connected time periods

* **Optimization**

Apart from the cross validation, the hyperparameter maxBins and maxDepth are to be tuned to give better model performance. Increasing maxBins allows the algorithm to consider more split candidates and make fine-grained split decisions. Results do not change much in this case, take small values as computational efficiency. maxDepth has to be less than 30, set to about the middle of 30. May cause over fit if the value is large.

# Result and discussion

## Result comparison from models

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **RMSE** | **Daily** | **Weekly** | **Bi-Weekly** | **Monthly** | **Quad-Monthly** |
| ***Moving Average*** | **0.93** | **1.29** | **1.72** | **2.40** | **4.95** |
| ***Single Exponential Smoothing*** | **0.86** | **1.91** | **2.78** | **3.72** | **4.96** |
| ***Holt's Linear Trend*** | **0.86** | **1.91** | **2.77** | **3.67** | **4.72** |
| ***Linear Regression*** | **3.90** | **5.39** | **6.06** | **12.40** | **25.41** |
| ***Decision Trees*** | **3.43** | **7.32** | **10.62** | **14.16** | **23.13** |
| ***Random Forest*** | **3.31** | **7.14** | **10.45** | **12.35** | **20.51** |
| ***Gradient Boosting Trees*** | **3.83** | **7.63** | **10.54** | **12.97** | **20.29** |

Table 1 - RMSE comparison

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SMAPE | Daily | Weekly | Bi-Weekly | Monthly | Quad-Monthly |
| *Moving Average* | **2.02** | **3.02** | **4.08** | **5.76** | **12.65** |
| *Single Exponential Smoothing* | **1.99** | **4.73** | **6.84** | **9.96** | **13.72** |
| *Holt's Linear Trend* | **1.99** | **4.71** | **6.83** | **9.90** | **13.58** |
| *Linear Regression* | **2.92** | **2.58** | **2.92** | **5.93** | **8.69** |
| *Decision Trees* | **1.36** | **2.95** | **4.57** | **5.87** | **10.37** |
| *Random Forest* | **1.33** | **2.94** | **4.46** | **4.46** | **8.89** |
| *Gradient Boosting Trees* | **1.46** | **3.04** | **4.22** | **4.41** | **6.98** |

Table 2 - SMAPE comparison

From the horizon wise, larger time gap gives larger error for all model, and it can be interpreted that longer time span have larger variance and model give more errors.

Comparing the models, the statistical models performed better than machine learning models in RMSE. The reason is the statistical model have only one step to predict. However, in SMAPE, the Gradient Boosting Tree model have the lowest error which indicate it has the best prediction among all models.

## Feature and parameter review

* **Parameters in exponential smoothing**

During the exponential smoothing running, most of the computing cost is used on optimization, where the program is run in loop. By reducing the step size, the parameters will be closer to the most optimized value, but it means more loops and longer time.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Quad-monthly | RMSE | | SMAPE | | Run time |
| SES | DES | SES | DES |
| *Step = 0.1* | **5.20** | **4.83** | **14.56** | **14.35** | **8.67s** |
| *Step = 0.005* | **4.96** | **4.72** | **13.72** | **13.58** | **38.42min** |

Table 3 - Exponential Smoothing Step Size Comparison

From the test above, the step 0.1 will run 100 times and cost 8.67s; the step 0.005 will run 40000 times and cost 38.42min. 265 times extra cost only give 1% reduce in SMAPE and 0.1 reduce in RMSE. This indicates further improvement on accuracy of closing to best optimized parameter is not cost-worthy.

* **Feature importance ranking in tree models**

The build-in function in tree models give the optimized portion of the features for tree models. The weight of the feature reflects the importance, while higher portion refers to higher importance. The ranked important features are shown as below.

|  |  |  |
| --- | --- | --- |
| Feature | Portion | Importance ranking |
| difference in close price lag by 2 period | 0.151553 | 1 |
| changed in difference in adj close price between period 1 and 2 | 0.144938 | 2 |
| changed in difference in sp500 price between period 1 and 2 | 0.120975 | 3 |
| difference in close price lag by 1 period | 0.111037 | 4 |
| changed in difference in nasdaq price between period 1 and 2 | 0.109571 | 5 |
| changed in difference in high price between period 1 and 2 | 0.100013 | 6 |
| changed in difference in close price between period 1 and 2 | 0.089546 | 7 |
| changed in difference in volume between period 1 and 2 | 0.069527 | 8 |
| changed in difference in open price between period 1 and 2 | 0.053959 | 9 |
| changed in difference in low price between period 1 and 2 | 0.048881 | 10 |

Table 4 - Feature importance ranking for tree models

## Implementation challenge and review

* **Limitation of Scala language**

During the time of implementing the model with Scala, the Scala shows its power of faster processing speed. However, when it comes to the packages to extend the functionality, the options for Scala is very limited. This becomes an obstacle when try to make plot of the variables, and no packages could be found. Instead, the data was ported back to python and visualized by pyplot.

* **Feature engineering for tree model**

The limitation of the tree models are the values must be provided previously. Therefore, when first tried raw price with raw features, the model gave high SMAPE and RMSE. This motivated the idea to the ‘change in difference’.

* **Missing spike for 4-month data set**

For the longer horizon data, there is the feed-in test data miss the spike between selected points. This led to the missing high value predictions during the 4-month horizon test. Therefore, the shorter horizon is recommended for higher accuracy of the prediction with tree models.

A close up of a map

Description automatically generated

Figure 3 - Missing Spike

# Reference for exponential smoothing modelling

* Time Series Forecasting Performance Measures With Python. (2019, June 28). Retrieved from <https://machinelearningmastery.com/time-series-forecasting-performance-measures-with-python/>
* Zhang, A. L. (2018, September 21). How to Build Exponential Smoothing Models Using Python: Simple Exponential Smoothing, Holt, and... Retrieved June 30, 2019, from <https://medium.com/datadriveninvestor/how-to-build-exponential-smoothing-models-using-python-simple-exponential-smoothing-holt-and-da371189e1a1>
* Forecasting: Principles and Practice. (n.d.). Retrieved June 30, 2019, from <https://otexts.com/fpp2/expsmooth.html>

# Appendix - Code

## Moving average

//Import Libraries

**import** **pandas** **as** **pd**

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

**import** **time**

**from** **pandas** **import** Series

**from** **numpy** **import** mean

**from** **math** **import** sqrt

**from** **sklearn.metrics** **import** mean\_squared\_error

*#from ts.flint import FlintContext*

**from** **pyspark.sql** **import** Window

**from** **pyspark.sql** **import** functions **as** F

**from** **pyspark.sql.types** **import** StructType

*#from ts.flint import windows*

*#flintContext = FlintContext(sqlContext)*

**from** **pyspark.ml.feature** **import** VectorAssembler

**from** **pyspark.ml.regression** **import** LinearRegression

**from** **pyspark.sql.types** **import** DoubleType, IntegerType, StringType

//Read **in** Data

df = spark.read.csv('FileStore/tables/AAPL.csv', inferSchema=**True**, header=**True**, sep=',', nullValue='')

//Drop Columns do **not** need

df = df.drop('Open', 'High', 'Low', 'Adj Close', 'Volume')

df.show(5)

df.toPandas()

//Get values **from** **the** dataframe

df1=df.toPandas()

series\_time = df1["Close"].values

df = df.withColumn('Close',df['Close'].cast('float'))

df.printSchema()

//calculate the moving average can created a dataframe

*#window\_0\_days = Window.rowsBetween(0, 0)*

window\_2\_days = Window.rowsBetween(-1, 0)

window\_5\_days = Window.rowsBetween(-5, 0)

window\_10\_days = Window.rowsBetween(-10, 0)

window\_30\_days = Window.rowsBetween(-30, 0)

window\_120\_days = Window.rowsBetween(-120, 0)

moving\_avg\_2 = F.avg(df['Close']).over(window\_2\_days)

moving\_avg\_5 = F.avg(df['Close']).over(window\_5\_days)

moving\_avg\_10 = F.avg(df['Close']).over(window\_10\_days)

moving\_avg\_30 = F.avg(df['Close']).over(window\_30\_days)

moving\_avg\_120 = F.avg(df['Close']).over(window\_120\_days)

*#df = df.withColumn('moving\_avg\_0\_day', moving\_avg\_0)*

df = df.withColumn('moving\_avg\_2\_day', moving\_avg\_2)

df = df.withColumn('moving\_avg\_5\_day', moving\_avg\_5)

df = df.withColumn('moving\_avg\_10\_day', moving\_avg\_10)

df = df.withColumn('moving\_avg\_30\_day', moving\_avg\_30)

df = df.withColumn('moving\_avg\_120\_day', moving\_avg\_120)

df.show(20)

//Define sMAPE function

**def** smape(y\_true, y\_pred):

denominator = (np.abs(y\_true) + np.abs(y\_pred)) / 200.0

diff = np.abs(np.array(y\_true) - np.array(y\_pred)) / denominator

diff[denominator == 0] = 0.0

**return** np.nanmean(diff)

X=series\_time

*# This is the window of how many days of moving avg.*

*# Pick 1, 5, 10, 20, 80 for 1 day Naive, 1-week, 2-week, 1-month, 4-month*

window = 1

history = [float(X[i]) **for** i **in** range(window)]

test = []

*# test = [X[i] for i in range(window, len(X))]*

**for** i **in** range(window, len(X)):

**if**(X[i] != "null"):

test.append(float(X[i]))

predictions = list()

*# walk forward over time steps in test*

**for** t **in** range(len(test)):

*#length depends on how many days you want to predict after the current days.*

length = len(history)

*#length = len(history) - window*

yhat = mean([history[i] **for** i **in** range(length-window,length)])

**if** test[t] != "null":

obs = float(test[t])

predictions.append(yhat)

history.append(obs)

*# MASE ERROR*

mase\_error = mean\_squared\_error(test, predictions)

print('Test MSE: %.3f' % mase\_error)

*# RMASE ERROR*

root\_mase\_error = sqrt(mean\_squared\_error(test, predictions))

print('Test RMSE: %.3f' % root\_mase\_error)

*# SMAPE ERROR*

smape\_error = smape(test, predictions)

print('Test SMAPE: %.3f' % smape\_error)

*# plot*

plt.plot(test)

plt.plot(predictions, color='red')

plt.show()

*# zoom plot*

plt.plot(test[0:100])

plt.plot(predictions[0:100], color='red')

plt.show()

vectorAssembler = VectorAssembler(inputCols = ['moving\_avg\_5\_day'], outputCol = 'features')

df\_T = vectorAssembler.transform(df)

df\_T = df\_T.select(['features', 'Close'])

df\_T.show(5)

splits = df\_T.randomSplit([0.8, 0.2])

train\_df = splits[0]

test\_df = splits[1]

lr = LinearRegression(featuresCol = 'features', labelCol='Close', maxIter=10, regParam=0.3, elasticNetParam=0.8)

lr\_model = lr.fit(train\_df)

print("Coefficients: " + str(lr\_model.coefficients))

print("Intercept: " + str(lr\_model.intercept))

trainingSummary = lr\_model.summary

print("RMSE: %f" % trainingSummary.rootMeanSquaredError)

print("r2: %f" % trainingSummary.r2)

lr\_predictions = lr\_model.transform(test\_df)

lr\_predictions.select("prediction","Close","features").show(5)

**from** **pyspark.ml.evaluation** **import** RegressionEvaluator

lr\_evaluator = RegressionEvaluator(predictionCol="prediction", \

labelCol="Close",metricName="r2")

print("R Squared (R2) on test data = %g" % lr\_evaluator.evaluate(lr\_predictions))

test\_result = lr\_model.evaluate(test\_df)

print("Root Mean Squared Error (RMSE) on test data = %g" % test\_result.rootMeanSquaredError)

## Exponential smoothing

*# --- Import tools/libaries --- #*

*# Python libararies*

**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

**import** **matplotlib.pyplot** **as** **plt**

**import** **array**

**import** **time**

**from** **pandas** **import** Series

**from** **numpy** **import** mean

**from** **math** **import** sqrt

**from** **sklearn.metrics** **import** mean\_squared\_error

*# Spark libararies*

**from** **pyspark.sql.types** **import** DoubleType *# Used to change data type*

**import** **pyspark.sql.functions** **as** **F**

**import** **pyspark.sql.window** **as** **W**

*# --- Programme Input --- #*

*# the address of the time series record of the stock*

file\_address = '/FileStore/tables/AAPL.csv'

*# how accurate the factor optimization will be*

opt\_accuracy = 0.005

*# --- Data preparation --- #*

*# import data and select the desired data*

raw\_data = spark.read.format('csv').options(header='true', inferSchema='true').load(file\_address).drop('Open', 'High', 'Low', 'Adj Close', 'Volume')

raw\_data.show(5)

*# verify if there is null in the data and drop the null row*

raw\_data.count()

raw\_data.filter(raw\_data.Close == "null").count()

raw\_data1 = raw\_data.filter(raw\_data.Close != "null")

raw\_data1.count()

*# change the close price from string to double*

df = raw\_data1.withColumn("Close",raw\_data1["Close"].cast(DoubleType()))

df.printSchema()

*# convert to pandas DataFrame for python use*

df1 = df.toPandas()

ts1 = df1["Close"].values

*# --- Data over varied horizon --- #*

*# Function to create new time series according to required horizon*

**def** SeriesGenerator(ts,period):

new\_ts = np.array([])

**for** i **in** range(0,len(ts),period):

**if** i + period <= len(ts):

new\_ts = np.append(new\_ts,ts[i])

**else**:

**break**

**return** new\_ts

*# Time serieses over varied horizon*

ts\_daily = SeriesGenerator(ts1,1)

ts\_weekly = SeriesGenerator(ts1,5)

ts\_biweekly = SeriesGenerator(ts1,10)

ts\_monthly = SeriesGenerator(ts1,20)

ts\_quamonthly = SeriesGenerator(ts1,40)

*# --- Modelling --- #*

*# Single Exponential Smoothing Model*

**def** SES(obs,alpha):

*# Set the alpha to tune the model. Optimization could be improved by varying the values. 0.1 is a relatively proper value.*

a = alpha

*# prepare the list to store predictions*

pred = list()

pred.append(obs[0]) *# fill in the first element as no inital prediction, same as observation*

*# Single Exponential Smoothing method deployed*

**for** i **in** range(1,len(obs),1):

predx = a \* obs[i-1] + (1-a) \* pred[i-1]

pred.append(predx)

**return** pred;

*# Holt's Linear Trend Model (aka double exponential method)*

**def** DES(obs, alpha, beta):

*# prepare the values*

*# trend*

trd = list()

trd.append(0)

*# level*

lvl = list()

lvl.append(obs[0])

*# prediction: fill in the first two elements as no inital prediction, same as observation*

pred = list()

pred.append(obs[0])

pred.append(obs[0])

*# Holt't Linear Trend method deployed*

**for** i **in** range(2,len(obs),1):

lvlx = alpha \* obs[i-1] + (1 - alpha) \* (lvl[i-2] + trd[i-2])

lvl.append(lvlx)

trdx = beta \* (lvl[i-1] - lvl[i-2]) + (1 - beta) \* trd[i-2]

trd.append(trdx)

predx = lvl[i-1] + trd[i-1]

pred.append(predx)

**return** pred;

*# --- Analysis --- #*

*# RMSE(Root Mean Squared Error) define*

**def** RMSE(observations, predictions):

mse = mean\_squared\_error(observations, predictions)

rmse = sqrt(mse)

**return** rmse;

*# SMAPE(Symmetric mean absolute percentage error) define*

**def** SMAPE(observations, predictions):

smape = 100/len(observations) \* np.sum(2 \* np.abs(predictions - observations) / (np.abs(observations) + np.abs(predictions))) *# in %*

**return** smape;

*# --- Optimization --- #*

*# find the minimum RMSE/SMAPE by varying a/b*

**def** MinSelection\_SES\_RMSE(obs,accuracy):

best\_result = RMSE(obs,SES(obs,accuracy))

best\_accuracy = accuracy

rng = int(1/accuracy-1)

**for** i **in** range(1,rng,1):

result = RMSE(obs,SES(obs,i\*accuracy))

**if** result < best\_result:

best\_result = result

best\_a = i\*accuracy

best\_pred = SES(obs,best\_a)

**return** [best\_result,best\_pred,best\_a]

**def** MinSelection\_SES\_SMAPE(obs,accuracy):

best\_result = SMAPE(obs,SES(obs,accuracy))

best\_accuracy = accuracy

rng = int(1/accuracy-1)

**for** i **in** range(1,rng,1):

result = SMAPE(obs,SES(obs,i\*accuracy))

**if** result < best\_result:

best\_result = result

best\_a = i\*accuracy

best\_pred = SES(obs,best\_a)

**return** [best\_result,best\_pred,best\_a]

**def** MinSelection\_DES\_RMSE(obs,accuracy):

best\_a = accuracy

best\_b = accuracy

best\_result = RMSE(obs,DES(obs,best\_a,best\_b))

rng = int(1/accuracy-1)

**for** i **in** range(1,rng,1):

**for** j **in** range(1,rng,1):

result = RMSE(obs,DES(obs,i\*accuracy,j\*accuracy))

**if** result < best\_result:

best\_result = result

best\_a = i\*accuracy

best\_b = j\*accuracy

best\_pred = DES(obs,best\_a,best\_b)

**return** [best\_result,best\_pred,best\_a,best\_b]

**def** MinSelection\_DES\_SMAPE(obs,accuracy):

best\_a = accuracy

best\_b = accuracy

best\_result = SMAPE(obs,DES(obs,best\_a,best\_b))

rng = int(1/accuracy-1)

**for** i **in** range(1,rng,1):

**for** j **in** range(1,rng,1):

result = SMAPE(obs,DES(obs,i\*accuracy,j\*accuracy))

**if** result < best\_result:

best\_result = result

best\_a = i\*accuracy

best\_b = j\*accuracy

best\_pred = DES(obs,best\_a,best\_b)

**return** [best\_result,best\_pred,best\_a,best\_b]

*# --- Result Computation and Presentation Tool --- #*

*# Calculate the result and present*

**def** report(obs,accuracy,txt):

min\_SES\_RMSE = MinSelection\_SES\_RMSE(obs,accuracy)

min\_SES\_SMAPE = MinSelection\_SES\_SMAPE(obs,accuracy)

min\_DES\_RMSE = MinSelection\_DES\_RMSE(obs,accuracy)

min\_DES\_SMAPE = MinSelection\_DES\_SMAPE(obs,accuracy)

**print**("Over the ",txt," Horizon")

**print**("With Simple Exponential Smoothing Model")

**print**("The optimized RMSE = ",min\_SES\_RMSE[0],", while factor alpha = ",min\_SES\_RMSE[2])

**print**("The optimized SMAPE = ",min\_SES\_SMAPE[0],", while factor alpha = ",min\_SES\_SMAPE[2])

**print**("With Holt't Linear Trend Model")

**print**("The optimized RMSE = ",min\_DES\_RMSE[0],", while factor alpha = ",min\_DES\_RMSE[2]," and beta = ",min\_DES\_RMSE[3])

**print**("The optimized SMAPE = ",min\_DES\_SMAPE[0],", while factor alpha = ",min\_DES\_SMAPE[2]," and beta = ",min\_DES\_SMAPE[3])

**print**()

**return** [obs,min\_SES\_SMAPE[1],min\_DES\_SMAPE[1]]

*# Visualize the result*

**def** viz(obs,ses,des,txt):

x = plt.figure() *# this line helps to split the relationship between figures*

plt.subplot(2, 2, 1)

plt.plot(obs, color='black')

plt.plot(ses, color='magenta')

plt.title('Single Exponential Smoothing')

plt.ylabel('price')

plt.subplot(2, 2, 3)

plt.plot(obs, color='black',label='Actual')

plt.plot(des, color='magenta',label='Prediction')

plt.title('Holt Linear Trend')

plt.xlabel('time'+'('+txt+')')

plt.ylabel('price')

plt.subplot(2, 2, 2)

plt.plot(obs[-round(len(obs)//10):], color='black',label='Actual')

plt.plot(ses[-round(len(obs)//10):], color='magenta',label='Prediction')

plt.title('SES Zoom-in')

plt.subplot(2, 2, 4)

plt.plot(obs[-round(len(obs)//10):], color='black',label='Actual')

plt.plot(des[-round(len(obs)//10):], color='magenta',label='Prediction')

plt.title('HLT Zoom-in')

plt.xlabel('time'+'('+txt+')')

display()

*# --- Result Presentation --- #*

daily = report(ts\_daily,opt\_accuracy,'Daily')

weekly = report(ts\_weekly,opt\_accuracy,'Weekly')

biweekly = report(ts\_biweekly,opt\_accuracy,'Bi-Weekly')

monthly = report(ts\_monthly,opt\_accuracy,'Monthly')

quamonthly = report(ts\_quamonthly,opt\_accuracy,'4-Month')

*# --- Visualized Results --- #*

*# Daily*

viz(daily[0],daily[1],daily[2],'Day')

*# Weekly*

viz(weekly[0],weekly[1],weekly[2],'Week')

*# Bi-Weekly*

viz(biweekly[0],biweekly[1],biweekly[2],'Bi-Week')

*# Monthly*

viz(monthly[0],monthly[1],monthly[2],'Month')

*# Quad-Monthly*

viz(quamonthly[0],quamonthly[1],quamonthly[2],'4-Month')

## Linear regression

*// Import libraries*

**import** **org.apache.spark.ml.Pipeline**

**import** **org.apache.spark.ml.evaluation.RegressionEvaluator**

**import** **org.apache.spark.ml.feature.VectorIndexer**

**import** **org.apache.spark.ml.feature.Interaction**

**import** **org.apache.spark.ml.feature.VectorAssembler**

**import** **org.apache.spark.ml.linalg.Vector**

**import** **org.apache.spark.sql.Row**

**import** **org.apache.spark.sql.expressions.Window**

**import** **scala.math.\_**

**import** **org.apache.spark.sql.functions.\_**

**import** **org.apache.spark.sql.types.\_**

**import** **org.apache.spark.ml.tuning.**{**CrossValidator**, **ParamGridBuilder**}

**import** **org.apache.spark.sql.functions.\_**

**import** **org.apache.spark.ml.feature.PCA**

*//set project to use default database in databricks*

sqlContext.sql("use default")

*// Load data tables*

*//apple stock data*

**var** training **=** spark.table("aapl\_train\_csv")

**var** testing **=** spark.table("aapl\_test\_csv")

*//SP500 index data*

**val** sp500 **=** spark.table("sp500\_csv")

*//NASDAQ index data*

**val** msft **=** spark.table("msft\_csv")

**val** goog **=** spark.table("goog\_csv")

*//joining apple data with SP500 data using date*

training **=** training.join(sp500, training.col("Date") === sp500.col("sp500Date"))

testing **=** testing.join(sp500, testing.col("Date") === sp500.col("sp500Date"))

*//joing apple data with NASDAQ data using date*

training **=** training.join(msft, training.col("Date") === msft.col("msftDate"))

testing **=** testing.join(msft, testing.col("Date") === msft.col("msftDate"))

*//joing apple data with NASDAQ data using date*

training **=** training.join(goog, training.col("Date") === goog.col("googDate"))

testing **=** testing.join(goog, testing.col("Date") === goog.col("googDate"))

*//create a partition window base on date*

**val** partitionwindow **=** **Window**.orderBy("date")

*//create a lag 1 time period of apple close price variable*

**var** lagtest **=** lag("close",84,0).over(partitionwindow)

*//creata a new column with lag 1 time period close price*

**var** training1 **=** training.withColumn("lag1",lagtest)

*//All the features imported into the model must be at least 1 time period lagged or else there is leaking the future in the results*

*//create a lag 1 time period of apple open price variable*

lagtest **=** lag("open",84,0).over(partitionwindow)

training1 **=** training1.withColumn("open1",lagtest)

*//create a lag 1 time period of apple high price variable*

lagtest **=** lag("high",84,0).over(partitionwindow)

training1 **=** training1.withColumn("high1",lagtest)

*//create a lag 1 time period of apple low price variable*

lagtest **=** lag("low",84,0).over(partitionwindow)

training1 **=** training1.withColumn("low1",lagtest)

*//create a lag 1 time period of apple adjusted close price variable*

lagtest **=** lag("adj close",84,0).over(partitionwindow)

training1 **=** training1.withColumn("adj close1",lagtest)

lagtest **=** lag("volume",84,0).over(partitionwindow)

*//create a lag 1 time period of apple trade volume variable*

training1 **=** training1.withColumn("volume1",lagtest)

lagtest **=** lag("share",84,0).over(partitionwindow)

*//create a lag 1 time period of apple number of shares variable*

training1 **=** training1.withColumn("share1",lagtest)

*//create a lag 1 time period of SP500 close price variable*

lagtest **=** lag("sp500Close",84,0).over(partitionwindow)

training1 **=** training1.withColumn("sp500Close1",lagtest)

*//create a lag 1 time period of NASDAQ close price variable*

lagtest **=** lag("msftClose",84,0).over(partitionwindow)

training1 **=** training1.withColumn("msftClose1",lagtest)

lagtest **=** lag("googClose",84,0).over(partitionwindow)

training1 **=** training1.withColumn("googClose1",lagtest)

*//creating a 5 period of look back window*

*//create a lag 2 time period of apple close price variable*

lagtest **=** lag("close",168,0).over(partitionwindow)

**var** training2 **=** training1.withColumn("lag2",lagtest)

*//create a lag 3 time period of apple close price variable*

lagtest **=** lag("close",252,0).over(partitionwindow)

training2 **=** training2.withColumn("lag3",lagtest)

*//create a lag 4 time period of apple close price variable*

lagtest **=** lag("close",336,0).over(partitionwindow)

training2 **=** training2.withColumn("lag4",lagtest)

*//create a lag 5 time period of apple close price variable*

lagtest **=** lag("close",420,0).over(partitionwindow)

training2 **=** training2.withColumn("lag5",lagtest)

*//create features as log change between each time period's close price*

*// difference between lag 1 time period and 2 time periods*

training2 **=** training2.withColumn("diff12",log10($"lag1"/$"lag2"))

*// diference between lag 2 and lag 3 time periods*

training2 **=** training2.withColumn("diff23",log10($"lag2"/$"lag3"))

*// diference between lag 3 and lag 4 time periods*

training2 **=** training2.withColumn("diff34",log10($"lag3"/$"lag4"))

*// diference between lag 4 and lag 5 time periods*

training2 **=** training2.withColumn("diff45",log10($"lag4"/$"lag5"))

*//create 1 time period lag of market capitalization*

training2 **=** training2.withColumn("marketcap",($"lag1"\*$"share1"))

*//creating moving average variable as a feature. the look back period is from 11 period ago to 1 period ago*

*//using lag1 as the close price from a period ago and average up to 10 periods before*

training2 **=** training2.withColumn("movingAverage", avg(training2("lag1"))

.over( **Window**.partitionBy("date").rowsBetween(-252,0)) )

training2 **=** training2.withColumn("label",$"close")

*//extracting the only relatve feature columns to a new training dataframe*

**var** training3 **=** training2.select("lag1","lag2","lag3","lag4","lag5","open1","high1","low1","adj close1","volume1","diff12","diff23","diff34","diff45","movingAverage","marketcap","sp500Close1","msftClose1","googClose1","close","label")

*//drop any NAN or null value from the table*

training3 **=** training3.na.drop()

*// create a vector to group all the features into one feature vector*

**val** n **=** training3.select("label").count

**var** t2 **=** training3.select("label").agg(sum("label"))

**var** t3 **=** t2.withColumn("mean", $"sum(label)"/n)

**val** m**=** t3.first().getDouble(1)

println(m)

**var** f1**=** training3.withColumn("Mean", lit(m))

**var** f2**=** f1.withColumn("Std", sqrt(pow($"label"-lit(m), 2)/n))

**var** training4**=** f2.withColumn("Zscore", ($"label"-$"Mean")/$"Std")

lagtest **=** lag("close",84,0).over(partitionwindow)

**var** testing1 **=** testing.withColumn("lag1",lagtest)

*//All the features imported into the model must be at least 1 time period lagged or else there is leaking the future in the results*

*//create a lag 1 time period of apple open price variable*

lagtest **=** lag("open",84,0).over(partitionwindow)

testing1 **=** testing1.withColumn("open1",lagtest)

*//create a lag 1 time period of apple high price variable*

lagtest **=** lag("high",84,0).over(partitionwindow)

testing1 **=** testing1.withColumn("high1",lagtest)

*//create a lag 1 time period of apple low price variable*

lagtest **=** lag("low",84,0).over(partitionwindow)

testing1 **=** testing1.withColumn("low1",lagtest)

*//create a lag 1 time period of apple adjusted close price variable*

lagtest **=** lag("adj close",84,0).over(partitionwindow)

testing1 **=** testing1.withColumn("adj close1",lagtest)

lagtest **=** lag("volume",84,0).over(partitionwindow)

*//create a lag 1 time period of apple trade volume variable*

testing1 **=** testing1.withColumn("volume1",lagtest)

lagtest **=** lag("share",84,0).over(partitionwindow)

*//create a lag 1 time period of apple number of shares variable*

testing1 **=** testing1.withColumn("share1",lagtest)

*//create a lag 1 time period of SP500 close price variable*

lagtest **=** lag("sp500Close",84,0).over(partitionwindow)

testing1 **=** testing1.withColumn("sp500Close1",lagtest)

*//create a lag 1 time period of NASDAQ close price variable*

lagtest **=** lag("msftClose",84,0).over(partitionwindow)

testing1 **=** testing1.withColumn("msftClose1",lagtest)

lagtest **=** lag("googClose",84,0).over(partitionwindow)

testing1 **=** testing1.withColumn("googClose1",lagtest)

*//creating a 5 period of look back window*

*//create a lag 2 time period of apple close price variable*

lagtest **=** lag("close",168,0).over(partitionwindow)

**var** testing2 **=** testing1.withColumn("lag2",lagtest)

*//create a lag 3 time period of apple close price variable*

lagtest **=** lag("close",252,0).over(partitionwindow)

testing2 **=** testing2.withColumn("lag3",lagtest)

*//create a lag 4 time period of apple close price variable*

lagtest **=** lag("close",336,0).over(partitionwindow)

testing2 **=** testing2.withColumn("lag4",lagtest)

*//create a lag 5 time period of apple close price variable*

lagtest **=** lag("close",420,0).over(partitionwindow)

testing2 **=** testing2.withColumn("lag5",lagtest)

*//create features as log change between each time period's close price*

*// difference between lag 1 time period and 2 time periods*

testing2 **=** testing2.withColumn("diff12",log10($"lag1"/$"lag2"))

*// diference between lag 2 and lag 3 time periods*

testing2 **=** testing2.withColumn("diff23",log10($"lag2"/$"lag3"))

*// diference between lag 3 and lag 4 time periods*

testing2 **=** testing2.withColumn("diff34",log10($"lag3"/$"lag4"))

*// diference between lag 4 and lag 5 time periods*

testing2 **=** testing2.withColumn("diff45",log10($"lag4"/$"lag5"))

*//create 1 time period lag of market capitalization*

testing2 **=** testing2.withColumn("marketcap",($"lag1"\*$"share1"))

*//creating moving average variable as a feature. the look back period is from 11 period ago to 1 period ago*

*//using lag1 as the close price from a period ago and average up to 10 periods before*

testing2 **=** testing2.withColumn("movingAverage", avg(testing2("lag1"))

.over( **Window**.partitionBy("date").rowsBetween(-252,0)) )

testing2 **=** testing2.withColumn("label",$"close")

*//extracting the only relatve feature columns to a new training dataframe*

**var** testing3 **=** testing2.select("lag1","lag2","lag3","lag4","lag5","open1","high1","low1","adj close1","volume1","diff12","diff23","diff34","diff45","movingAverage","marketcap","sp500Close1","msftClose1","googClose1", "close","label")

*//drop any NAN or null value from the table*

testing3 **=** testing3.na.drop()

*// Building a Z Score Feature*

**val** o **=** testing3.select("label").count

**var** t4 **=** testing3.select("label").agg(sum("label"))

**var** t5 **=** t4.withColumn("mean", $"sum(label)"/o)

**val** p**=** t5.first().getDouble(1)

**var** f3**=** testing3.withColumn("Mean", lit(p))

**var** f4**=** f3.withColumn("Std", sqrt(pow($"label"-lit(p), 2)/o))

**var** testing4**=** f4.withColumn("Zscore", ($"label"-$"Mean")/$"Std")

**val** assembler **=** **new** **VectorAssembler**()

.setInputCols(**Array**("lag1","lag2","lag3","lag4","lag5","open1","high1","low1","adj close1","volume1","diff12","diff23","diff34","diff45","movingAverage","marketcap","sp500Close1","msftClose1"))

.setOutputCol("features")

*//transform the dataframe into output assembler*

**val** output1 **=** assembler.transform(training4)

**val** output2 **=** assembler.transform(testing4)

*//putting features and close price into finalized data*

**val** train\_Data **=** output1.select("features","label")

**val** test\_Data **=** output2.select("features","label")

*// Principal Component Analysis to reduce the time taken for program excecution*

**val** pca **=** **new** **PCA**()

.setInputCol("features")

.setOutputCol("pcafeatures")

.setK(12)

.fit(train\_Data)

**val** featureIndexer **=** **new** **VectorIndexer**()

.setInputCol("features")

.setOutputCol("indexedFeatures")

.setMaxCategories(4)

.fit(train\_Data)

testing3.select("label").count

**import** **org.apache.spark.ml.regression.LinearRegression**

**import** **org.apache.spark.ml.tuning.**{**ParamGridBuilder**, **TrainValidationSplit**}

**import** **org.apache.spark.mllib.evaluation.RegressionMetrics**

**import** **org.apache.spark.ml.tuning.**{**CrossValidator**, **ParamGridBuilder**}

**val** lr **=** **new** **LinearRegression**()

.setMaxIter(10)

.setLabelCol("label")

.setFeaturesCol("features")

*// Building a pipeline*

**val** pipeline **=** **new** **Pipeline**().setStages(**Array**(pca, featureIndexer, lr))

*// Building a parameter grid for hyperparameter tuning*

**val** paramGrid **=** **new** **ParamGridBuilder**()

.addGrid(lr.regParam, **Array**(0.1, 0.01))

.addGrid(lr.fitIntercept)

.addGrid(lr.elasticNetParam, **Array**(0.0, 1.0))

.build()

*// Train Validation set could be used for cross validation when k fold cross validation takes unrealistic time. Not used in this case*

**val** tvs **=** **new** **TrainValidationSplit**()

.setEstimator(pipeline) *// the estimator can also just be an individual model rather than a pipeline*

.setEvaluator(**new** **RegressionEvaluator**().setLabelCol("label"))

.setEstimatorParamMaps(paramGrid)

.setTrainRatio(0.75)

*// K fold cross validation*

**val** cv **=** **new** **CrossValidator**()

.setEstimator(pipeline)

.setEvaluator(**new** **RegressionEvaluator**().setLabelCol("label"))

.setEstimatorParamMaps(paramGrid)

.setNumFolds(5)

**val** cvmodel **=** cv.fit(train\_Data)

*// Make predictions on test set*

**var** predictions **=** cvmodel.transform(test\_Data)

predictions.show()

*// Compute RMSE*

**var** evaluator **=** **new** **RegressionEvaluator**()

.setLabelCol("label")

.setPredictionCol("prediction")

.setMetricName("rmse")

**var** rmse **=** evaluator.evaluate(predictions)

println(s"Root Mean Squared Error (RMSE) on test data = $rmse")

*// Compute Smape*

**var** number **=** predictions.select("prediction").count()

**var** abspred**=**predictions.withColumn("abspred", when(col("prediction") < 0,col("prediction")\*(-1)).

otherwise(col("prediction")))

**var** abstrue**=**abspred.withColumn("abstrue", when(col("label") < 0,col("label")\*(-1)).otherwise(col("label")))

**var** diff**=**abstrue.withColumn("diff", $"abstrue"-$"abspred")

**var** absdiff **=** diff.withColumn("absdiff", when(col("diff") < 0,col("diff")\*(-1)).otherwise(col("diff")))

**var** total**=**absdiff.withColumn("total", $"abspred"+$"abstrue")

**var** twotime**=**total.withColumn("2time", $"absdiff"\*2)

**var** summ**=**twotime.withColumn("sum", $"2time"/$"total")

**var** smape **=** summ.select("sum").agg(sum("sum"))

summ.show()

smape **=** smape.withColumn("final", $"sum(sum)"\*100/number)

smape.show()

## Decision tree + Random forest + Gradient boosting tree

*///////////////////// importing libraries including decision tree, gradient boosting trees and random forest*

**import** **org.apache.spark.ml.regression.**{**GBTRegressionModel**, **GBTRegressor**}

**import** **org.apache.spark.ml.regression.**{**RandomForestRegressionModel**, **RandomForestRegressor**}

**import** **org.apache.spark.ml.regression.**{**DecisionTreeRegressor**,**DecisionTreeRegressionModel**}

**import** **org.apache.spark.ml.Pipeline**

**import** **org.apache.spark.ml.evaluation.RegressionEvaluator**

**import** **org.apache.spark.ml.feature.VectorIndexer**

**import** **org.apache.spark.ml.feature.Interaction**

**import** **org.apache.spark.ml.feature.VectorAssembler**

**import** **org.apache.spark.ml.linalg.Vector**

**import** **org.apache.spark.sql.Row**

**import** **org.apache.spark.sql.expressions.Window**

**import** **org.apache.spark.ml.tuning.**{**CrossValidator**, **ParamGridBuilder**}

**import** **org.apache.spark.sql.functions.\_**

**import** **org.apache.spark.ml.feature.PCA**

*/////////////////////////////////////////////////////////////////////////////////////////////*

*////////////////////////Data imports//////////////////////////////////////////////////////////////*

*//set project to use default database in databricks*

sqlContext.sql("use default")

*//apple stock data*

**var** aapl **=** spark.table("aapl\_complete\_csv")

*////////////////////////////////////////////data transformation//////////////////////////////////////////////*

*//create a partition window base on date*

**val** partitionwindow **=** **Window**.orderBy("date")

*//create a lag 4 months of apple close price variable*

**var** lagtest **=** lag("close",84,0).over(partitionwindow)

*//creata a new column with lag 4 months close price*

aapl **=** aapl.withColumn("lag1",lagtest)

*//creata a new column with the difference between current and lagged price to create stationary data*

aapl **=** aapl.withColumn("diff1",($"close"-$"lag1"))

*//create a lag 4 months of apple open price column*

lagtest **=** lag("open",84,0).over(partitionwindow)

aapl**=** aapl.withColumn("open1",lagtest)

*//create a lag 4 months of apple high price column*

lagtest **=** lag("high",84,0).over(partitionwindow)

aapl **=**aapl.withColumn("high1",lagtest)

*//create a lag 4 months of apple low price column*

lagtest **=** lag("low",84,0).over(partitionwindow)

aapl **=** aapl.withColumn("low1",lagtest)

*//create a lag 4 months of apple adjusted close price column*

lagtest **=** lag("adj close",84,0).over(partitionwindow)

aapl **=** aapl.withColumn("adj close1",lagtest)

*//create a lag 4 months of apple trade volume column*

lagtest **=** lag("volume",84,0).over(partitionwindow)

aapl **=** aapl.withColumn("volume1",lagtest)

*//create a lag 4 months of SP500 close price column*

lagtest **=** lag("sp500close",84,0).over(partitionwindow)

aapl **=** aapl.withColumn("sp500close1",lagtest)

*//create a lag 4 month of NASDAQ close price column*

lagtest **=** lag("nasdaqclose",84,0).over(partitionwindow)

aapl**=** aapl.withColumn("nasdaqclose1",lagtest)

*//create difference between current and lagged value for open, high, low, adj close, volumn, sp500, and NASDAQ to create stationary data*

aapl **=** aapl.withColumn("diffopen",($"open"-$"open1"))

aapl **=** aapl.withColumn("diffhigh",($"high"-$"high1"))

aapl **=** aapl.withColumn("difflow",($"low"-$"low1"))

aapl **=** aapl.withColumn("diffadj",($"adj close"-$"adj close1"))

aapl **=** aapl.withColumn("diffvolume",($"volume"-$"volume1"))

aapl **=** aapl.withColumn("diffsp",($"sp500close"-$"sp500close1"))

aapl **=** aapl.withColumn("diffnasdaq",($"nasdaqclose"-$"nasdaqclose1"))

*//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*////////////////////////////////////////////Feature Engineering/////////////////////////////////////////////////////////////*

*//Take lag of 4 months on the close price difference variable*

lagtest **=** lag("diff1",84,0).over(partitionwindow)

aapl **=** aapl.withColumn("difflag1",lagtest)

*//Take lag of 4 months on the open price difference variable*

lagtest **=** lag("diffopen",84,0).over(partitionwindow)

aapl**=** aapl.withColumn("diffopen1",lagtest)

*//Take lag of 4 months on the high price difference variable*

lagtest **=** lag("diffhigh",84,0).over(partitionwindow)

aapl **=**aapl.withColumn("diffhigh1",lagtest)

*//Take lag of 4 months on the low price difference variable*

lagtest **=** lag("difflow",84,0).over(partitionwindow)

aapl **=** aapl.withColumn("difflow1",lagtest)

*//Take lag of 4 months on the adjusted close price difference variable*

lagtest **=** lag("diffadj",84,0).over(partitionwindow)

aapl **=** aapl.withColumn("diffadj1",lagtest)

*//Take lag of 4 months on the volumn difference variable*

lagtest **=** lag("diffvolume",84,0).over(partitionwindow)

aapl **=** aapl.withColumn("diffvolume1",lagtest)

*//Take lag of 4 months on the sp500 close price difference variable*

lagtest **=** lag("diffsp",84,0).over(partitionwindow)

aapl **=** aapl.withColumn("diffsp500close1",lagtest)

*//Take lag of 4 months on the NASDAQ close price difference variable*

lagtest **=** lag("diffnasdaq",84,0).over(partitionwindow)

aapl**=** aapl.withColumn("diffnasdaqclose1",lagtest)

*//Take lag of 8 months on the open price difference variable*

lagtest **=** lag("diffopen",168,0).over(partitionwindow)

aapl**=** aapl.withColumn("diffopen2",lagtest)

*//Take lag of 8 months on the high price difference variable*

lagtest **=** lag("diffhigh",168,0).over(partitionwindow)

aapl **=**aapl.withColumn("diffhigh2",lagtest)

*//Take lag of 8 months on the low price difference variable*

lagtest **=** lag("difflow",168,0).over(partitionwindow)

aapl **=** aapl.withColumn("difflow2",lagtest)

*//Take lag of 8 months on the adjusted close price difference variable*

lagtest **=** lag("diffadj",168,0).over(partitionwindow)

aapl **=** aapl.withColumn("diffadj2",lagtest)

*//Take lag of 8 months on the volumn difference variable*

lagtest **=** lag("diffvolume",168,0).over(partitionwindow)

aapl **=** aapl.withColumn("diffvolume2",lagtest)

*//Take lag of 8 months on the sp500 close price difference variable*

lagtest **=** lag("diffsp",168,0).over(partitionwindow)

aapl **=** aapl.withColumn("diffsp500close2",lagtest)

*//Take lag of 8 months on the NASDAQ close price difference variable*

lagtest **=** lag("diffnasdaq",168,0).over(partitionwindow)

aapl**=** aapl.withColumn("diffnasdaqclose2",lagtest)

*//Take second order difference between lag 4 month difference value and lag 8 month difference value for open, high, low, adj close, volume, sp500 close, and NASDAQ close*

*// only lag value can be use to create features to avoid data leakage from the future.*

aapl **=** aapl.withColumn("diffopenlag",$"diffopen1"-$"diffopen2")

aapl **=** aapl.withColumn("diffhighlag",$"diffhigh1"-$"diffhigh2")

aapl **=** aapl.withColumn("difflowlag",$"difflow1"-$"difflow2")

aapl **=** aapl.withColumn("diffadjlag",$"diffadj1"-$"diffadj2")

aapl **=** aapl.withColumn("diffvolumelag",$"diffvolume1"-$"diffvolume2")

aapl **=** aapl.withColumn("diffsplag",$"diffsp500close1"-$"diffsp500close2")

aapl **=** aapl.withColumn("diffnasdaqlag",$"diffnasdaqclose1"-$"diffnasdaqclose2")

*//Take lag of 8 months on the close price difference variable*

lagtest **=** lag("diff1",168,0).over(partitionwindow)

**var** aapl2 **=**aapl.withColumn("difflag2",lagtest)

*//Take second order difference between lag 4 month difference value and lag 8 month difference value for close price*

aapl2 **=** aapl2 .withColumn("diff12",($"difflag1"-$"difflag2"))

*//creating moving average variable as a feature. the look back period for 1 month for short term trend*

aapl2 **=** aapl2 .withColumn("movingAverage", avg(aapl2("difflag1"))

.over( **Window**.partitionBy("date").rowsBetween(-21,0)) )

*//creating moving average variable as a feature. the look back period for 1 year for long term trend*

aapl2 **=** aapl2 .withColumn("phase", avg(aapl2("difflag1"))

.over( **Window**.partitionBy("date").rowsBetween(-252,0)) )

*//calculate moving z score as feature*

*// create a short term window for 1 month*

**var** w **=** **Window**.orderBy("date").rowsBetween(-21, 0)

*// get moving 1 month standard deviation*

aapl2 **=** aapl2 .withColumn("std", stddev("difflag1").over(w))

*//get z score by taking current value- moving average and divided by standard deviation*

aapl2 **=** aapl2.withColumn("zscore",($"difflag1"-$"movingAverage")/$"std")

*// create a long term window for 1 year*

w **=** **Window**.orderBy("date").rowsBetween(-252, 0)

*// get standard deviation for 1 year period*

aapl2 **=** aapl2 .withColumn("std1", stddev("difflag1").over(w))

*//get z score by taking current value- moving average and divided by standard deviation*

aapl2 **=** aapl2.withColumn("zscore1",($"difflag1"-$"phase")/$"std1")

*//copy the changes in close price into label column as target variable*

aapl2 **=** aapl2 .withColumn("label",$"diff1")

*//split the training and testing data base on date*

**var** training **=** aapl2.select("\*").where($"date"<"2017-11-01" && $"date">"2004-01-01")

**var** testing **=** aapl2.select("\*").where($"date">"2017-11-01")

*//filter out only the required column into final dataframe*

**var** training3 **=** training.select("difflag1","difflag2","diff12","diffopenlag","diffhighlag","difflowlag","diffadjlag","diffvolumelag","movingAverage","diffsplag","diffnasdaqlag","lag1","zscore","zscore1","label","phase")

**var** testing3 **=** testing.select("difflag1","difflag2","diff12","diffopenlag","diffhighlag","difflowlag","diffadjlag","diffvolumelag","movingAverage","diffsplag","diffnasdaqlag","lag1","zscore1","zscore","label","phase")

*//drop any NAN or null value from the table*

training3 **=** training3.na.drop()

testing3 **=** testing3.na.drop()

*///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*///////////////////Model Building/////////////////////////////////////////////////////////////////////////////////////////////////////*

*// create feature vector*

**var** assembler **=** **new** **VectorAssembler**()

.setInputCols(**Array**("diffopenlag","diffhighlag","difflowlag","difflag1","difflag2","diff12","diffadjlag","diffvolumelag","movingAverage","diffsplag","diffnasdaqlag","phase","zscore","zscore1"))

.setOutputCol("features")

*//transform the databframe into output assembler for both training and testing data*

**var** output1 **=** assembler.transform(training3)

**var** output2 **=** assembler.transform(testing3)

*//putting features and lable into finalized data*

**var** train\_Data **=** output1.select("features","label")

**var** test\_Data **=** output2.select("features","label")

*//create feature indexer*

**var** featureIndexer **=** **new** **VectorIndexer**()

.setInputCol("features")

.setOutputCol("indexedFeatures")

.setMaxCategories(4)

.fit(train\_Data)

*//create decision tree model and input feature vector and label column*

**val** dt **=** **new** **DecisionTreeRegressor**()

.setLabelCol("label")

.setFeaturesCol("indexedFeatures")

*//crete a pipeline*

**var** pipeline **=** **new** **Pipeline**()

.setStages(**Array**(featureIndexer, dt))

*//create a parameter search grid*

**var** paramGrid **=** **new** **ParamGridBuilder**()

.addGrid(dt.maxDepth,**Array**(5,10,20))

.addGrid(dt.maxBins, **Array**(20, 100, 200))

.build()

*// creates cross validation model to search for best parameter combination*

**var** cv **=** **new** **CrossValidator**()

.setEstimator(pipeline)

.setEvaluator(**new** **RegressionEvaluator**)

.setEstimatorParamMaps(paramGrid)

.setNumFolds(5)

.setParallelism(3)

*// Run cross-validation, and choose the best set of parameters.*

**var** cvModel **=** cv.fit(train\_Data)

*//////////////////////////////////Performance Evaluation/////////////////////////////////*

*// get decision tree prediction*

**var** predictions **=** cvModel.transform(test\_Data)

*// get RMSE from evaluator*

**var** evaluator **=** **new** **RegressionEvaluator**()

.setLabelCol("label")

.setPredictionCol("prediction")

.setMetricName("rmse")

**var** rmse **=** evaluator.evaluate(predictions)

*//get the feature importance*

**var** model **=** pipeline.fit(train\_Data)

**var** importance **=** model.stages(1).asInstanceOf[DecisionTreeRegressionModel].featureImportances

*//////////////////calculate SMAPE/////////////////////////////////////////////////////////////*

*//Get prediction results and target variable into a new dataframe*

**var** newpred **=** predictions.select("prediction","label")

*//Add row number as a new column*

newpred **=** newpred.withColumn("rows",monotonically\_increasing\_id())

*// get lagged value from original data*

**var** newtest **=** testing3.select("lag1")

*// add row number as a new column*

newtest **=** newtest.withColumn("rows1",monotonically\_increasing\_id())

*// join prediction, target variable and 4 months lagged close price*

**var** pred **=** newpred.join(newtest, newpred.col("rows") === newtest.col("rows1"))

*//drop NA if any*

pred **=** pred.na.drop()

*// the model predicts the changes in close price, need to add back to the previous period close price to get the final predicted close price.*

pred **=** pred.withColumn("predictionback",($"prediction")+ $"lag1")

*//add back to the previous period close price to get the final actual close price.*

pred **=** pred.withColumn("label1",($"label")+ $"lag1")

*// count the number of test data*

**var** number **=** predictions.select("prediction").count()

*// get absolute predcition results*

**var** abspred**=**pred.withColumn("abspred", when(col("predictionback") < 0,col("predictionback")\*(-1)).

otherwise(col("predictionback")))

*//get absolute actual results*

**var** abstrue**=**abspred.withColumn("abstrue", when(col("label1") < 0,col("label1")\*(-1)).otherwise(col("label1")))

*// calculate the difference between prediction and actual*

**var** diff**=**abstrue.withColumn("diff", $"label1"-$"predictionback")

*// get absolute on the difference between prediction and actual*

**var** absdiff **=** diff.withColumn("absdiff", when(col("diff") < 0,col("diff")\*(-1)).otherwise(col("diff")))

*//get total value by adding prediction and actual results*

**var** total**=**absdiff.withColumn("total", $"abspred"+$"abstrue")

*// times the absolute difference by 2*

**var** twotime**=**total.withColumn("2time", $"absdiff"\*2)

*//get the total error by taking 2 times absolute difference divided by sum of prediction and atual value*

**var** summ**=**twotime.withColumn("sum", $"2time"/$"total")

**var** smape **=** summ.select("sum").agg(sum("sum"))

*// use the sum of error \*100 and divid by the total number of test data to get SMAPE as percentage*

smape **=** smape.withColumn("final", $"sum(sum)"\*100/number)

*//display SMAPE value*

smape.show()

*/////////////////////////////change to ensemble tree models///////////////////////////////////////////////////////////////////////////////*

*///////Reduce in features compare to decision tree base on feature importance and prevent overfitting*

training3 **=** training.select("difflag1","difflag2","diff12","diffadjlag","diffvolumelag","movingAverage","diffsplag","diffnasdaqlag","lag1","label","phase")

testing3 **=** testing.select("difflag1","difflag2","diff12","diffadjlag","diffvolumelag","movingAverage","diffsplag","diffnasdaqlag","lag1","label","phase")

*//drop any NAN or null value from the table*

training3 **=** training3.na.drop()

testing3 **=** testing3.na.drop()

*///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*

*///////////////////Model Building/////////////////////////////////////////////////////////////////////////////////////////////////////*

*// create feature vector*

assembler **=** **new** **VectorAssembler**()

.setInputCols(**Array**("difflag1","difflag2","diff12","diffadjlag","diffvolumelag","movingAverage","diffsplag","diffnasdaqlag","phase"))

.setOutputCol("features")

*//transform the databframe into output assembler for both training and testing data*

output1 **=** assembler.transform(training3)

output2 **=** assembler.transform(testing3)

*//putting features and lable into finalized data*

train\_Data **=** output1.select("features","label")

test\_Data **=** output2.select("features","label")

*//create feature indexer*

featureIndexer **=** **new** **VectorIndexer**()

.setInputCol("features")

.setOutputCol("indexedFeatures")

.setMaxCategories(4)

.fit(train\_Data)

*//create graident boosting tree model*

**val** gbt **=** **new** **GBTRegressor**()

.setLabelCol("label")

.setFeaturesCol("indexedFeatures")

*//create pipeline*

pipeline **=** **new** **Pipeline**()

.setStages(**Array**(featureIndexer, gbt))

*//create parameter search grid*

paramGrid **=** **new** **ParamGridBuilder**()

.addGrid(gbt.maxDepth,**Array**(2,5,10))

.addGrid(gbt.maxBins, **Array**(20, 100, 200))

.addGrid(gbt.maxIter, **Array**(5, 10, 20))

.build()

*//create cross valiation model*

cv **=** **new** **CrossValidator**()

.setEstimator(pipeline)

.setEvaluator(**new** **RegressionEvaluator**)

.setEstimatorParamMaps(paramGrid)

.setNumFolds(5)

.setParallelism(3)

*// Run cross-validation, and choose the best set of parameters.*

cvModel **=** cv.fit(train\_Data)

*/////////////////////////////////////////////////////////////////////////////////Model Evaluation/////////////////////////////////////////////*

*//get predictions*

predictions **=** cvModel.transform(test\_Data)

*// get feature importance*

**var** model **=** pipeline.fit(train\_Data)

importance **=** model.stages(1).asInstanceOf[GBTRegressionModel].featureImportances

*//calcuate RMSE*

evaluator **=** **new** **RegressionEvaluator**()

.setLabelCol("label")

.setPredictionCol("prediction")

.setMetricName("rmse")

rmse **=** evaluator.evaluate(predictions)

*//////////////////calculate SMAPE/////////////////////////////////////////////////////////////*

*//Get prediction results and target variable into a new dataframe*

newpred **=** predictions.select("prediction","label")

*//Add row number as a new column*

newpred **=** newpred.withColumn("rows",monotonically\_increasing\_id())

*// get lagged value from original data*

newtest **=** testing3.select("lag1")

*// add row number as a new column*

newtest **=** newtest.withColumn("rows1",monotonically\_increasing\_id())

*// join prediction, target variable and 4 months lagged close price*

pred **=** newpred.join(newtest, newpred.col("rows") === newtest.col("rows1"))

*//drop NA if any*

pred **=** pred.na.drop()

*// the model predicts the changes in close price, need to add back to the previous period close price to get the final predicted close price.*

pred **=** pred.withColumn("predictionback",($"prediction")+ $"lag1")

*//smoothing technique to smooth out the prediction results base on previous and post prediction results*

pred **=** pred .withColumn("prediction1", avg(pred("predictionback"))

.over( **Window**.partitionBy().rowsBetween(-10,10) ))

*//add back to the previous period close price to get the final actual close price.*

pred **=** pred.withColumn("label1",($"label")+ $"lag1")

*// count the number of test data*

number **=** predictions.select("prediction").count()

*// get absolute predcition results*

abspred**=**pred.withColumn("abspred", when(col("prediction1") < 0,col("prediction1")\*(-1)).

otherwise(col("prediction1")))

*//get absolute actual results*

abstrue**=**abspred.withColumn("abstrue", when(col("label1") < 0,col("label1")\*(-1)).otherwise(col("label1")))

*// calculate the difference between prediction and actual*

diff**=**abstrue.withColumn("diff", $"label1"-$"prediction1")

*// get absolute on the difference between prediction and actual*

absdiff **=** diff.withColumn("absdiff", when(col("diff") < 0,col("diff")\*(-1)).otherwise(col("diff")))

*//get total value by adding prediction and actual results*

total**=**absdiff.withColumn("total", $"abspred"+$"abstrue")

*// times the absolute difference by 2*

twotime**=**total.withColumn("2time", $"absdiff"\*2)

*//get the total error by taking 2 times absolute difference divided by sum of prediction and atual value*

summ**=**twotime.withColumn("sum", $"2time"/$"total")

smape **=** summ.select("sum").agg(sum("sum"))

*// use the sum of error \*100 and divid by the total number of test data to get SMAPE as percentage*

smape **=** smape.withColumn("final", $"sum(sum)"\*100/number)

*//display SMAPE value*

smape.show()

*////create random forest model/////*

**val** rf **=** **new** **RandomForestRegressor**()

.setLabelCol("label")

.setFeaturesCol("indexedFeatures")

*// create Pipeline.*

**val** pipeline1 **=** **new** **Pipeline**()

.setStages(**Array**(featureIndexer, rf))

*// create parameter search grid*

**val** paramGrid1 **=** **new** **ParamGridBuilder**()

.addGrid(rf.maxDepth,**Array**(5,15,30))

.addGrid(rf.maxBins, **Array**(10, 100, 200))

.addGrid(rf.numTrees, **Array**(5, 10, 20))

.build()

**val** cv1 **=** **new** **CrossValidator**()

.setEstimator(pipeline1)

.setEvaluator(**new** **RegressionEvaluator**)

.setEstimatorParamMaps(paramGrid1)

.setNumFolds(5)

.setParallelism(5)

**val** cvModel1 **=** cv1.fit(train\_Data)

*/////////////////////Performance Evaluation///////////////////////////////////////////////////////////*

*//prediction*

predictions **=** cvModel1.transform(test\_Data)

*// Print the coefficients and intercept for linear regression*

*//calculate RMSE*

evaluator **=** **new** **RegressionEvaluator**()

.setLabelCol("label")

.setPredictionCol("prediction")

.setMetricName("rmse")

rmse **=** evaluator.evaluate(predictions)

*//get feature importance*

**var** model1 **=** pipeline1.fit(train\_Data)

mportance **=** model1.stages(1).asInstanceOf[RandomForestRegressionModel].featureImportances

*//////////////////calculate SMAPE/////////////////////////////////////////////////////////////*

*//Get prediction results and target variable into a new dataframe*

newpred **=** predictions.select("prediction","label")

*//Add row number as a new column*

newpred **=** newpred.withColumn("rows",monotonically\_increasing\_id())

*// get lagged value from original data*

newtest **=** testing3.select("lag1")

*// add row number as a new column*

newtest **=** newtest.withColumn("rows1",monotonically\_increasing\_id())

*// join prediction, target variable and 4 months lagged close price*

pred **=** newpred.join(newtest, newpred.col("rows") === newtest.col("rows1"))

*//drop NA if any*

pred **=** pred.na.drop()

*// the model predicts the changes in close price, need to add back to the previous period close price to get the final predicted close price.*

pred **=** pred.withColumn("predictionback",($"prediction")+ $"lag1")

*//smoothing technique to smooth out the prediction results base on previous and post prediction results*

pred **=** pred .withColumn("prediction1", avg(pred("predictionback"))

.over( **Window**.partitionBy().rowsBetween(-10,10) ))

*//add back to the previous period close price to get the final actual close price.*

pred **=** pred.withColumn("label1",($"label")+ $"lag1")

*// count the number of test data*

number **=** predictions.select("prediction").count()

*// get absolute predcition results*

abspred**=**pred.withColumn("abspred", when(col("prediction1") < 0,col("prediction1")\*(-1)).

otherwise(col("prediction1")))

*//get absolute actual results*

abstrue**=**abspred.withColumn("abstrue", when(col("label1") < 0,col("label1")\*(-1)).otherwise(col("label1")))

*// calculate the difference between prediction and actual*

diff**=**abstrue.withColumn("diff", $"label1"-$"prediction1")

*// get absolute on the difference between prediction and actual*

absdiff **=** diff.withColumn("absdiff", when(col("diff") < 0,col("diff")\*(-1)).otherwise(col("diff")))

*//get total value by adding prediction and actual results*

total**=**absdiff.withColumn("total", $"abspred"+$"abstrue")

*// times the absolute difference by 2*

twotime**=**total.withColumn("2time", $"absdiff"\*2)

*//get the total error by taking 2 times absolute difference divided by sum of prediction and atual value*

summ**=**twotime.withColumn("sum", $"2time"/$"total")

smape **=** summ.select("sum").agg(sum("sum"))

*// use the sum of error \*100 and divid by the total number of test data to get SMAPE as percentage*

smape **=** smape.withColumn("final", $"sum(sum)"\*100/number)

*//display SMAPE value*

smape.show()