MIE1628 Project

Time series analysis for Apple stock price prediction

Hyun Jun Shim
1004691942
Group 12

1. Introduction

The objective of the report is to explore time series model through predicting the value of the apple stock price from 1980 to 2018. The model is built on predicting close price. The apple stock price from 1980 to 2018, which consists of 9478 records, are analyzed in this report. To predict the values accurately, three machine learning algorithm are used: Linear Regression, Moving Average, and Decision Tree Regressor. Multiple features are selected for the algorithm. After then, the dataset are split into training and testing data and the models will be trained. Through comparing the result between the algorithm, the best algorithm is chosen for the apple stock prediction model. The models are built on PySpark.

2. Dataset

The apple dataset consists opening, high, low, closing, adjusted closing price and volume of the data for each date. Table 1 shows the example of the apple stock data set.

Table 1. Apple Stock Dataset

Date	Open	High	Low	Close	Adj Close	Volume
1980-12-08	0.513393	0.515625	0.513393	0.513393	0.022919	117258400
1980-12-15	0.488839	0.506696	0.450893	0.504464	0.02252	122533600
1980-12-22	0.529018	0.636161	0.529018	0.633929	0.0283	46972800
1980-12-29	0.642857	0.645089	0.609375	0.616071	0.027503	54863200
1981-01-05	0.604911	0.604911	0.540179	0.569196	0.02541	49476000
1981-01-12	0.569196	0.569196	0.544643	0.553571	0.024712	22125600
1981-01-19	0.587054	0.591518	0.569196	0.584821	0.026108	33583200
1981-01-26	0.578125	0.578125	0.504464	0.504464	0.02252	41647200
1981-02-02	0.477679	0.515625	0.475446	0.513393	0.022919	23144800

3. Target Variable

The target variable is the predicted closing price and they are produced in a selected horizon. For example, if the selected horizon is weekly horizon, the prediction model is built by the weekly closing price data. The horizons in the report are chosen as one day, one week, two weeks, one month and three months.

4. Feature Selection

In order to increase the accuracy of the model, some features are selected and transformed for each model. For Linear Regression, high, low, open, close and volume are added as the features. As the Linear Regression is the one of the simplest model, basic features from the apple stock price data are selected. Since Moving Average model is only based on mean values, the mean of the closing prices is selected. For the Decision Tree model, close, high, low, minimum, and maximum price from apple data are selected as features. Since the Tree model is very complex compared to other two, there are some additions to these basic features, which are lag and mean. Like Moving Average, mean value of the closing price over particular horizon are added. Also there are lag features called lag 1 and 2, which represents the values from previous horizons. For example, lag 1 from daily horizon shows the value from yesterday and lag 2 from daily as the value from 2 days ago. Also, lag 2 from weekly horizon means the value from 2 weeks ago. Table 2 shows the selected features for each model.

Table 2. Feature Selection for Three Models

	Features
Linear Regression	Close, High, Low, Open, Volume
Moving Average	Close, Mean
Decision Tree	Close, High, Low, Minimum, Maximum, Mean, Lag1, Lag2

5. Model

5.1 Linear Regression

Linear regression is one of the models that are widely used throughout finance. Linear regression is a method used to model a relationship between a dependent variable (y), and an independent variable (x). With simple linear regression, there will only be one independent variable x. There can be many independent variables which would fall under the category of multiple linear regression. One of example independent variable is the date. The date will be represented by an integer starting at 1 for the first date going up to the length of the vector of dates which can vary depending on the time series data. In this project, the dependent variable is stock price. As the linear regression algorithm finds the linear relationship between dependent and independent variable, it will predict the future output based on the relationship, which in this case, y = ax+b.

5.2 Moving Average

A moving average is an algorithm to get an overall idea of the trends in a data set. It is an average of any subset of numbers. The moving average is extremely useful for forecasting long-term trends. It can be calculated for any period of time. For example, if there is sales data for a twenty-year period, the moving average algorithm can calculate a five-year moving average, a four-year moving average, a three-year moving average and so on. Stock market analysts often use a 50 or 200 day moving average to help them see trends in the stock market and (hopefully) forecast where the stocks are headed.(1)
Overall, moving calculates the unweighted mean of the last n samples. The parameter n is often called the window size, because the algorithm can be thought of as a window that slides over the data points.

5.3 Decision Tree Regressor

Decision tree builds regression models in the form of a tree structure. It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed. The final result is a tree with decision nodes and leaf nodes. A decision node has two or more branches. Leaf node represents a classification or decision. The topmost decision node in a tree which corresponds to the best predictor is called root node. To predict the outcome in each leaf node, the average outcome of the training data in this node is used. Decision trees can handle both categorical and numerical data. (3)

6. Model Analysis

To analyze which model is best to predict apple stock price, Symmetric mean absolute percentage error(SMAPE) and Root Mean Square Error (RMSE) are used to compare the accuracy between the models.

$$\text{SMAPE} = \frac{100\%}{n} \sum_{t-1}^{n} \frac{|F_t - A_t|}{(|A_t| + |F_t|)/2}$$

Figure 1. SMAPE formula

$$RMSE = \sqrt{\overline{(f - o)^2}}$$

Figure 2. RMSE Formula

The prediction is also graphed along with closing price to confirm whether there was underfitting or overfitting issue.

6.1 Linear Regression

The linear regression model was developed based on the time line from 1980 to 2018. Firstly, adjacent close, high, low, open, close and volume are imported in the model. They are then normalized with a log function, which will reduce some noise when the features are fit into the model. The simple return values are added to the columns using the formal in Figure 3.

$$R(t) = \frac{S(t) - S(0)}{S(0)} = \frac{S(t)}{S(0)} - 1$$

Figure 3. Simple Return Formula

There are volume difference columns, subtracting today's volume from the price of horizon days ago. Finally, after some lagging transformations, there are three types of features, which are log_type, return_type and volume_diff_type. The dataset is split into 80% training data and 20% validation data. Then, the linear regression model is trained with the training dataset. There is also a param grid builder to find the best hyper-parameters.

Table 3 shows the comparison of SMAPE and RMSE over daily, weekly, biweekly, monthly, and quarterly horizon over 1980 to 2018 timeline. SMAPE for decision tree ranges from 0.627 to 5.16 and RMSE ranges calculated as 0.027 to 0.216.

Table 3: The SMAPE and RMSE for all horizons (Linear Regression Model)

horizon	One day	One week	Bi-week	One month	Three months
SMAPE	0.627	1.076	1.653	2.19	5.16
RMSE	0.027	0.0476	0.0719	0.099	0.216

Figure 4 and 5 shows the graph of the close price against the predicted values based on 1985 to 2018 timeline. The daily, weekly, bi weekly, monthly, and quarterly graph shows the prediction to be good fit with the closing price curve. Therefore, there is no problem of overfitting in this model.

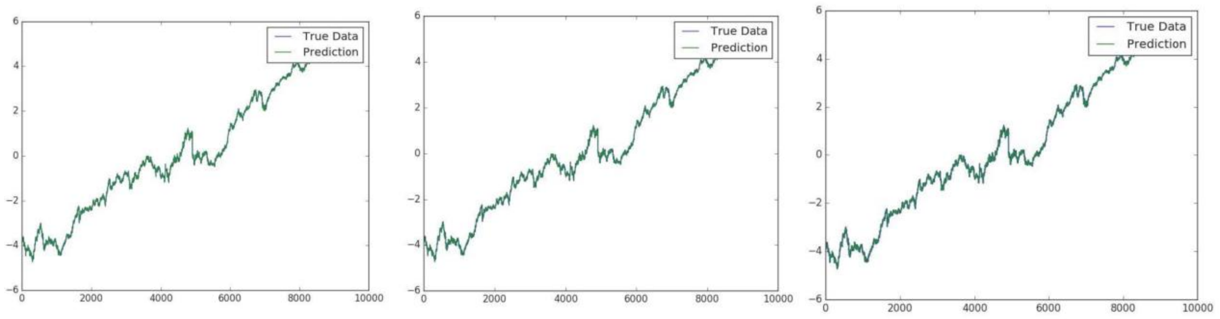

Figure 4: Close prediction vs Prediction in daily, weekly and bi weekly order in Linear Regression Model

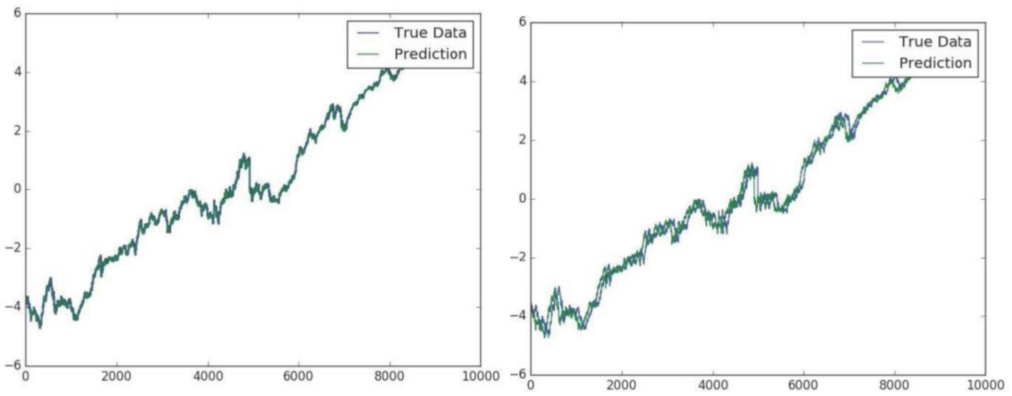

Figure 5: Close prediction vs Prediction in monthly and quarterly order in Linear Regression Model

6.2 Moving Average

The moving average algorithm works through moving a window through the dataset and calculating the mean of the data in the window function for every step. After getting the series of average data, these data are used to predict the closing price. Based on this idea, the window length has been set from 1 day to a relatively large number, and the length were compared to observe which one makes the least metrics. After then, the moving average will be built on this length to predict the real future stock price.
In the code, window length has been set from 1 day to 80 days for every horizon. Finally, the length that produces the least SMAPE has been found and the graph of prediction cure with that length has been plotted. Figure 6 and 7 shows the window length against the SMAPE for daily, weekly, biweekly, monthly, and quarterly horizon. The figures show that the SMAPE rises as the length increases. In other words, 1 day length is shown to be the best choice for the model.

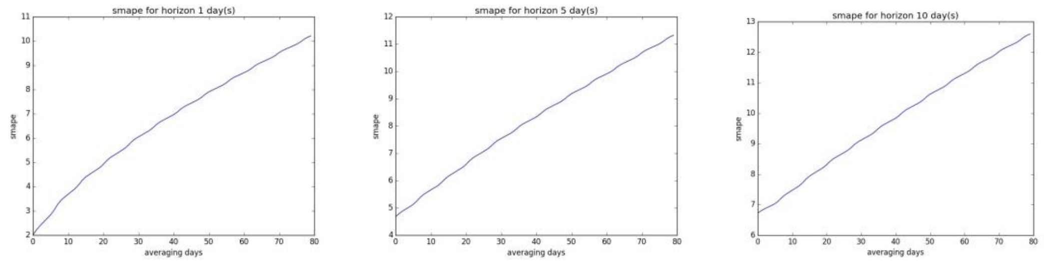

Figure 6. Moving average days vs SMAPE for daily, weekly, and bi weekly horizon

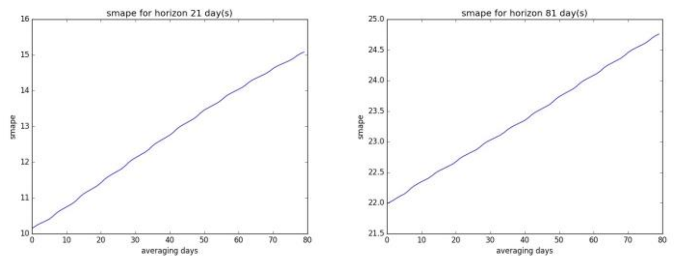

Figure 7. Moving average days vs SMAPE for monthly and quarterly horizon

Table 4 shows the comparison between SMAPE and RMSE over daily, weekly, biweekly, monthly, and quarterly horizon over 1985 to 2018 timeline. SMAPE for moving average ranges from 2.01 to 22.44 and RMSE ranges calculated as 0.67 to 6.67.

Table 4: The SMAPE and RMSE for all horizons (Moving Average Model Model)

horizon	One day	One week	biweek	One month	Three months
smape	2.007	4.676	6.731	10.147	22.44
rmse	0.666	1.489	2.107	3.155	6.667
Window length	1	1	1	1	1

Figure 8 and 9 shows the graph of the close price against the predicted values calculated by moving average model. All figures show that models produces good fit graph which means there is no issue of overfitting in the model.

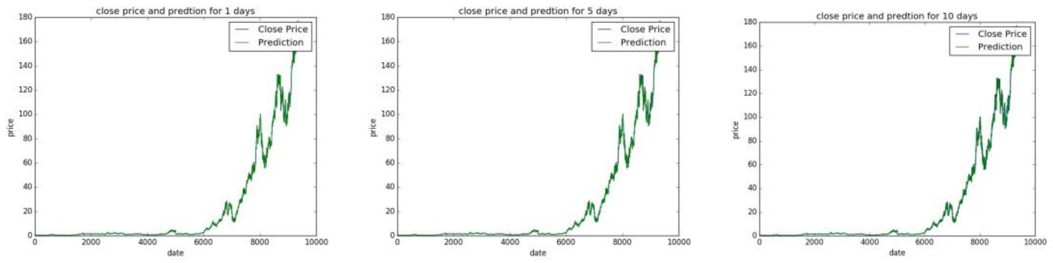

Figure 8. Close prediction vs Prediction for daily, weekly and bi weekly order in Moving Average Model

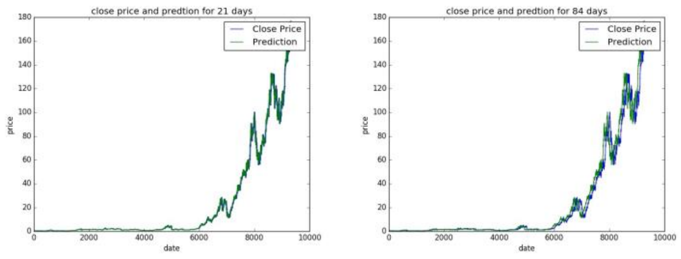

Figure 9. Close prediction vs Prediction monthly and quarterly order in Moving Average Model

6.3 Decision Tree

The decision tree model is developed based on the time line from 2000 to 2010. In order to reduce variance between data, this timeline is chosen rather than 1980 to 2018. To reduce the noise of the data, the original values of the features, such as close, high, low, mean, max, and minimum price, are calculated into logarithm values. For example, if the close price is $1.2, the price has been changed into log (1.2), which is 0.08. The figure 10 shows the comparison between the close price and the logarithm value of the price. The left one which is the original values shows the variance in the data while the right one, where the values are transformed into logarithm values, shows smooth curve which makes easier for the model to predict the values.

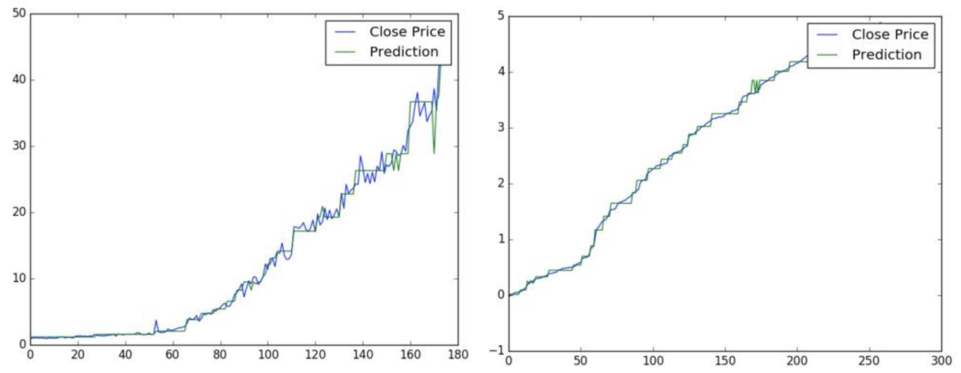

Figure 10. Original Daily Close Price vs Logarithm Value of the Daily Close Price

Table 5 shows the comparison of SMAPE and RMSE over daily, weekly, biweekly, monthly, and quarterly horizon over 2000 to 2010 timeline. SMAPE for decision tree ranges from 8.07 to 10.68 and RMSE ranges calculated as 0.05 to 0.052.

Table 5: The SMAPE and RMSE for all horizons (Decision Tree)

horizon	One day	One week	Bi-week	One month	Three months
SMAPE	8.07	6.959	9.132	10.517	10.678
RMSE	0.050	0.0578	0.0495	0.0592	0.0521

Figure 11 and 12 shows the graph of the close price against the predicted values based on 2000 to 2010 timeline. While the daily, weekly and bi weekly graph shows the prediction as good fit with the close price, the monthly and quarterly graph from figure 3 shows there are some variance in prediction curve. Nevertheless, it seems to be minor; which is why, there is no problem of overfitting in the model.

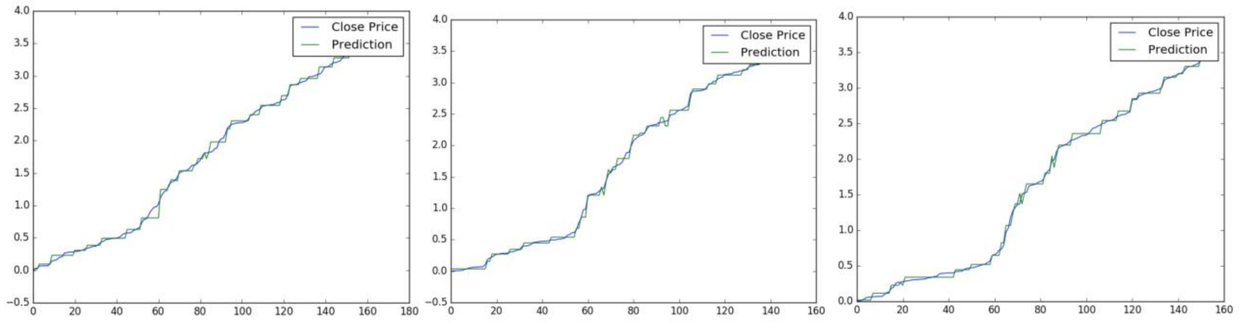

Figure 11. Close prediction vs Prediction for daily, weekly and bi weekly order in Decision Tree Model

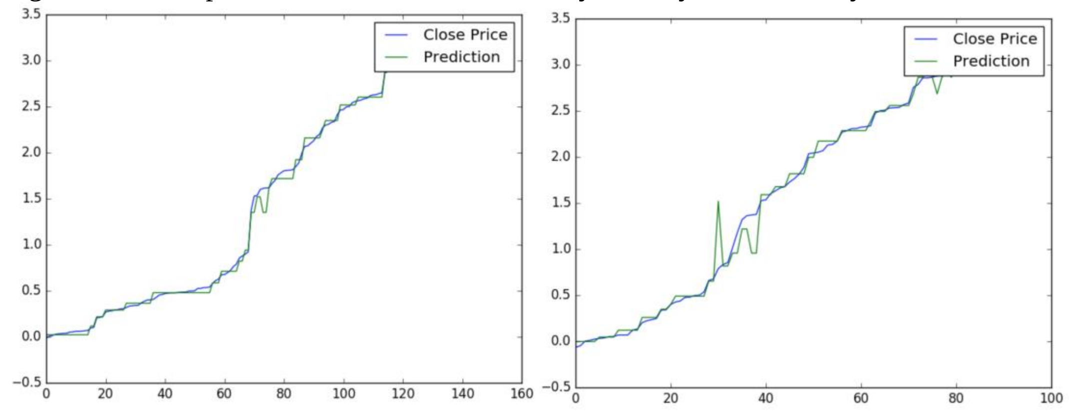

Fig 12. Close prediction vs SMAPE, RMSE for monthly and quarterly order Decision Tree Model

As the decision model carries multiple features, such as lag, low, high, minimum, maximum, mean, and close price, feature importance function is used to analyze which feature has the biggest impact on the prediction model. Table 6 shows the ranking of the feature importance on each five model. It shows that most of the model is effected by lag price. Therefore, the decision tree regressor model predicts the closing price through analyzing majorly lag value.

Table 6. Feature Importance

	Rank 1	Rank 2	Rank 3
Daily Model	Low price	Lag 1 price(1 day ago)	Lag 2 price(2 day ago)
Weekly Model	Lag 1 price(1 week ago)	Lag 2 price(2 weeks ago)	
Biweekly Model	Lag 1 price(2 weeks ago)	Lag 2 price(1 month ago)	Low price
Monthly Model	Lag 1 price(1 month ago)	Lag 2 price(2 months ago)	
Quarterly Model	Maximum price	Lag 2 price(6 months ago)	minimum price

7. Conclusion

Table 7 shows the comparison of the result between all three models.

Model	Used Data	Feature Selection	Train/Test Split	SMAPE (Daily)	SMAPE (Quarterly)	Overfitting?
Linear Regression	Apple Stock Price	Close Return Volume	80/20	0.627	5.16	NO
Moving Average	Apple Stock Price	Close Mean	-	2.007	22.44	NO
Decision Tree	Apple Stock Price	Close Lag 1,2 High Low Minimum Maximum Mean	70/30	8.07	10.678	NO

With no overfitting issues in these models, Linear Regression models shows the least SMAPE out of all three models. Decision tree model includes 8 features, which is the highest feature collection out of all models, but shows higher SMAPE on daily horizon. The range between daily and quarterly SMAPE for the tree model shows smaller compared to previous two models which shows unstable. While Moving Average model shows decent number of SMAPE in daily horizon, its SMAPE in quarterly horizon shows relatively high. On the other hand, Linear Regression shows the least SMAPE value both in daily and quarterly horizon. With no overfitting issues in these models, Linear Regression models is proven to be the best model out of all three.

Reference

(1) Stephanie (2013). *Statistics How to*. Retrieved from https://www.statisticshowto.datasciencecentral.com/moving-average/

(2) Sayad, S. *Decision Tree - Classification* Retrieved from https://www.saedsayad.com/decision_tree.htm

Appendix

Python Code
Linear Regression

```
from pyspark.sql import SQLContext,Window
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
import pyspark.sql.functions as sf
import pyspark.sql.functions as Column

import pandas as pd
import datetime
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import *
import pyspark.sql.functions as sf
from pyspark.sql.window import Window
import numpy as np

from math import log
from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler

from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml.regression import GBTRegressor
import matplotlib.pyplot as plt
from pyspark.ml.regression import DecisionTreeRegressor
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.sql.functions import abs, sqrt
from pyspark.sql.types import FloatType
from pyspark.sql.types import DoubleType

sqlContext = SQLContext(sc)
df = sqlContext.sql("SELECT * FROM aapl_csv WHERE YEAR(aapl_csv.date) BETWEEN 1980 AND 
2018")
df.show(10)
```

```
+-------------------+--------+--------+--------+--------+---------+---------+ | Date| Open| High| Low| Close|Adj Close| 
Volume| +-------------------+--------+--------+--------+--------+---------+---------+ |1980-12-08
```

```
00:00:00|0.513393|0.515625|0.513393|0.513393| 0.022919|117258400| |1980-12-15
00:00:00|0.488839|0.506696|0.450893|0.504464| 0.02252|122533600| |1980-12-22
00:00:00|0.529018|0.636161|0.529018|0.633929| 0.0283| 46972800| |1980-12-29
00:00:00|0.642857|0.645089|0.609375|0.616071| 0.027503| 54863200| |1981-01-05
00:00:00|0.604911|0.604911|0.540179|0.569196| 0.02541| 49476000| |1981-01-12
00:00:00|0.569196|0.569196|0.544643|0.553571| 0.024712| 22125600| |1981-01-19
00:00:00|0.587054|0.591518|0.569196|0.584821| 0.026108| 33583200| |1981-01-26
00:00:00|0.578125|0.578125|0.504464|0.504464| 0.02252| 41647200| |1981-02-02
00:00:00|0.477679|0.515625|0.475446|0.513393| 0.022919| 23144800| |1981-02-09
00:00:00|0.491071|0.491071|0.455357|0.455357| 0.020328| 18664800| +-------------------+--------+--------+--
------+--------+---------+---------+ only showing top 10 rows
```

```
df = df.withColumn("ticker", sf.lit("AAPL"))
df = df.withColumn("Log_Adj_Close", sf.log("Adj close"))
df = df.withColumn("log_open", sf.log("Open"))
df = df.withColumn("log_high", sf.log("High"))
df = df.withColumn("log_low", sf.log("Low"))
df = df.withColumn("log_Volume", sf.log("Volume"))
df.show(10)
```

```
+-------------------+--------+--------+--------+--------+---------+---------+------+-------------------+-------------------+--------
------------+-------------------+------------------+ | Date| Open| High| Low| Close|Adj Close| Volume|ticker|
Log_Adj_Close| log_open| log_high| log_low| log_Volume| +-------------------+--------+--------+--------+--------
+---------+---------+------+-------------------+-------------------+-------------------+-------------------+------------------+
|1980-12-08 00:00:00|0.513393|0.515625|0.513393|0.513393| 0.022919|117258400| AAPL| -
3.775789018104867| -0.666713645230959| -0.6623755218931916| -0.666713645230959|
18.57989060451556| |1980-12-15 00:00:00|0.488839|0.506696|0.450893|0.504464| 0.02252|122533600|
AAPL|-3.7933514757106477|-0.7157220870735409| -0.6798440607489181|-0.7965252181821475|
18.62389583605389| |1980-12-22 00:00:00|0.529018|0.636161|0.529018|0.633929| 0.0283| 46972800|
AAPL| -3.564893474332945|-0.6367328212376964| -0.4523036030236285|-0.6367328212376964|
17.66507926874505| |1980-12-29 00:00:00|0.642857|0.645089|0.609375|0.616071| 0.027503|
54863200| AAPL| -3.593460189350501|-0.4418329745012861|-0.43836698720922546|-
0.4953214372300254| 17.82035337206788| |1981-01-05
00:00:00|0.604911|0.604911|0.540179|0.569196| 0.02541| 49476000| AAPL|-3.6726124816500643|-
0.5026739392106726| -0.5026739392106726|-0.6158547128701838|17.716998261477038| |1981-01-12
00:00:00|0.569196|0.569196|0.544643|0.553571| 0.024712| 22125600| AAPL|-3.7004663233658563|-
0.5635304401980189| -0.5635304401980189|-0.6076247448267358|16.912245867134633| |1981-01-19
00:00:00|0.587054|0.591518|0.569196|0.584821| 0.026108| 33583200| AAPL| -3.645513498209166|-
0.5326384701994643| -0.5250631649193582|-0.5635304401980189|17.329536499891752| |1981-01-26
00:00:00|0.578125|0.578125|0.504464|0.504464| 0.02252| 41647200| AAPL|-3.7933514757106477|-
0.5479651707154474| -0.5479651707154474| -0.684258799514541| 17.54474469742654| |1981-02-02
00:00:00|0.477679|0.515625|0.475446|0.513393| 0.022919| 23144800| AAPL| -3.775789018104867|-
0.7388163201972744| -0.6623755218931916| -0.743501968114417|16.957280691234974| |1981-02-09
00:00:00|0.491071|0.491071|0.455357|0.455357| 0.020328| 18664800| AAPL| -3.895756032964274|-
0.7111665587902772| -0.7111665587902772|-0.7866735522963082|16.742149955030982| +---------------
----+--------+--------+--------+--------+---------+---------+------+-------------------+-------------------+--------------------+---
----------------+------------------+ only showing top 10 rows
```

```
w = Window().partitionBy("ticker").orderBy("date")
#Predictions with Adj close
df=df.withColumn("pdaily", sf.lag("Log_Adj_Close", 1).over(w))
df = df.withColumn("pweekly", sf.lag("Log_Adj_Close", 5).over(w))
df = df.withColumn("pbiweekly", sf.lag("Log_Adj_Close", 10).over(w))
df = df.withColumn("pmonthly", sf.lag("Log_Adj_Close", 21).over(w))
df = df.withColumn("pquarterly", sf.lag("Log_Adj_Close", 84).over(w))

#Returns
df = df.withColumn("log_return", -1* (sf.col("Log_Adj_Close") - sf.lag("Log_Adj_Close", 1).over(w)) /
sf.lag("Log_Adj_Close", 5).over(w))
df = df.withColumn("weekly_log_return", -1* (sf.col("Log_Adj_Close") - sf.lag("Log_Adj_Close",
5).over(w)) / sf.lag("Log_Adj_Close", 10).over(w))
df = df.withColumn("biweekly_log_return", -1* (sf.col("Log_Adj_Close") - sf.lag("Log_Adj_Close",
10).over(w)) / sf.lag("Log_Adj_Close", 21).over(w))
df = df.withColumn("monthly_log_return", -1* (sf.col("Log_Adj_Close") - sf.lag("Log_Adj_Close",
21).over(w)) / sf.lag("Log_Adj_Close", 42).over(w))
df = df.withColumn("quarterly_log_return", -1* (sf.col("Log_Adj_Close") - sf.lag("Log_Adj_Close",
84).over(w)) / sf.lag("Log_Adj_Close", 84).over(w))

#Volume
df = df.withColumn("daily_volume_diff", (sf.col("log_Volume") - sf.lag("log_Volume", 1).over(w)))
df = df.withColumn("weekly_volume_diff", (sf.col("log_Volume") - sf.lag("log_Volume", 5).over(w)))
df = df.withColumn("biweekly_volume_diff", (sf.col("log_Volume") - sf.lag("log_Volume",
10).over(w)))
df = df.withColumn("monthly_volume_diff", (sf.col("log_Volume") - sf.lag("log_Volume", 21).over(w)))
df = df.withColumn("quarterly_volume_diff", (sf.col("log_Volume") - sf.lag("log_Volume",
84).over(w)))

df=df.drop("ticker", "Close", "Adj_close", "Open", "High", "Low", "Volume")

# Removing Null values
df = df.na.drop()

# Converting to vector
featureassembler =
VectorAssembler(inputCols=['Log_Adj_Close','log_open','log_high','log_low','log_return','weekly_log_re
turn','biweekly_log_return',
'monthly_log_return','quarterly_log_return','daily_volume_diff', 'weekly_volume_diff',
                                    'biweekly_volume_diff',   'monthly_volume_diff','quarterly_volume_diff', ],
                                   outputCol="features")

output = featureassembler.transform(df)

df = output.select('Date', 'features','pdaily','pweekly','pbiweekly','pmonthly','pquarterly')

lr = LinearRegression(maxIter=10)
paramGrid = ParamGridBuilder()\
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.fitIntercept, [False, True])\
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
    .build()

#Splitting in to training 80% and 20%
tvs = TrainValidationSplit(estimator=lr,
                     estimatorParamMaps=paramGrid,
                     evaluator=RegressionEvaluator(),
                         trainRatio=0.8)

#Predictions -Daily
train_df_daily = df.withColumnRenamed("pdaily", "label").select("features", "label")
test_df_daily = df.withColumnRenamed("pdaily", "label").select("features", "label")



model = tvs.fit(train_df_daily)


lr_predictions = model.transform(test_df_daily)

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
                labelCol="label",metricName="rmse")

lr_predictions = model.transform(test_df_daily)

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
                labelCol="label")

#SMAPE

import numpy as np

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

Prediction = np.array(lr_predictions.select('prediction').rdd.map(lambda row : row[0]).collect())
label = np.array(lr_predictions.select('label').rdd.map(lambda row : row[0]).collect())

print(smape(label, Prediction))
10.37439546

#RMSE
def rmse(A, B):
    return np.sqrt(((A - B) ** 2).mean())
print(rmse(Prediction,label))
0.0852854651229

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(lr_predictions.select('label').rdd.map(lambda row : row[0]).collect(), label='True Data')

ax.plot(lr_predictions.select('prediction').rdd.map(lambda row : row[0]).collect(), label='Prediction')
plt.legend()
display(fig)

#Predictions Weekly
train_df_weekly = df.withColumnRenamed("pweekly", "label").select("features", "label")
test_df_weekly = df.withColumnRenamed("pweekly", "label").select("features", "label")
```

```
model = tvs.fit(train_df_weekly)


lr_predictions = model.transform(test_df_weekly)

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
                labelCol="label",metricName="rmse")
lr_predictions = model.transform(test_df_weekly)

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
                labelCol="label")

#SMAPE
import numpy as np

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

Prediction = np.array(lr_predictions.select('prediction').rdd.map(lambda row : row[0]).collect())
label = np.array(lr_predictions.select('label').rdd.map(lambda row : row[0]).collect())

print(smape(label, Prediction))
19.0970212138

#RMSE
def rmse(A, B):
    return np.sqrt(((A - B) ** 2).mean())
print(rmse(Prediction,label))
0.156825117212
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(lr_predictions.select('label').rdd.map(lambda row : row[0]).collect(), label='True Data')

ax.plot(lr_predictions.select('prediction').rdd.map(lambda row : row[0]).collect(), label='Prediction')
plt.legend()
display(fig)

#Predictions Biweekly
train_df_biweekly = df.withColumnRenamed("pbiweekly", "label").select("features", "label")
test_df_biweekly = df.withColumnRenamed("pbiweekly", "label").select("features", "label")

model = tvs.fit(train_df_biweekly)


lr_predictions = model.transform(test_df_biweekly)

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
                labelCol="label",metricName="rmse")
lr_predictions = model.transform(test_df_biweekly)
```

```
lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
            labelCol="label")

#SMAPE and RMSE
import numpy as np

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

Prediction = np.array(lr_predictions.select('prediction').rdd.map(lambda row : row[0]).collect())
label = np.array(lr_predictions.select('label').rdd.map(lambda row : row[0]).collect())

print(smape(label, Prediction))
25.2135552564

def rmse(A, B):
   return np.sqrt(((A - B) ** 2).mean())
print(rmse(Prediction,label))
0.218256477553

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(lr_predictions.select('label').rdd.map(lambda row : row[0]).collect(), label='True Data')

ax.plot(lr_predictions.select('prediction').rdd.map(lambda row : row[0]).collect(), label='Prediction')
plt.legend()
display(fig)

#Predictions Monthly
train_df_monthly = df.withColumnRenamed("pmonthly", "label").select("features", "label")
test_df_monthly = df.withColumnRenamed("pmonthly", "label").select("features", "label")

model = tvs.fit(train_df_monthly)


lr_predictions = model.transform(test_df_monthly)

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
            labelCol="label",metricName="rmse")
lr_predictions = model.transform(test_df_monthly)

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
            labelCol="label")

import numpy as np

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

Prediction = np.array(lr_predictions.select('prediction').rdd.map(lambda row : row[0]).collect())
label = np.array(lr_predictions.select('label').rdd.map(lambda row : row[0]).collect())
```

```
print(smape(label, Prediction))
33.2032824344

def rmse(A, B):
    return np.sqrt(((A - B) ** 2).mean())
print(rmse(Prediction,label))
0.312385810419

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(lr_predictions.select('label').rdd.map(lambda row : row[0]).collect(), label='True Data')

ax.plot(lr_predictions.select('prediction').rdd.map(lambda row : row[0]).collect(), label='Prediction')
plt.legend()
display(fig)
#Quarterly
train_df_quarterly = df.withColumnRenamed("pquarterly", "label").select("features", "label")
test_df_quarterly = df.withColumnRenamed("pquarterly", "label").select("features", "label")

model = tvs.fit(train_df_quarterly)


lr_predictions = model.transform(test_df_quarterly)

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
                labelCol="label",metricName="rmse")
lr_predictions = model.transform(test_df_quarterly)

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
                labelCol="label")
import numpy as np

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

Prediction = np.array(lr_predictions.select('prediction').rdd.map(lambda row : row[0]).collect())
label = np.array(lr_predictions.select('label').rdd.map(lambda row : row[0]).collect())

print(smape(label, Prediction))

def rmse(A, B):
    return np.sqrt(((A - B) ** 2).mean())
print(rmse(Prediction,label))
```

Moving Average

```
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.types import FloatType
from pyspark.sql.types import DoubleType
import matplotlib.pyplot as plt

average_days = 1
horizons = [1,5,10,21,84]
sqlContext = SQLContext(sc)
df = sqlContext.sql("SELECT aapl.date, aapl.close FROM aapl WHERE YEAR(aapl.date) BETWEEN
1980 AND 2018")
df.show()
+-------------------+--------+ | date| close| +-------------------+--------+ |1980-12-08 00:00:00|0.513393| |1980-12-
15 00:00:00|0.504464| |1980-12-22 00:00:00|0.633929| |1980-12-29 00:00:00|0.616071| |1981-01-05
00:00:00|0.569196| |1981-01-12 00:00:00|0.553571| |1981-01-19 00:00:00|0.584821| |1981-01-26
00:00:00|0.504464| |1981-02-02 00:00:00|0.513393| |1981-02-09 00:00:00|0.455357| |1981-02-16
00:00:00|0.433036| |1981-02-23 00:00:00|0.473214| |1981-03-02 00:00:00|0.457589| |1981-03-09
00:00:00|0.397321| |1981-03-16 00:00:00|0.459821| |1981-03-23 00:00:00|0.441964|

def get_mean_lag(n,horizon):
  df = sqlContext.sql("SELECT aapl.date, aapl.close,mean(aapl.close) OVER ( ORDER BY CAST(date
as timestamp) RANGE BETWEEN INTERVAL {} DAYS PRECEDING AND CURRENT ROW) AS
mean FROM aapl WHERE YEAR(aapl.date) BETWEEN 1980 AND 2018".format(n))

  partitionwindow = Window.orderBy("date")
  lags = 1
  lagtest = lag("mean",horizon,0).over(partitionwindow)
  df = df.withColumn("mean_lag{}".format(1),lagtest)
  df=df.withColumn("index", monotonically_increasing_id())
  df=df.filter(df["index"]>=horizon-1)
  df=df.where("mean_lag1!=0")
  return df

  import numpy as np
  def smape(mean,prediction):
    return 100/len(mean)*np.sum(2*np.abs(prediction-mean)/(np.abs(mean)+np.abs(prediction)))

  def rmse(A, B):
      return np.sqrt(((A - B) ** 2).mean())

  def get_metrics(df):
    prediction = np.array(df.select('mean_lag1').rdd.map(lambda row : row[0]).collect())
    close = np.array(df.select('close').rdd.map(lambda row : row[0]).collect())
    #print( "the smape when averaging {} days is {} for {} days
horizon".format(n,smape(close,prediction),horizon))
    return smape(close,prediction),rmse(close,prediction)

  def plot_prediction(df,horizon):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(df.select('mean_lag1').rdd.map(lambda row : row[0]).collect(), label='Close Price')
    ax.plot(df.select('close').rdd.map(lambda row : row[0]).collect(), label='Prediction')
    ax.set_title("close price and predtion for {0} days".format(horizon))
    ax.set_xlabel('date')
    ax.set_ylabel('price')
    plt.legend()
    display(fig)

  def plot_smape(smape_list,horizon):
    x = range(len(smape_list))
    fig, ax = plt.subplots()
    ax.plot(x,smape_list)
    ax.set_title("smape for horizon {0} day(s)".format(horizon))
    ax.set_xlabel('averaging days')
    ax.set_ylabel('smape')
    display(fig)

  def plot_rmse(rmse_list,horizon):
    x = range(len(rmse_list))
    fig, ax = plt.subplots()
    ax.plot(x,rmse_list)
    ax.set_title("rmse for horizon {0} day(s)".format(horizon))
    ax.set_xlabel('averaging days')
    ax.set_ylabel('rmse')
    display(fig)

  def get_metrics_container(m,horizons):
    smape_container=[]
    rmse_container=[]
    for horizon in horizons:
      smape_list=[]
      rmse_list=[]
      for i in range(m):
        df = get_mean_lag(i,horizon)
        smape,rmse=get_metrics(df)
        smape_list.append(smape)
        rmse_list.append(rmse)
      smape_container.append(smape_list)
      rmse_container.append(rmse_list)
    return smape_container,rmse_container

  def get_argmin(arr):
    arr = np.array(arr)
    return np.argmin(arr,axis=1)

  def get_min(arr):
    arr = np.array(arr)
    return np.amin(arr,axis=1)

  smape_container,rmse_container=get_metrics_container(80,horizons)
```

```
#SMAPE for whole horizons
smape_argmin=get_argmin(smape_container)
smape_min = get_min(smape_container)
print("smape for day,week,biweek,month,3 months")
print(smape_argmin)
print(smape_min)

#RMSE for whole horizons
smape_argmin=get_argmin(smape_container)
smape_min = get_min(smape_container)
print("smape for day,week,biweek,month,3 months")
print(smape_argmin)
print(smape_min)

#RMSE for whole horizons
rmse_argmin=get_argmin(rmse_container)
rmse_min = get_min(rmse_container)
print("rmse for day,week,biweek,month,3 months")
print(rmse_argmin)
print(rmse_min)
#Daily Plot
plot_prediction(get_mean_lag(smape_argmin[0],1),1)
plot_smape(smape_container[0],1)
plot_rmse(rmse_container[0],1)
#Weekly Plot
plot_prediction(get_mean_lag(smape_argmin[1],5),5)
plot_smape(smape_container[1],5)
plot_rmse(rmse_container[1],5)
#Biweekly Plot
plot_prediction(get_mean_lag(smape_argmin[2],10),10)
plot_smape(smape_container[2],10)
plot_rmse(rmse_container[2],10)
#Monthly Plot
plot_prediction(get_mean_lag(smape_argmin[3],21),21)
plot_smape(smape_container[3],21)
plot_rmse(rmse_container[3],21)
#Quarterly Plot
plot_prediction(get_mean_lag(smape_argmin[4],84),84)
plot_smape(smape_container[4],84)
plot_rmse(rmse_container[4],84)
```

Decision Tree

```
from pyspark.sql import SQLContext, Window
from pyspark.sql.functions import *
from pyspark.ml.regression import DecisionTreeRegressor
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
import pyspark.sql.functions as sf
```

```
 import numpy as np
 import matplotlib.pyplot as plt
 #read data
 df1= sqlContext.sql("SELECT aapl.date, aapl.close, max(aapl.close) OVER ( ORDER BY CAST(date
as timestamp) RANGE BETWEEN INTERVAL 2 DAYS PRECEDING AND CURRENT ROW) AS
max FROM aapl WHERE YEAR(aapl.date) BETWEEN 2000 AND 2010")
 df1.show()

 def get_mean_lag(lags):
   df0 = sqlContext.sql("SELECT aapl.date, aapl.low,aapl.high, aapl.close,mean(aapl.close) OVER
( ORDER BY CAST(date as timestamp) RANGE BETWEEN INTERVAL {} DAYS PRECEDING
AND CURRENT ROW) AS mean,min(aapl.close) OVER ( ORDER BY CAST(date as timestamp)
RANGE BETWEEN INTERVAL {} DAYS PRECEDING AND CURRENT ROW) AS
min,max(aapl.close) OVER ( ORDER BY CAST(date as timestamp) RANGE BETWEEN INTERVAL
{} DAYS PRECEDING AND CURRENT ROW) AS max FROM aapl WHERE YEAR(aapl.date)
BETWEEN 2000 AND 2010".format(lags,lags,lags))
   partitionwindow = Window.orderBy("date")
   lagtest = lag("close",lags,0).over(partitionwindow)
   df = df0.withColumn("lag{}".format(lags),lagtest)

   lagtest = lag("close",2*lags,0).over(partitionwindow)
   df = df.withColumn("lag{}".format(2*lags),lagtest)
   lagtest = lag("high",lags,0).over(partitionwindow)
   df = df.withColumn("high_lag{}".format(lags),lagtest)
   lagtest = lag("low",lags,0).over(partitionwindow)
   df = df.withColumn("low_lag{}".format(lags),lagtest)
   partitionwindow = Window.orderBy("date")
   df=df.withColumn("index", monotonically_increasing_id())
   lagtest = lag("mean",lags,0).over(partitionwindow)
   df = df.withColumn("mean_lag".format(lags),lagtest)
   lagtest = lag("max",lags,0).over(partitionwindow)
   df = df.withColumn("max_lag".format(lags),lagtest)
   lagtest = lag("min",lags,0).over(partitionwindow)
   df = df.withColumn("min_lag".format(lags),lagtest)

   df = df.withColumn("close_log",log(df["close"]))
   df = df.withColumn("lag{}_log".format(lags),log(df["lag{}".format(lags)]))
   df = df.withColumn("lag{}_log".format(2*lags),log(df["lag{}".format(2*lags)]))
   df = df.withColumn("high_lag{}_log".format(lags),log(df["high_lag{}".format(lags)]))
   df = df.withColumn("low_lag{}_log".format(lags),log(df["low_lag{}".format(lags)]))
   df = df.withColumn("mean_lag_log",log(df["mean_lag"]))
   df = df.withColumn("max_lag_log",log(df["max_lag".format(lags)]))
   df = df.withColumn("min_lag_log",log(df["min_lag".format(lags)]))

   df=df.filter(df["index"]>=2*lags+1)
   df=df.where("lag{}!=0".format(2*lags))

   df =
df.select("index","close_log","lag{}_log".format(lags),"lag{}_log".format(2*lags),"high_lag{}_log".for
mat(lags),"low_lag{}_log".format(lags),"mean_lag_log","max_lag_log","min_lag_log")
   return df
```

```
  df_daily = get_mean_lag(1)
  df_daily.show()
```

```
+-----+------------------+------------------+------------------+------------------+------------------+------------------+-------------------+------------------+ |index| close_log| lag1_log| lag2_log| high_lag1_log| low_lag1_log| mean_lag_log| max_lag_log| min_lag_log| +-----+------------------+------------------+------------------+------------------+------------------+------------------+------------------+------------------+ | 3|1.2890849789719805|1.3801369430700665|1.2773312527998073|1.4677098184490185|1.2773312527998073|1.3801369430700665|1.3801369430700665|1.3801369430700665| | 4|1.3499267539860522|1.2890849789719805|1.3801369430700665|1.4056373239630215|1.2791962255635234|1.2890849789719805|1.2890849789719805|1.2890849789719805| | 5| 1.356847270138411|1.3499267539860522|1.2890849789719805|1.3682757465262971|1.2163953243244932|1.3499267539860522|1.3499267539860522|1.3499267539860522| | 6|1.3795753389610308| 1.356847270138411|1.3499267539860522| 1.431037298686475|1.3306447496414526| 1.356847270138411| 1.356847270138411| 1.356847270138411|
```

```
  def pre_process(df,lags):
    #df = df.select("lag{}".format(lags),"lag{}".format(2*lags),"mean_lag","max_lag","min_lag","high_lag{}".format(lags),"low_lag{}".format(lags),"close")
    featureassembler = VectorAssembler(inputCols=["close_log","lag{}_log".format(lags),"lag{}_log".format(2*lags),"high_lag{}_log".format(lags),"low_lag{}_log".format(lags),"mean_lag_log","max_lag_log","min_lag_log"],outputCol="features")
    output = featureassembler.transform(df)
    finalized_data = output.select("features","close_log")
    return finalized_data

  df_daily_pre_process = pre_process(df_daily,1)
  df_daily_pre_process.show()
```

```
+--------------------+------------------+ | features| close_log| +--------------------+------------------+ |[1.28908497897198...|1.2890849789719805| |[1.34992675398605...|1.3499267539860522| |[1.35684727013841...| 1.356847270138411| |[1.37957533896103...|1.3795753389610308| |[1.37167907626873...|1.3716790762687363| |[1.51982584749440...|1.5198258474944089| |[1.50209120337720...|1.5020912033772047| |[1.49610929112709...|1.4961092911270952| |[1.60001869051760...|1.6000186905176088|
```

```
def model_training(df,train_ratio):
  dt_reg = DecisionTreeRegressor
  train_data,test_data = df.randomSplit([train_ratio,1-train_ratio])
  regressor = dt_reg(featuresCol="features",labelCol="close_log")
  regressor = regressor.fit(train_data)
  pred_results = regressor.transform(test_data)

  return pred_results,regressor.featureImportances

  pre_results_daily,feature_impotance_daily = model_training(df_daily_pre_process,0.7)
  print(feature_impotance_daily)
  pre_results_daily.show()
```

```
(8,[0,1,2,3],[0.998755888558,3.26428749308e-05,0.000922534131298,0.000288934435663]) +--------------------+--------------------+--------------------+ | features| close_log| prediction| +--------------------+--------------------+--------------------+ |[-0.0649193936603...|-0.06491939366030292|0.001383629201928...|
|[0.01206986523058...|0.012069865230580918|0.001383629201928...| |[0.03578115078867...|
0.03578115078867798| 0.06914449528295573| |[0.04674712620697...| 0.04674712620697081|
0.06914449528295573| |[0.05015019129408...|0.050150191294084094| 0.06914449528295573|
|[0.06230415179249...| 0.06230415179249471| 0.06914449528295573| |[0.11651200601811...|
0.11651200601811011| 0.06914449528295573|
```

```
def smape(Close, Prediction):
    return 100/len(Close) * np.sum(2 * np.abs(Prediction - Close) / (np.abs(Close) + np.abs(Prediction)))

def rmse(A, B):
    return np.sqrt(((A - B) ** 2).mean())

def get_metrice(pred_results):
  pred_results = pred_results.withColumn("squared_error",pow((col("close_log") - col("prediction")),2))
  Prediction = np.array(pred_results.select('prediction').rdd.map(lambda row : row[0]).collect())
  Close = np.array(pred_results.select('close_log').rdd.map(lambda row : row[0]).collect())
  return smape(Close, Prediction),rmse(Prediction,Close)

#Daily SMape and RMSE
smape_daliy,rmse_daily = get_metrice(pre_results_daily)
print(smape_daliy)
print(rmse_daily)

def plot_result(pred_results):
  fig = plt.figure()
  ax = fig.add_subplot(111)
  ax.plot(pred_results.select('close_log').rdd.map(lambda row : row[0]).collect(), label='Close Price')
  ax.plot(pred_results.select('prediction').rdd.map(lambda row : row[0]).collect(), label='Prediction')
  plt.legend()
  display(fig)

# weekly
df_weekly = get_mean_lag(5)
df_weekly_pre_process = pre_process(df_weekly,5)
pre_results_weekly,feature_impotance_weekly = model_training(df_weekly_pre_process,0.7)
print(feature_impotance_weekly)
smape_weekly,rmse_weekly = get_metrice(pre_results_weekly)
print(smape_weekly)
print(rmse_weekly)
plot_result(pre_results_weekly)

# biweekly
df_biweekly = get_mean_lag(5)
df_biweekly_pre_process = pre_process(df_biweekly,5)
pre_results_biweekly,feature_impotance_biweekly = model_training(df_biweekly_pre_process,0.7)
print(feature_impotance_biweekly)
smape_biweekly,rmse_biweekly = get_metrice(pre_results_biweekly)
print(smape_biweekly)
print(rmse_biweekly)
plot_result(pre_results_biweekly)
#Monthly
```

```
df_month = get_mean_lag(21)
df_month_pre_process = pre_process(df_month,21)
pre_results_month,feature_impotance_month = model_training(df_month_pre_process,0.7)
print(feature_impotance_month)
smape_month,rmse_month = get_metrice(pre_results_month)
print(smape_month)
print(rmse_month)
plot_result(pre_results_month)

#3month
df_4month = get_mean_lag(63)
df_4month_pre_process = pre_process(df_4month,63)
pre_results_4month,feature_impotance_4month = model_training(df_4month_pre_process,0.7)
print(feature_impotance_4month)
smape_4month,rmse_4month = get_metrice(pre_results_4month)
print(smape_4month)
print(rmse_4month)
plot_result(pre_results_4month)
```