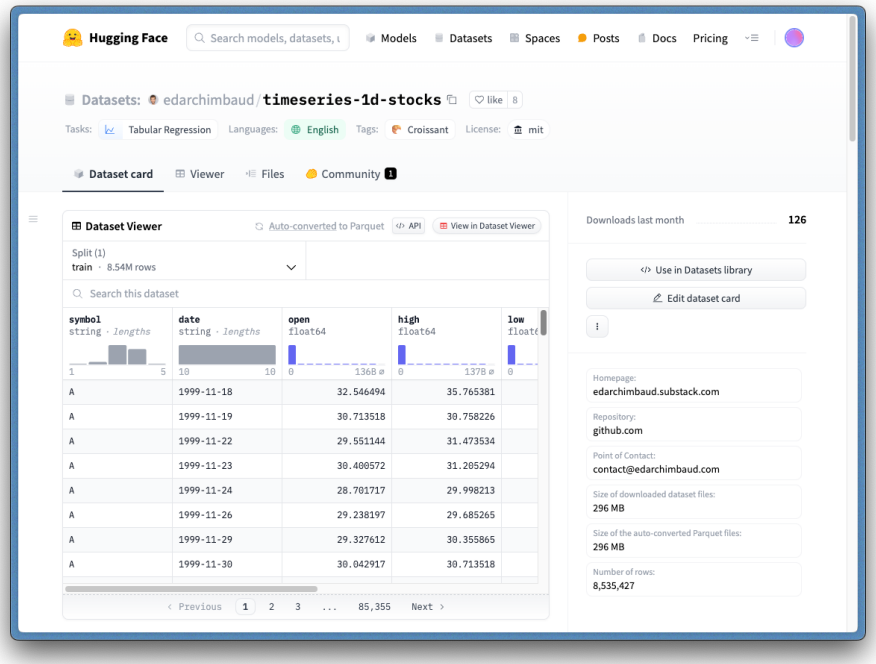
Supervised Learning Project Report

Rishabh Kaushick
College of Engineering
Northeastern University
Toronto, ON
kaushick.r@northeastern.edu

Abstract

In this project, I worked a time-series data to predict the closing price of Apple stock. I performed feature engineering through mathematical and unsupervised clustering approaches. I benchmarked and evaluated classical machine learning models – random forest regressors and XGBoost regressors along with deep learning LSTM model.

Dataset



Dataset	Name	Dataset Characteristics	Attribute Characteristics	Associated Task	Number of Instances	Number of Attributes
1	Time Series Forecasting of S&P 500 stocks	Multivariate	Real	Regression/ Forecasting	8,535,427	7

The dataset that I have chosen is from Hugging Face on the time-series data of the S&P 500 stocks over time.

What is this dataset interesting?

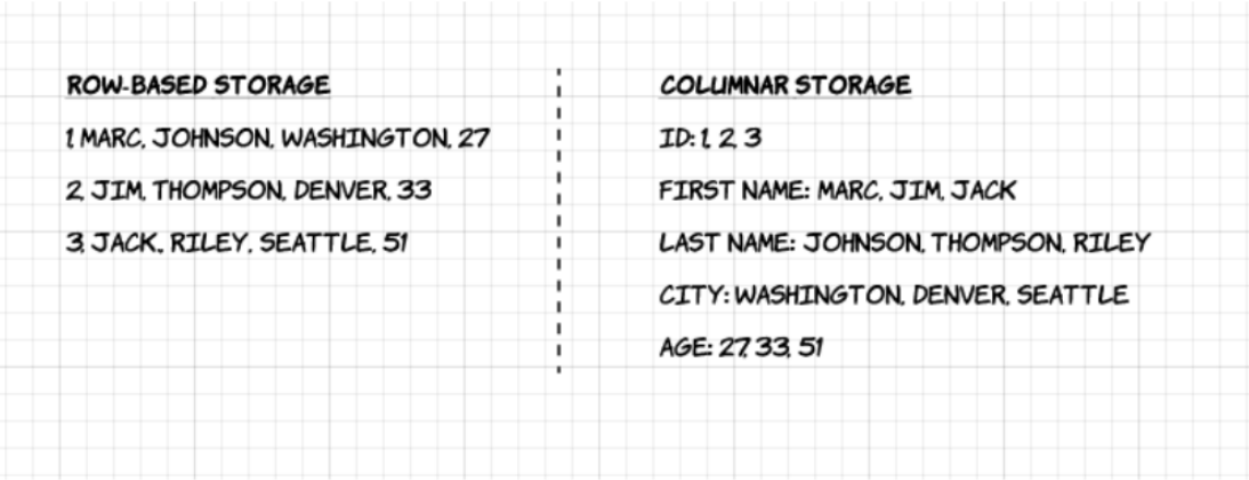
The S&P 500, i.e. Standard & Poor's 500 is a weighted stock market index of the 500 leading public companies in the United States of America. The companies present in S&P500 are from a wide range of industries like technology, healthcare, finance to name a few. I find this dataset interesting, because it has the details from 1980s until 2023 of all the companies, their opening, closing and volume of trade. From this we can see the performance of each of the companies and try to make a forecast, helping me to decide whether I should invest in one of the companies in the S&P 500 stock market index.

Step 0: Do We Need a Machine Learning Approach for This Problem?

Even before machine learning or deep learning models, we used to have statistical ways of understanding the data trend and making predictions. I believe that the prediction of the S&P 500 stock index is a problem where we are trying to find patterns and trends in the data. Therefore, this is a good application for machine learning and deep learning.

Loading The Dataset

Since we have sourced this dataset through Kaggle, I have used the huggingface_hub library. This dataset is saved in the Apache Parquet (.parquet*)* format, which stores data in the column-wise format as shown below. CSV on the other hand stores data in the row-wise format.



We need libraries *pyarrow* or *fastparquet* to convert this file into a pandas dataframe:

```python
from huggingface_hub import hf_hub_download
import pandas as pd
import pyarrow
import fastparquet

REPO_ID = "edarchimbaud/timeseries-1d-stocks"
FILENAME = "data/train-00000-of-00002-b533cd2d30403f22.parquet"

# since the data is in paraquet format, we read it using read_parquet() function in pandas
dataset = pd.read_parquet(
    hf_hub_download(repo_id=REPO_ID, filename=FILENAME, repo_type="dataset"),
    engine='pyarrow'
)
```

[4] ✓ 4.0s Python

Exploratory Data Analysis

While exploring the data, we can see that there is tremendous amount of data. Therefore, I have filtered the data to hold just the data of the top 10 stocks. This way I can better understand the type of data and trends of all the features, for the entire timeline of the top 10 companies.

```
# Since we are working on a huge amounts of data, let us start with the top 10 stocks first.

# Let us look at the big 5 stocks today - MSFT, AAPL, NVDA, AMZN, META
top10_stocks = ['MSFT', 'AAPL', 'NVDA', 'AMZN', 'GOOGL', 'TSLA', 'GOOG', 'BRK.B', 'META', 'UNH']

top10_stocks_df = dataframe[dataframe['symbol'].isin(top10_stocks)]
top10_stocks_df.head(-15)
```

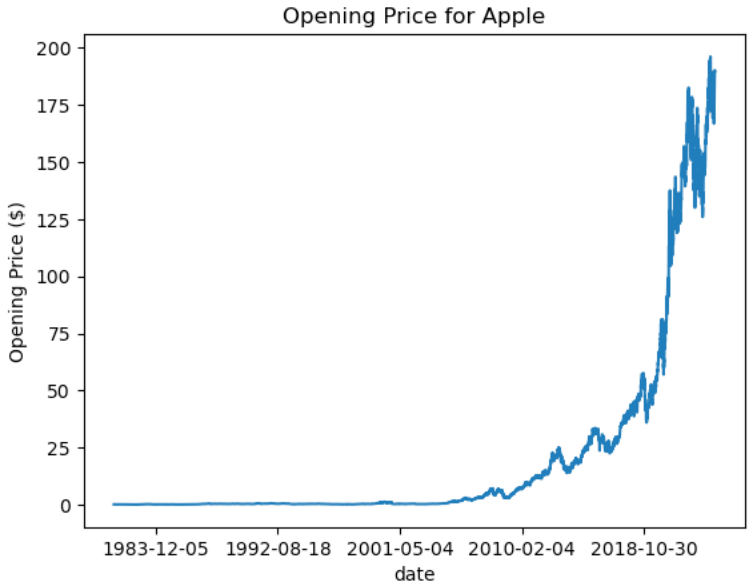
[116] ✓ 0.0s

date	symbol	open	high	low	close	adj_close	volume
1980-12-12	AAPL	0.128348	0.128906	0.128348	0.128348	0.099319	469033600.0
1980-12-15	AAPL	0.122210	0.122210	0.121652	0.121652	0.094137	175884800.0
1980-12-16	AAPL	0.113281	0.113281	0.112723	0.112723	0.087228	105728000.0
1980-12-17	AAPL	0.115513	0.116071	0.115513	0.115513	0.089387	86441600.0
1980-12-18	AAPL	0.118862	0.119420	0.118862	0.118862	0.091978	73449600.0
...	...	...	...	...	...	...	...
2023-10-24	UNH	522.859985	530.669983	522.070007	525.000000	525.000000	1978200.0
2023-10-25	UNH	527.270020	532.359985	520.080017	530.210022	530.210022	2380100.0
2023-10-26	UNH	525.700012	530.469971	522.520020	528.359985	528.359985	2675800.0
2023-10-27	UNH	525.989990	527.739990	521.260010	524.659973	524.659973	2585600.0
2023-10-30	UNH	525.000000	531.820007	522.940002	529.989990	529.989990	2555400.0

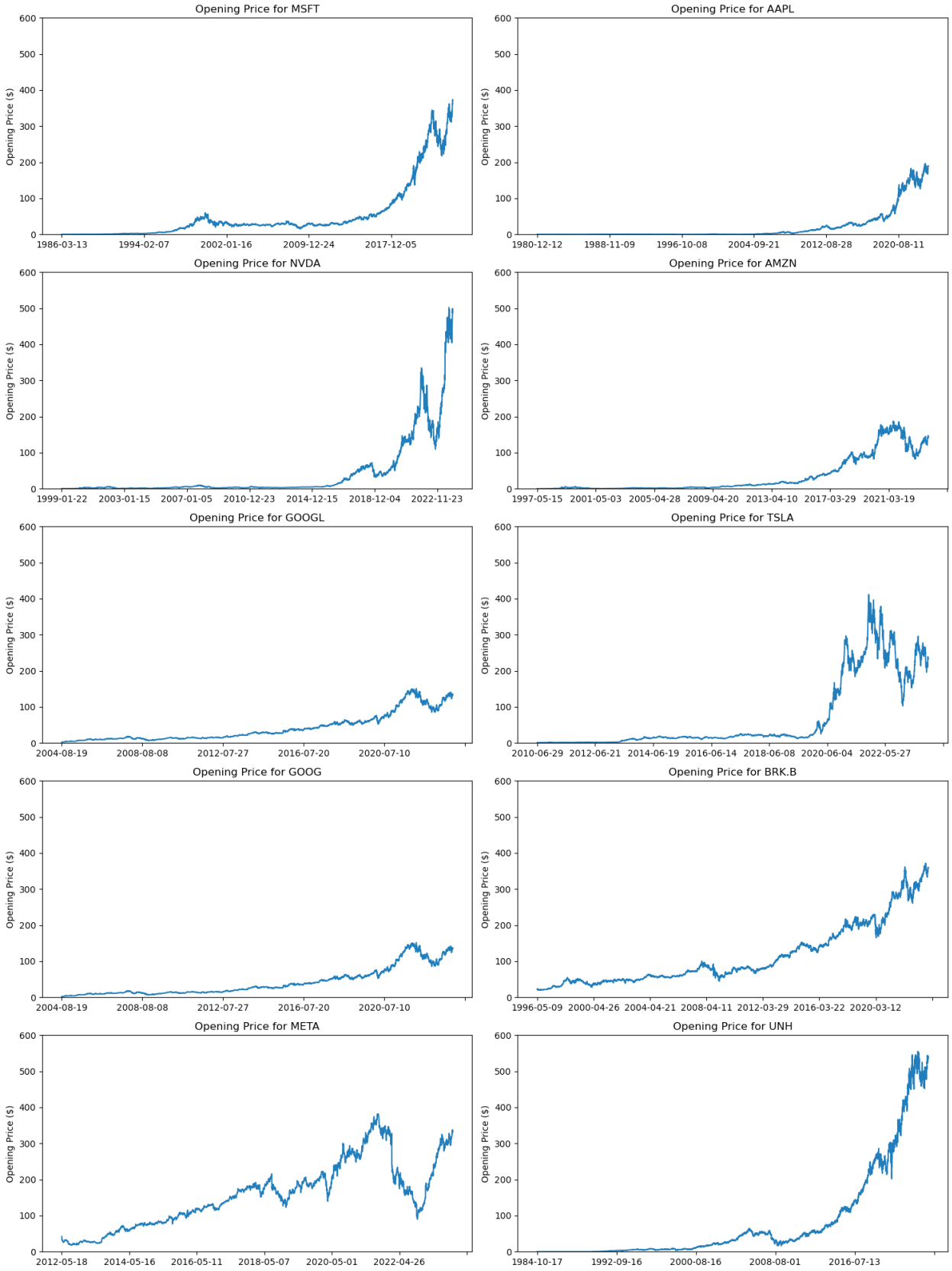
65983 rows × 7 columns

For just the top 10 stocks, we can see there are a total of 65,983 rows.

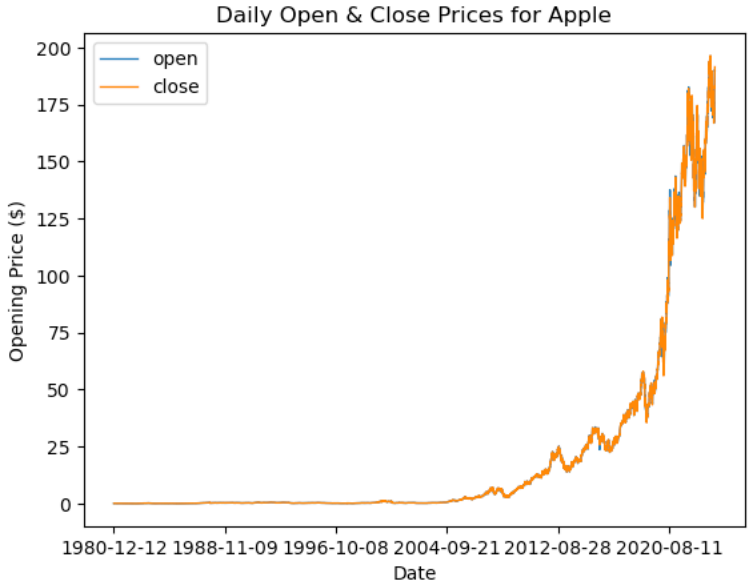
Let us start by visualizing the opening price specifically for Apple stocks over time. We can see below that the open price has significantly increased from 2018.



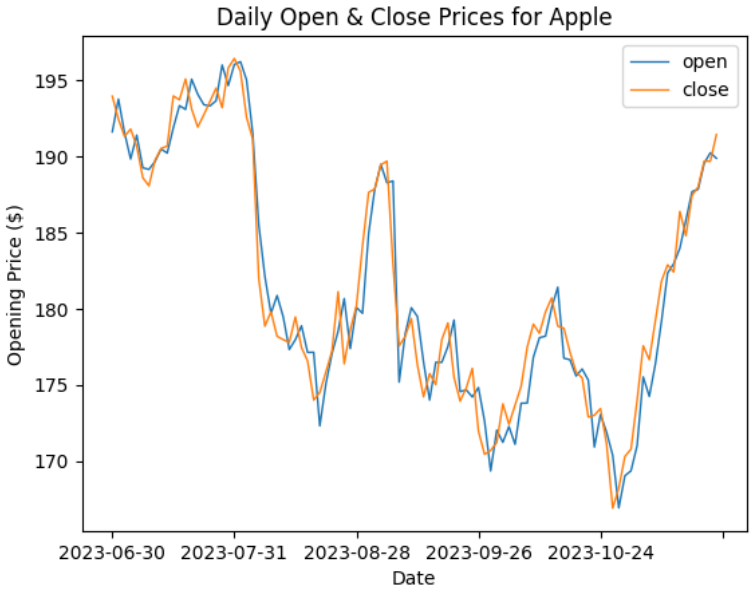
Similarly, let us look at the opening price for the top 10 stocks in the S&P 500:



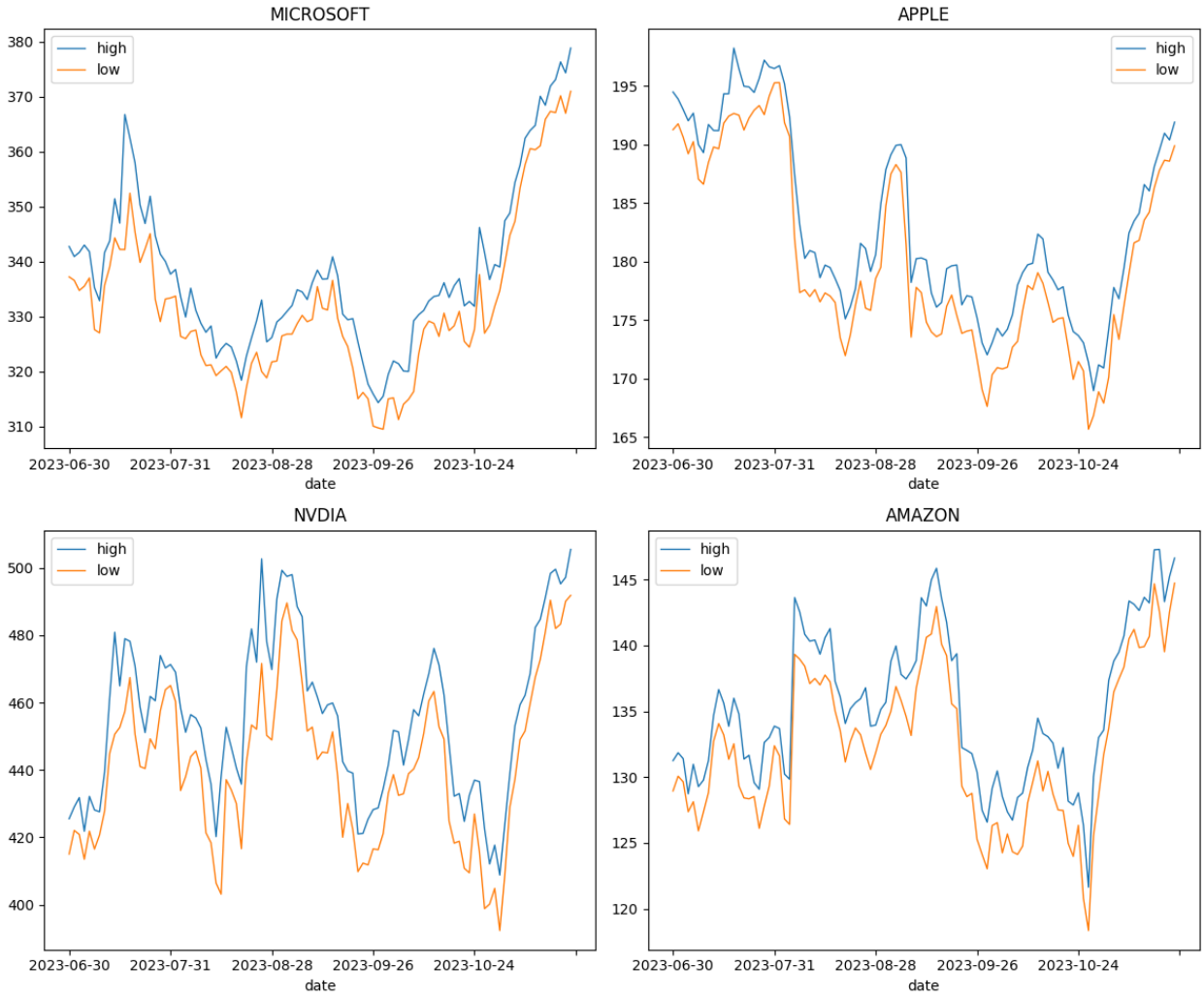
When we plot the open and close prices for Apple, we can see that the price change is not very different. The close price lies on the same price and sometimes higher than that of the open price.



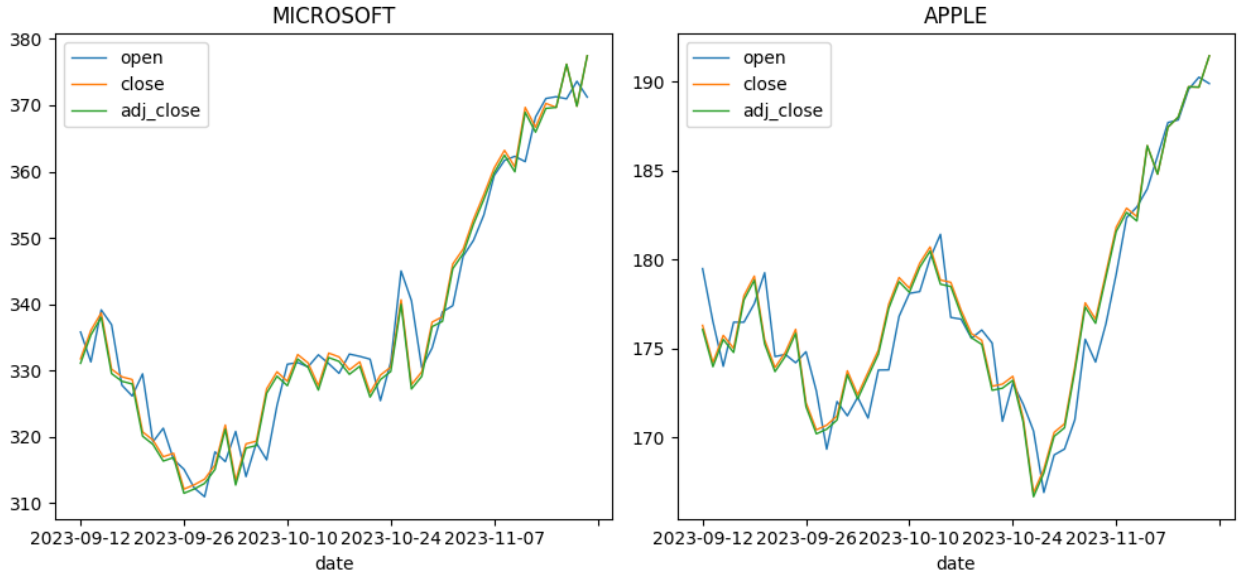
Zooming in on the same graph above shows us a better understanding of the data:



Let us try to see the difference there is based on the high and low prices of different stocks. Plotting a graph of the top 4 companies (as shown below). Like the above graph, we can see that the high and low daily price is nearly the same, without drastic differences.



Viewing the recent open, close, and adjusted closing price data for Microsoft & Apple:



From the above, we can see that Microsoft had a lower adjusted closing price (as shown in green line) compared to the actual closing price (as shown in orange line).

Correlation Matrix

While looking into the type of data, we can see that all the columns have high correlation between each other. Even from the graphs which we plotted above, all the values of the high, low, open, close and adj_close columns impact each other.

```
corr_matrix.style.background_gradient()
```

	open	high	low	close	adj_close	volume
open	1.000000	0.999903	0.999887	0.999796	0.998963	-0.240197
high	0.999903	1.000000	0.999833	0.999894	0.999086	-0.239625
low	0.999887	0.999833	1.000000	0.999895	0.999034	-0.240953
close	0.999796	0.999894	0.999895	1.000000	0.999165	-0.240290
adj_close	0.998963	0.999086	0.999034	0.999165	1.000000	-0.236320
volume	-0.240197	-0.239625	-0.240953	-0.240290	-0.236320	1.000000

We must fix this problem of this high correlated data. We will fix it during the feature engineering part of this project.

Defining The Problem

In this project, I will try to forecast the closing price for the stocks. I decided to do this because based on all the other parameters if we know the closing price each day and into the future, we can make a good decision whether we want to invest in a particular stock.

Data Pre-Processing

Handling Null Values

In this dataset, we can see that there are 202 null values.

```
null_rows_count = dataframe.isnull().sum(axis=1)
total_null_rows = (null_rows_count > 0).sum()
print("Total number of rows with at least one null value:", total_null_rows)
```

[193] ✓ 0.5s

```
Total number of rows with at least one null value: 202
```

```
rows_with_null = dataframe[dataframe.isnull().any(axis=1)]
rows_with_null.head(-10)
```

[196] ✓ 0.1s

date	symbol	open	high	low	close	adj_close	volume
1994-02-11	ATVI	NaN	NaN	NaN	NaN	NaN	NaN
1994-03-04	ATVI	NaN	NaN	NaN	NaN	NaN	NaN
1994-04-06	ATVI	NaN	NaN	NaN	NaN	NaN	NaN
1994-04-12	ATVI	NaN	NaN	NaN	NaN	NaN	NaN
1994-06-13	ATVI	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...
2023-10-30	RETA	NaN	NaN	NaN	NaN	NaN	NaN
2023-10-31	RETA	NaN	NaN	NaN	NaN	NaN	NaN
2023-11-01	RETA	NaN	NaN	NaN	NaN	NaN	NaN
2023-11-02	RETA	NaN	NaN	NaN	NaN	NaN	NaN
2023-11-03	RETA	NaN	NaN	NaN	NaN	NaN	NaN

192 rows × 7 columns

In a dataset where we have 8 million rows, we have 202 rows which have missing values. However, if we still want to impute the values, we can try using the forward fill (ffill) method. This way the last valid value of the columns will be used to fill the non-null values. We must make sure that the last value belongs to the same stock value. For this I have grouped the data using the 'symbol' column before using the ffill() method.

```
clean_df = dataframe.groupby('symbol')[['open', 'high', 'low', 'close', 'adj_close', 'volume']].ffill()
```

[216] ✓ 0.4s

```
null_rows_count = clean_df.isnull().sum(axis=1)
total_null_rows = (null_rows_count > 0).sum()
print("Total number of rows with at least one null value:", total_null_rows)
```

[217] ✓ 0.6s

```
Total number of rows with at least one null value: 0
```

Encoding Text Column

We only have one column – 'symbol' which refers to the code of the company. We will have at least 500 unique values. Furthermore, if some company recently enters the S&P 500, then there will be that many more companies in our dataset.

```
clean_df['symbol'].value_counts()
[223]  ✓ 0.1s

...  symbol
     SNA      11066
     DD       11066
     GE       11066
     GD       11066
     BK       11066
              ...
     CETY       212
     CVKD       211
     GNLX       207
     NFTG       193
     METCB      107
     Name: count, Length: 1490, dtype: int64
```

For this column, we cannot use label encoders as there is no inherent ordinal relationship in this column. Furthermore, we cannot use One-Hot-Encoders because there are too many unique data values making the model highly complex.

Binary Encoding for the Symbol Attribute

I have decided to use the binary encoding because it uses the concept of one hot encoding, however it is much more space efficient.

```
import category_encoders as ce

symbol_encoder = ce.BinaryEncoder(cols=['symbol'], return_df=True)
symbol_data_encoded = symbol_encoder.fit_transform(clean_df['symbol'])
symbol_data_encoded
```

[224] ✓ 5.5s Pyt

date	symbol_0	symbol_1	symbol_2	symbol_3	symbol_4	symbol_5	symbol_6	symbol_7	symbol_8
1999-11-18	0	0	0	0	0	0	0	0	0
1999-11-19	0	0	0	0	0	0	0	0	0
1999-11-22	0	0	0	0	0	0	0	0	0
1999-11-23	0	0	0	0	0	0	0	0	0
1999-11-24	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...
2023-11-14	1	0	1	1	1	0	1	0	0
2023-11-15	1	0	1	1	1	0	1	0	0
2023-11-16	1	0	1	1	1	0	1	0	0

Now I dropped the old Symbol column and added the data-frame containing the encoded values. However, I noticed that the encoded values were stored in int64 datatype. Since we have only 0 or 1 as values, we do not need such a large datatype, and therefore I changed the datatype to int8, saving more than 500 MB of memory space.

```
clean_df.info()
```

[226] ✓ 0.0s

```
<class 'pandas.core.frame.DataFrame'>
Index: 8535427 entries, 1999-11-18 to 2023-11-20
Data columns (total 18 columns):
 #   Column     Dtype
---  ------     -----
 0   symbol     object
 1   open       float64
 2   high       float64
 3   low        float64
 4   close      float64
 5   adj_close  float64
 6   volume     float64
 7   symbol_0   int64
 8   symbol_1   int64
 9   symbol_2   int64
 10  symbol_3   int64
 11  symbol_4   int64
 12  symbol_5   int64
 13  symbol_6   int64
 14  symbol_7   int64
 15  symbol_8   int64
 16  symbol_9   int64
 17  symbol_10  int64
dtypes: float64(6), int64(11), object(1)
memory usage: 1.2+ GB
```

```
clean_df.info()
```

[238] ✓ 0.0s

```
<class 'pandas.core.frame.DataFrame'>
Index: 8535427 entries, 1999-11-18 to 2023-11-20
Data columns (total 17 columns):
 #   Column     Dtype
---  ------     -----
 0   open       float64
 1   high       float64
 2   low        float64
 3   close      float64
 4   adj_close  float64
 5   volume     float64
 6   symbol_0   int8
 7   symbol_1   int8
 8   symbol_2   int8
 9   symbol_3   int8
 10  symbol_4   int8
 11  symbol_5   int8
 12  symbol_6   int8
 13  symbol_7   int8
 14  symbol_8   int8
 15  symbol_9   int8
 16  symbol_10  int8
dtypes: float64(6), int8(11)
memory usage: 577.6+ MB
```

Feature Engineering

Mathematical Approach

1. Daily Price Change

$$Daily\ Price\ Change = Close - Open$$

Daily Price Change

```
# Daily Price Change = Close - Open
clean_df['daily_price_change'] = clean_df['close']-clean_df['open']
clean_df[['close', 'open', 'daily_price_change']]
```

[239] ✓ 0.1s

date	close	open	daily_price_change
1999-11-18	31.473534	32.546494	-1.072960
1999-11-19	28.880545	30.713518	-1.832973
1999-11-22	31.473534	29.551144	1.922390
1999-11-23	28.612303	30.400572	-1.788269
1999-11-24	29.372318	28.701717	0.670601
...	...	...	...
2023-11-14	172.649994	171.410004	1.239990
2023-11-15	174.619995	172.490005	2.129990
2023-11-16	176.539993	175.029999	1.509994
2023-11-17	174.800003	177.410004	-2.610001
2023-11-20	176.059998	174.320007	1.739991

8535427 rows × 3 columns

2. Daily Price Percentage Change

$$Daily\ Price\ \%\ of\ Change = \frac{Close - Open}{Close} \times 100$$

Daily Price Percentage Change

```
clean_df['daily_price_change_percent'] = (clean_df['daily_price_change']/clean_df['close'])*100
clean_df[['close', 'open', 'daily_price_change', 'daily_price_change_percent']]
```

[240] ✓ 0.1s Python

date	close	open	daily_price_change	daily_price_change_percent
1999-11-18	31.473534	32.546494	-1.072960	-3.409087
1999-11-19	28.880545	30.713518	-1.832973	-6.346740
1999-11-22	31.473534	29.551144	1.922390	6.107957
1999-11-23	28.612303	30.400572	-1.788269	-6.250000
1999-11-24	29.372318	28.701717	0.670601	2.283105
...	...	...	...	...
2023-11-14	172.649994	171.410004	1.239990	0.718210
2023-11-15	174.619995	172.490005	2.129990	1.219786
2023-11-16	176.539993	175.029999	1.509994	0.855327
2023-11-17	174.800003	177.410004	-2.610001	-1.493136
2023-11-20	176.059998	174.320007	1.739991	0.988294

8535427 rows × 4 columns

3. Closing Price Moving Average

   Moving average is an indicator of market analysis to help determine trends. We calculate moving average using a window of data – in my case I will calculate it for 7 days, 30 days and 60 days. It is as follows:

Adjusted Closing Price Moving Average

```
moving_avg_day_list = [7, 30, 60]
new_df = pd.DataFrame()
for moving_avg in moving_avg_day_list:
    column = f'Moving Avg '+str(moving_avg)+' Days'
    new_df[column] = clean_df.groupby('symbol')['adj_close'].rolling(moving_avg).mean()
```

[58] ✓ 5.6s Python

```
new_df.head(10)
```

[59] ✓ 0.0s Python

		Moving Avg 7 Days	Moving Avg 30 Days	Moving Avg 60 Days
symbol	date			
A	1999-11-18	NaN	NaN	NaN
	1999-11-19	NaN	NaN	NaN
	1999-11-22	NaN	NaN	NaN
	1999-11-23	NaN	NaN	NaN
	1999-11-24	NaN	NaN	NaN
	1999-11-26	NaN	NaN	NaN
	1999-11-29	25.416827	NaN	NaN
	1999-11-30	25.259464	NaN	NaN
	1999-12-01	25.481941	NaN	NaN
	1999-12-02	25.492794	NaN	NaN

We have calculated the rolling average. However, we can see that for the Moving Avg 7 Days column - until the 7th record is NaN values. Similarly there are 30 & 60 NaNs in the last two columns respectively. I realized that the first 59 rows for each company will have NaN in the 'Moving Avg 60 Days' column. Therefore, let us check how many total rows have NaN values.

```
null_rows_count = clean_df.isnull().sum(axis=1)
total_null_rows = (null_rows_count > 0).sum()
print("Total number of rows with at least one null value:", total_null_rows)
```

[49] ✓ 0.9s Python

```
Total number of rows with at least one null value: 87910
```

We have 87,910 rows will null values. This is quite high number. Therefore we cannot drop all the records. We will fill the values into this but since the rolling window cannot be calculated, we will fill the value with '-1' suggesting to the algorithm that this is not a legitimate value.

+ Code + Markdown

```
clean_df.fillna(-1, inplace=True)
```

[50] ✓ 0.2s Python

```
null_rows_count = clean_df.isnull().sum(axis=1)
total_null_rows = (null_rows_count > 0).sum()
print("Total number of rows with at least one null value:", total_null_rows)
```

[51] ✓ 0.8s Python

```
Total number of rows with at least one null value: 0
```

Unsupervised Learning Approach

Feature Scaling using Min Max Scalar

Feature Scaling

Let us scale the data first using Min Max Scalar:

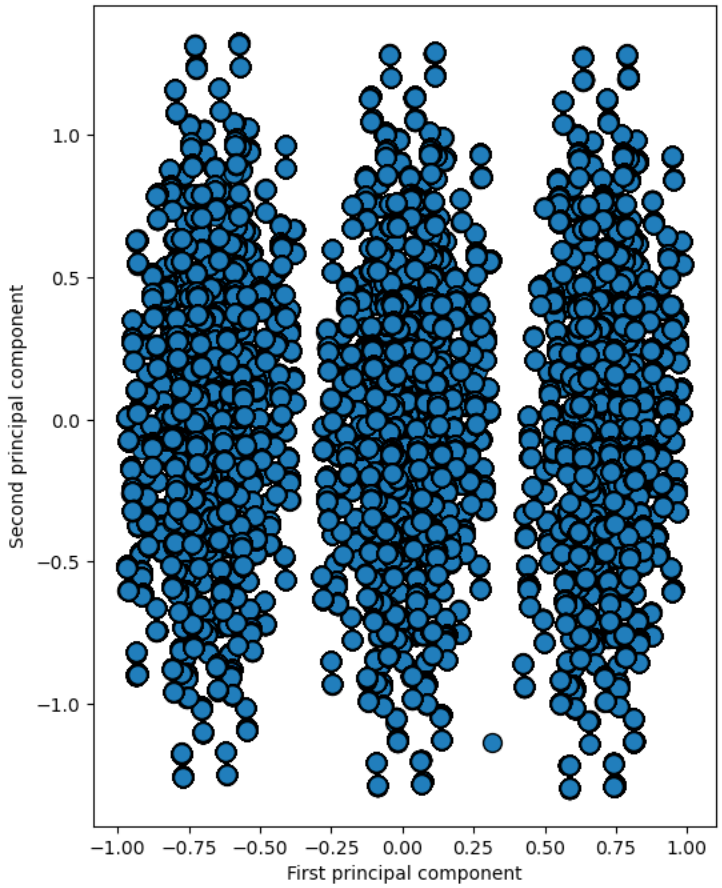
+ Code + Markdown

```
from sklearn.preprocessing import MinMaxScaler

scaled_df = pd.DataFrame()
# these do not need scaling:
scaled_df = clean_df[['symbol_0', 'symbol_1', 'symbol_2', 'symbol_3', 'symbol_4', 'symbol_5', 'symbol_6', 'symbol_7', 'symbol_8', 'sy

# rest of the columns need scaling:
scaler = MinMaxScaler()
scaled_df[['open', 'high', 'low', 'close', 'adj_close', 'volume', 'daily_price_change', 'Moving Avg 7 Days', 'Moving Avg 30 Days', 'M
scaled_df
```

[60] ✓ 2.0s Python

Performing Dimensionality Reduction using PCA

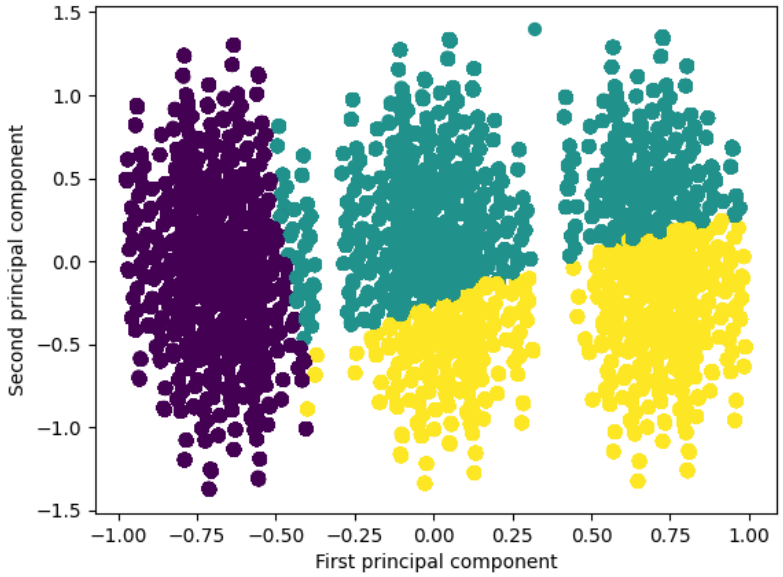
Before running the clustering, I have performed dimensionality reduction and plotted it against the first & second primary components.



From the above image, we can see that it creates three set of distinct clusters.

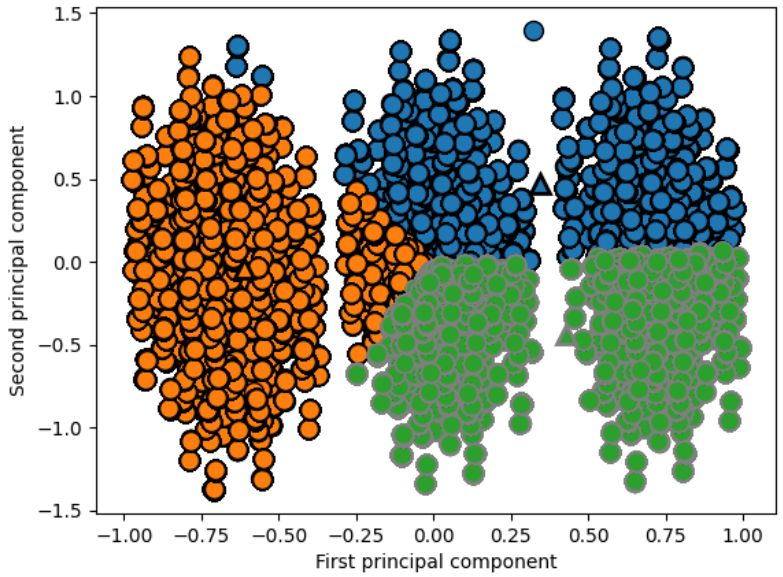
Using Gaussian Mixture Model (GMMs) for Clustering

I decided to use GMM model because unlike the K-Means models, it also takes into consideration the variance/ standard deviation in the data. However, the GMM model did not correctly identify the three clusters as I had hoped. After trying multiple different hyper-parameters for the initial cluster centers, it gave the graph as shown below.

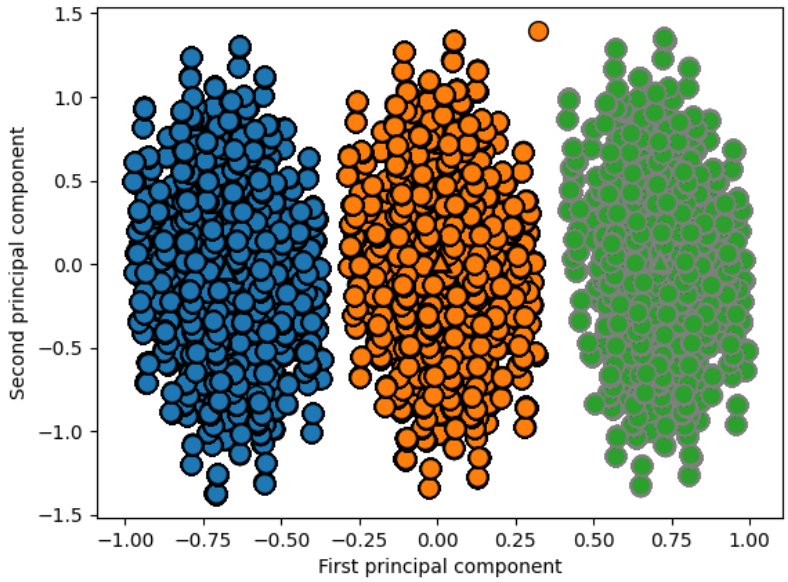


K-Means Clustering

I also tried to use the K-Means clustering but this too did not seem to correctly identify the clusters which are visible. This is still finding the local minimum and converging early.



In our situation, it is clearly visible that the cluster centers should be around (-0.75,0), (0.0, 0.0) and (0.75, 0). Therefore, I re-ran the k-means algorithm with values of the cluster centers and finally it gave me the below details, perfectly converging at the center of the three clusters.



As a part of the feature engineering, I will be using this value of the cluster id from the PCA data as a new feature.

Splitting The Data

In this project, I have created three sets:

1. Training set (First 70% i.e. 5,974,798 rows)
2. Validation Test (Next 15% i.e. 1,280,314 rows)
3. Testing set (Last 15% i.e. 1,280,314 rows)

I have decided to use 15% as the validation set. I will evaluate each model based on the validation set. And finally, after finding the best model, I will perform testing on the test set.

```
# Sorting the DataFrame by the datetime index
scaled_df = scaled_df.sort_index()

# Calculating the sizes for each set
total_size = len(scaled_df)
train_size = int(0.7 * total_size)
val_size = test_size = int(0.15 * total_size)

# Spliting the data into the three sets as mentioned above
train_set = scaled_df.iloc[:train_size]
val_set = scaled_df.iloc[train_size:train_size + val_size]
test_set = scaled_df.iloc[train_size + val_size:]

print(f"Training set size: {len(train_set)}")
print(f"Validation set size: {len(val_set)}")
print(f"Testing set size: {len(test_set)}")
```

[90] ✓ 0.8s

```
Training set size: 5974798
Validation set size: 1280314
Testing set size: 1280315
```

```
train_set.index
```

[93] ✓ 0.0s

```
Index(['1980-01-02', '1980-01-02', '1980-01-02', '1980-01-02', '1980-01-02',
       '1980-01-02', '1980-01-02', '1980-01-02', '1980-01-02', '1980-01-02',
       ...
       '2016-03-22', '2016-03-22', '2016-03-22', '2016-03-22', '2016-03-22',
       '2016-03-22', '2016-03-22', '2016-03-22', '2016-03-22', '2016-03-22'],
      dtype='object', name='date', length=5974798)
```

```
val_set.index
```

[96] ✓ 0.0s

```
Index(['2016-03-22', '2016-03-22', '2016-03-22', '2016-03-22', '2016-03-22',
       '2016-03-22', '2016-03-22', '2016-03-22', '2016-03-22', '2016-03-22',
       ...
       '2020-05-06', '2020-05-06', '2020-05-06', '2020-05-06', '2020-05-06',
       '2020-05-06', '2020-05-06', '2020-05-06', '2020-05-06', '2020-05-06'],
      dtype='object', name='date', length=1280314)
```

+ Code + Markdown

```
test_set.index
```

[94] ✓ 0.0s

```
Index(['2020-05-06', '2020-05-06', '2020-05-06', '2020-05-06', '2020-05-06',
       '2020-05-06', '2020-05-06', '2020-05-06', '2020-05-06', '2020-05-06',
       ...
       '2023-11-20', '2023-11-20', '2023-11-20', '2023-11-20', '2023-11-20',
       '2023-11-20', '2023-11-20', '2023-11-20', '2023-11-20', '2023-11-20'],
      dtype='object', name='date', length=1280315)
```

Model Selection

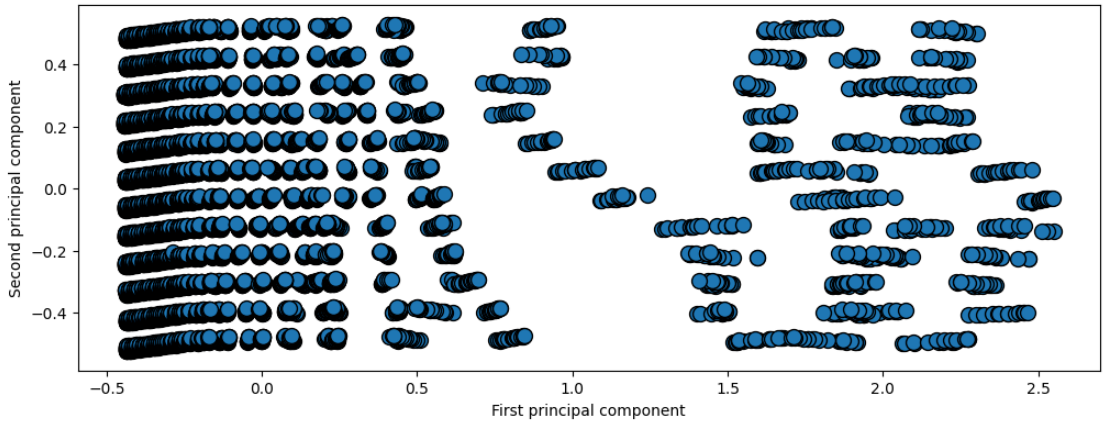
Classic Machine Learning Models

Random Forest Regressor

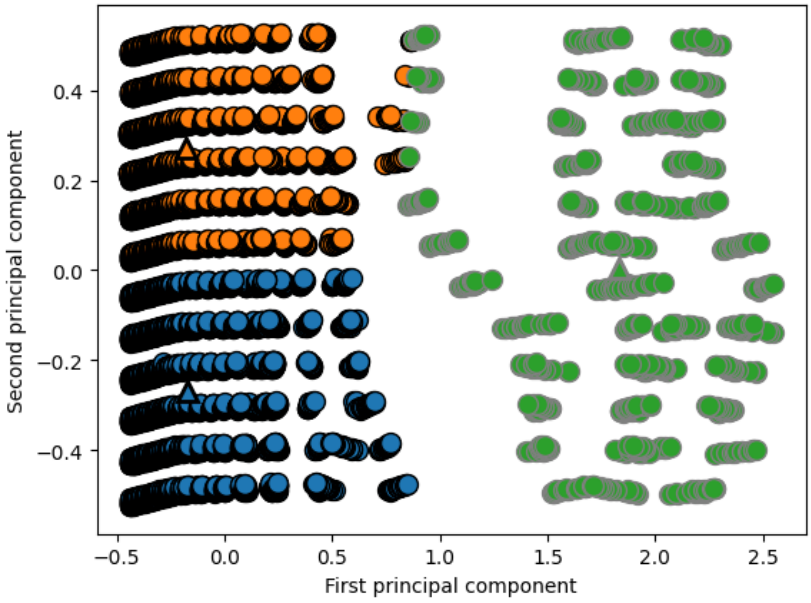
Random Forest is an ensemble learning approach which uses Bootstrap Aggregation (Bagging) ensemble learning approach.

After more than 2800 minutes of training the model (45 hours) I have no option but to abandon training the model on the entire dataset. Therefore, I will re-try to train the models on a subset of the data – only for 1 company (Apple).

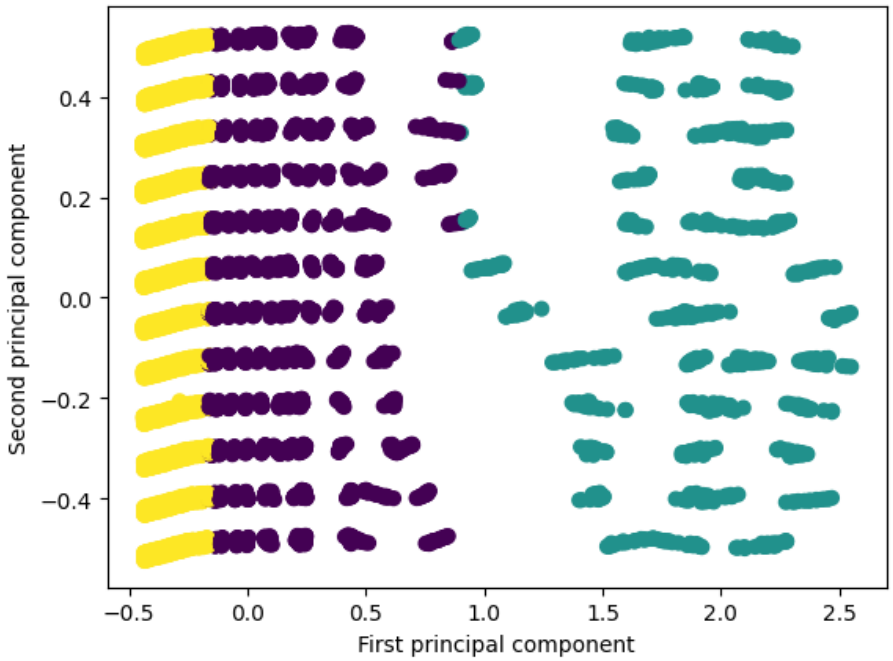
Dimensionality Reduction Again using PCA (on Apple's PCA Data Points)



K-Means Clustering Again on (Apple's PCA Data Points)



Gaussian Mixture Model Again (GMMs) for Clustering (on Apple data points)



We can see that the Gaussian mixture model has performed better as compared to the K-Means model as shown above. Therefore, let us use the Cluster ID from this GMM model as a new feature for our supervised learning model.

We can see that now the data has reduced to 34,219 records, and we have taken the timestamp from January 2010 to November 2023 (until the last data point).

Updating The Data Split

[82]:
```
# Calculating the sizes for each set
total_size = len(scaled_df)
train_size = int(0.7 * total_size)
val_size = test_size = int(0.15 * total_size)

# Spliting the data into the three sets as mentioned above
train_set = scaled_df.iloc[:train_size]
val_set = scaled_df.iloc[train_size:train_size + val_size]
test_set = scaled_df.iloc[train_size + val_size:]

print(f"Training set size: {len(train_set)}")
print(f"Validation set size: {len(val_set)}")
print(f"Testing set size: {len(test_set)}")
```

```
Training set size: 7536
Validation set size: 1615
Testing set size: 1616
```

Let us split the data again into train (7.536 rows), validation (1,615 rows), and test (1,616 rows) sets.

Fixing Highly Correlated Columns

[89]:
```
corr_matrix = scaled_df.corr()
corr_matrix.style.background_gradient()
```

[89]:

	daily_price_change_percent	open	high	low	close	adj_close	volume	daily_price_change
daily_price_change_percent	1.000000	0.020937	0.023096	0.023402	0.025639	0.025450	-0.009513	0.285092
open	0.020937	1.000000	0.999946	0.999934	0.999863	0.999624	-0.241050	0.048940
high	0.023096	0.999946	1.000000	0.999921	0.999935	0.999704	-0.240627	0.056508
low	0.023402	0.999934	0.999921	1.000000	0.999938	0.999701	-0.241735	0.057429
close	0.025639	0.999863	0.999935	0.999938	1.000000	0.999768	-0.241195	0.065456
adj_close	0.025450	0.999624	0.999704	0.999701	0.999768	1.000000	-0.243077	0.065858
volume	-0.009513	-0.241050	-0.240627	-0.241735	-0.241195	-0.243077	1.000000	-0.022526
daily_price_change	0.285092	0.048940	0.056508	0.057429	0.065456	0.065858	-0.022526	1.000000
Moving Avg 7 Days	0.021851	0.999734	0.999738	0.999670	0.999649	0.999426	-0.240769	0.052065
Moving Avg 30 Days	0.021415	0.998353	0.998418	0.998209	0.998255	0.998076	-0.239888	0.051161
Moving Avg 60 Days	0.021939	0.996722	0.996841	0.996580	0.996674	0.996554	-0.239606	0.054120
year	0.032496	0.667817	0.667661	0.667850	0.667737	0.656258	0.107789	0.033346
month	-0.015449	0.017451	0.017260	0.017460	0.017327	0.017341	-0.059727	-0.006468
day	-0.017023	-0.001341	-0.001283	-0.001148	-0.001249	-0.001242	-0.011081	0.005452
Cluster_ID	-0.015587	-0.370603	-0.370204	-0.370818	-0.370477	-0.352928	-0.039197	-0.013630

I have dropped the columns which had high correlation between each other, such as:

1. Open
2. High
3. Low
4. Adj_Close
5. Moving Avg 7 Days
6. Moving Avg 30 Days

Now the correlation matrix is as follows. Althought there is still high correlation between a few columns, this is okay for now.

```
corr_matrix = new_scaled_df.corr()
corr_matrix.style.background_gradient()
```

[91]:

	daily_price_change_percent	close	volume	daily_price_change	Moving Avg 60 Days	year	month	day	Cluster_ID
daily_price_change_percent	1.000000	0.025639	-0.009513	0.285092	0.021939	0.032496	-0.015449	-0.017023	-0.015587
close	0.025639	1.000000	-0.241195	0.065456	0.996674	0.667737	0.017327	-0.001249	-0.370477
volume	-0.009513	-0.241195	1.000000	-0.022526	-0.239606	0.107789	-0.059727	-0.011081	-0.039197
daily_price_change	0.285092	0.065456	-0.022526	1.000000	0.054120	0.033346	-0.006468	0.005452	-0.013630
Moving Avg 60 Days	0.021939	0.996674	-0.239606	0.054120	1.000000	0.666459	0.016810	-0.000688	-0.369838
year	0.032496	0.667737	0.107789	0.033346	0.666459	1.000000	-0.014740	-0.002395	-0.750839
month	-0.015449	0.017327	-0.059727	-0.006468	0.016810	-0.014740	1.000000	-0.002222	-0.012306
day	-0.017023	-0.001249	-0.011081	0.005452	-0.000688	-0.002395	-0.002222	1.000000	0.002922
Cluster_ID	-0.015587	-0.370477	-0.039197	-0.013630	-0.369838	-0.750839	-0.012306	0.002922	1.000000

Re-Running Random Forest Regression Model

```
[92]: from sklearn.ensemble import RandomForestRegressor

rf_regressor = RandomForestRegressor(n_estimators=100, max_depth=10, random_state=23, verbose=1, criterion='friedman_mse', oob_score=True)
rf_regressor.fit(train_X, train_y)
```

```
[Parallel(n_jobs=1)]: Done  49 tasks      | elapsed:    1.5s
```

```
[92]: RandomForestRegressor
RandomForestRegressor(criterion='friedman_mse', max_depth=10, oob_score=True,
                      random_state=23, verbose=1)
```

```
[93]: train_predicted = rf_regressor.predict(train_X)
print(train_predicted)
```

```
[0.00025988 0.00024097 0.00025988 ... 0.06272762 0.06163123 0.0616921 ]
[Parallel(n_jobs=1)]: Done  49 tasks      | elapsed:    0.0s
```

Let us compare the real values and the predicted value for the closing price:

```
[99]: close_price = pd.DataFrame()
close_price['real close'] = train_set['close']
close_price['pred close'] = train_predicted
close_price.head(-10)
```

[99]:

date	real close	pred close
1981-03-10	0.000261	0.000260
1981-03-11	0.000242	0.000241
1981-03-12	0.000261	0.000260
1981-03-13	0.000256	0.000255
1981-03-16	0.000276	0.000274
...	...	...
2010-12-29	0.058902	0.058916
2010-12-30	0.058606	0.058669
2010-12-31	0.058406	0.058485
2011-01-03	0.059680	0.059670
2011-01-04	0.059993	0.060054

7526 rows × 2 columns

From the above image, we can see that the predicted values are very close to the actual data.

Metrics to Evaluate the Random Forest Model

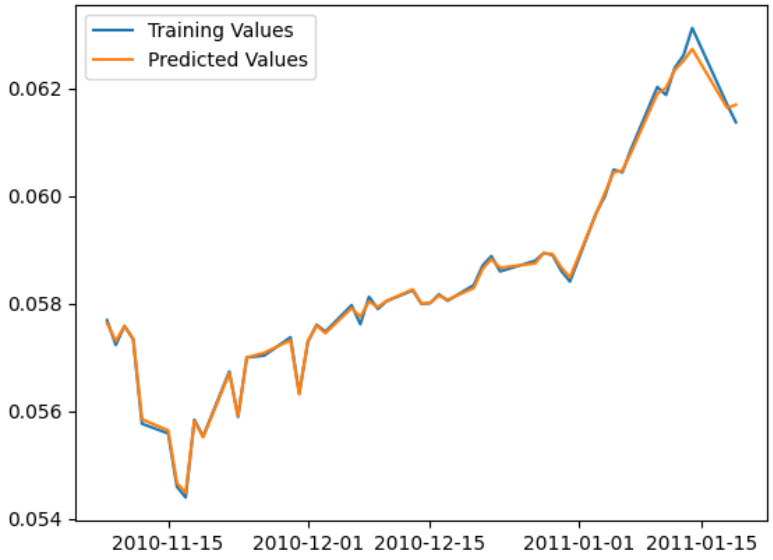
To understand how well this model performs, I have used mean squared error, root mean squared error and r squared and adjusted r squared metrics. Evaluating training prediction:

```
[101]: from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
rf_mean_sq_err = mean_squared_error(train_y, train_predicted)
r2 = r2_score(train_y,train_predicted)
print(f"The Mean Squared Error for Training Set is {rf_mean_sq_err}")
print(f"The Root Mean Squared Error for Training Set is {np.sqrt(rf_mean_sq_err)}")
print(f"The R2 score for Training Set is {r2}")
```

```
The Mean Squared Error for Training Set is 3.961628827077018e-10
The Root Mean Squared Error for Training Set is 1.9903840903396055e-05
The R2 score for Training Set is 0.9999967815350055
```

The mean squared error for the training set is very low at 3.96×10^{-10} and the root mean squared error is also very low at 1.99×10^{-5}. The R2 score is 0.999. This means that the training model can explain 99% of variance in the data. When the value of R2 score is 1, it means that the model has overfit on the data. Since our R2 score is almost 1, there is a high chance it has overfit on the data.

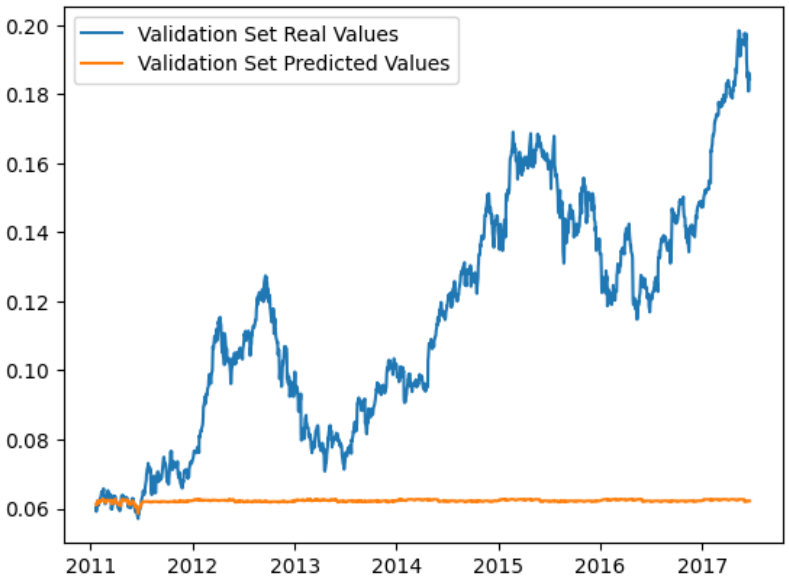
Plotting the training and predicted values together in a graph:



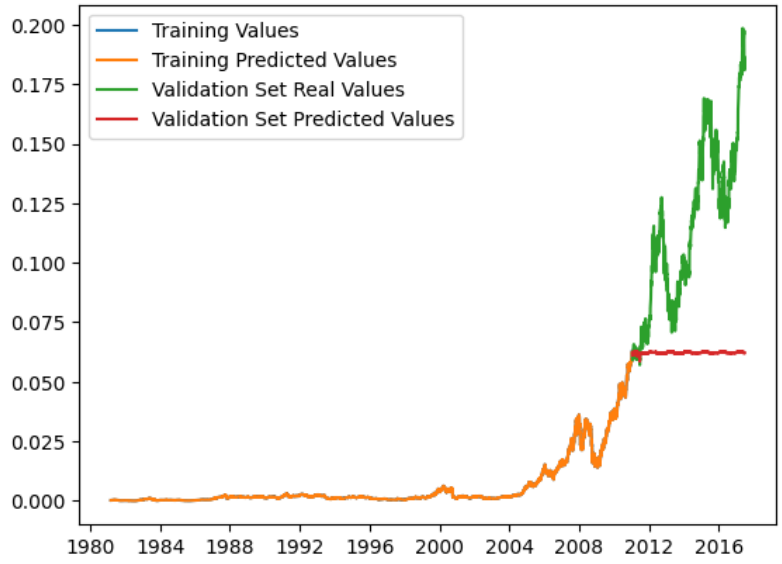
Evaluating validation set predictions:

```
[108]: # Let us now predict the data for the validation set
val_predicted = rf_regressor.predict(val_X)
rf_mean_sq_err = mean_squared_error(val_y, val_predicted)
r2 = r2_score(val_y,val_predicted)
print(f"The Mean Squared Error for Validation Set is {rf_mean_sq_err}")
print(f"The Root Mean Squared Error for Validation Set is {np.sqrt(rf_mean_sq_err)}")
print(f"The R2 score for Validation Set is {r2}")
```

```
The Mean Squared Error for Validation Set is 0.003964161198197244
The Root Mean Squared Error for Validation Set is 0.06296158509914791
The R2 score for Validation Set is -2.3502508925967356
[Parallel(n_jobs=1)]: Done  49 tasks      | elapsed:    0.0s
```



Initially, the model was working great in the training set but, in the prediction, set, we can see that this has worked very poorly. Plotting the training and validation line plots in one graph:



My intuition is that the training dataset does not generalize well to the validation data set. The data of the closing price from 1980 up to around 2011 is mostly flat with only a few sharp increases. However, from 2012 we can see with the green line that the closing price for Apple stock is sharply increasing.

Random Forest Regression Model 2

To fix the previous random forest model we will go back and recreate our train, validation, and test split but this time we will take the data only from 2018 onwards.

[110]:
```
# disregarding data before 2018.
new_scaled_df = new_scaled_df[pd.to_datetime('2017-12-31 00:00:00'):]
new_scaled_df.head(-10)
```

[110]:

date	daily_price_change_percent	close	volume	daily_price_change	Moving Avg 60 Days	year	month	day	Cluster_ID
2018-01-02	0.012191	0.219021	0.013774	0.441125	0.225159	0.880952	0.000000	0.033333	0
2018-01-03	-0.001742	0.218983	0.015909	0.406322	0.225538	0.880952	0.000000	0.066667	0
2018-01-04	0.002832	0.220001	0.012091	0.417779	0.225922	0.880952	0.000000	0.100000	0
2018-01-05	0.008914	0.222509	0.012752	0.433295	0.226349	0.880952	0.000000	0.133333	0
2018-01-08	0.000000	0.221681	0.011085	0.410673	0.226747	0.880952	0.000000	0.233333	0
...	...	...	...	...	...	...	...	...	...
2023-10-31	0.008315	0.869247	0.006043	0.493039	0.948266	1.000000	0.818182	1.000000	1
2023-11-01	0.017072	0.885540	0.007671	0.582947	0.947745	1.000000	0.909091	0.000000	1
2023-11-02	0.011545	0.903870	0.010420	0.529583	0.947690	1.000000	0.909091	0.033333	1
2023-11-03	0.013643	0.899186	0.010747	0.550463	0.947572	1.000000	0.909091	0.066667	1
2023-11-06	0.015901	0.912322	0.008602	0.575986	0.947700	1.000000	0.909091	0.166667	1

1472 rows × 9 columns

Random Forest Regressor Model 2

[116]:
```
rf_regressor2 = RandomForestRegressor(n_estimators=20, max_depth=5, random_state=23, verbose=0, criterion='friedman_mse', oob_score=True)
rf_regressor2.fit(train_X, train_y)
```

[116]:
```
RandomForestRegressor(criterion='friedman_mse', max_depth=5, n_estimators=20,
                      oob_score=True, random_state=23)
```

[118]:
```
# Model predicting values on Training Set
rf2_train_predicted = rf_regressor2.predict(train_X)

# Evaluating rf_regressor2 model
rf_mean_sq_err = mean_squared_error(train_y, rf2_train_predicted)
r2 = r2_score(train_y,rf2_train_predicted)
print('Evaluating rf_regressor2 model:')
print(f"The Mean Squared Error for Training Set is {rf_mean_sq_err}")
print(f"The Root Mean Squared Error for Training Set is {np.sqrt(rf_mean_sq_err)}")
print(f"The R2 score for Training Set is {r2}")
```

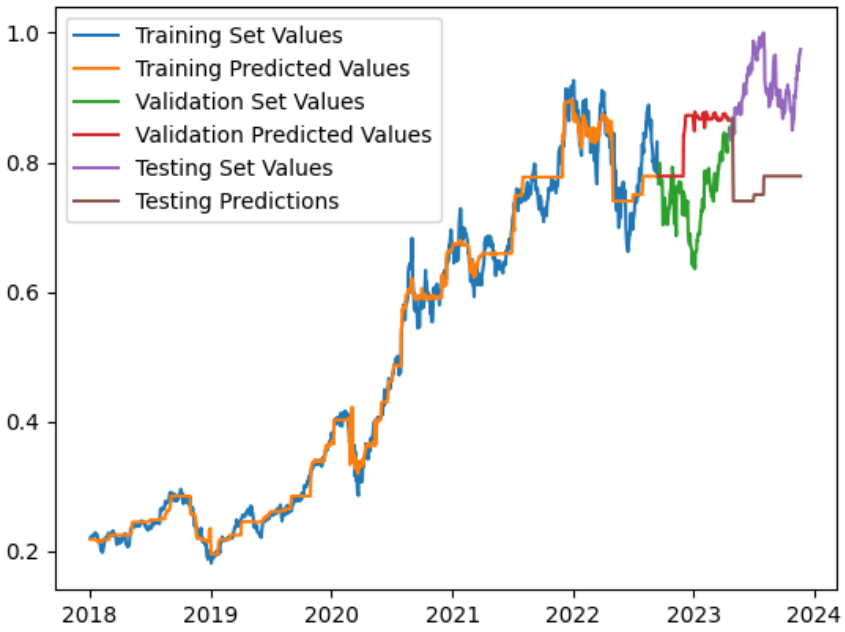
```
Evaluating rf_regressor2 model:
The Mean Squared Error for Training Set is 0.0004669893312075796
The Root Mean Squared Error for Training Set is 0.02160993593714659
The R2 score for Training Set is 0.9915586482080094
```

[120]:
```
# Model predicting values on Validation Set
rf2_val_predicted = rf_regressor2.predict(val_X)

# Evaluating rf_regressor2 model
rf_mean_sq_err = mean_squared_error(val_y, rf2_val_predicted)
r2 = r2_score(val_y,rf2_val_predicted)
print('Evaluating rf_regressor2 model on validation set:')
print(f"The Mean Squared Error for Validation Set is {rf_mean_sq_err}")
print(f"The Root Mean Squared Error for Validation Set is {np.sqrt(rf_mean_sq_err)}")
print(f"The R2 score for Validation Set is {r2}")
```

```
Evaluating rf_regressor2 model on validation set:
The Mean Squared Error for Validation Set is 0.011779656266548946
The Root Mean Squared Error for Validation Set is 0.10853412489419605
The R2 score for Validation Set is -3.878194625298546
```

```python
# Model predicting values on Test Set
rf2_test_predicted = rf_regressor2.predict(test_X)

# Evaluating rf_regressor2 model
rf_mean_sq_err = mean_squared_error(test_y, rf2_test_predicted)
r2 = r2_score(test_y,rf2_test_predicted)
print('Evaluating rf_regressor2 model on test set:')
print(f"The Mean Squared Error for Test Set is {rf_mean_sq_err}")
print(f"The Root Mean Squared Error for Test Set is {np.sqrt(rf_mean_sq_err)}")
print(f"The R2 score for Test Set is {r2}")
```

```
Evaluating rf_regressor2 model on test set:
The Mean Squared Error for Test Set is 0.02517503785379102
The Root Mean Squared Error for Test Set is 0.1586664358135993
The R2 score for Test Set is -13.95744422945438
```



This model is better than the previous. But still it looks like it is overfitting data in the training an unable to properly predict accurate values in the validation & test sets.

XGBoost Regressor

[124]:
```python
import xgboost as xgb

xgboost_regressor = xgb.XGBRegressor(n_estimators = 20, max_depth = 5)
xgboost_regressor.fit(train_X, train_y)
```

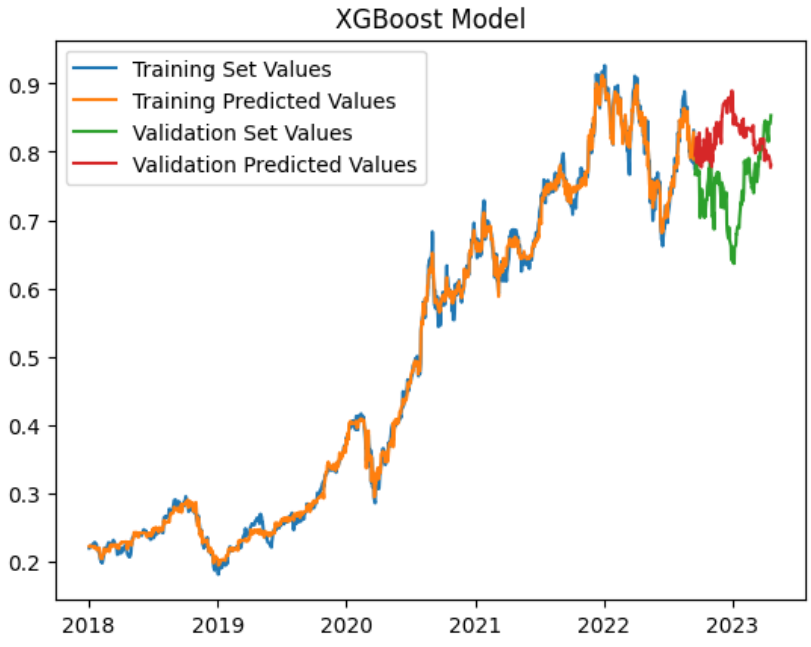
[124]:
```
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None, early_stopping_rounds=None,
             enable_categorical=False, eval_metric=None, feature_types=None,
             gamma=None, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=None, max_bin=None,
             max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=5, max_leaves=None,
             min_child_weight=None, missing=nan, monotone_constraints=None,
             multi_strategy=None, n_estimators=20, n_jobs=None,
             num_parallel_tree=None, random_state=None, ...)
```

[131]:
```
# XGBoost Model predicting values on Training Set
xgb_train_predicted = xgboost_regressor.predict(train_X)

# Evaluating rf_regressor2 model
rf_mean_sq_err = mean_squared_error(train_y, xgb_train_predicted)
r2 = r2_score(train_y, xgb_train_predicted)
print('Evaluating XGBoost model on Training Set:')
print(f"The Mean Squared Error is {rf_mean_sq_err}")
print(f"The Root Mean Squared Error is {np.sqrt(rf_mean_sq_err)}")
print(f"The R2 score is {r2}")
```

```
Evaluating XGBoost model on Training Set:
The Mean Squared Error is 9.470218502635089e-05
The Root Mean Squared Error is 0.009731504766805127
The R2 score is 0.9982881526282187
```

[132]:
```
# Model predicting values on Validation Set
xgb_val_predicted = xgboost_regressor.predict(val_X)

# Evaluating rf_regressor2 model
rf_mean_sq_err = mean_squared_error(val_y, xgb_val_predicted)
r2 = r2_score(val_y, xgb_val_predicted)
print('Evaluating XGBoost model on validation set:')
print(f"The Mean Squared Error for Validation Set is {rf_mean_sq_err}")
print(f"The Root Mean Squared Error for Validation Set is {np.sqrt(rf_mean_sq_err)}")
print(f"The R2 score for Validation Set is {r2}")
```

```
Evaluating XGBoost model on validation set:
The Mean Squared Error for Validation Set is 0.010207109422143262
The Root Mean Squared Error for Validation Set is 0.10103024013701671
The R2 score for Validation Set is -3.22697107591586
```

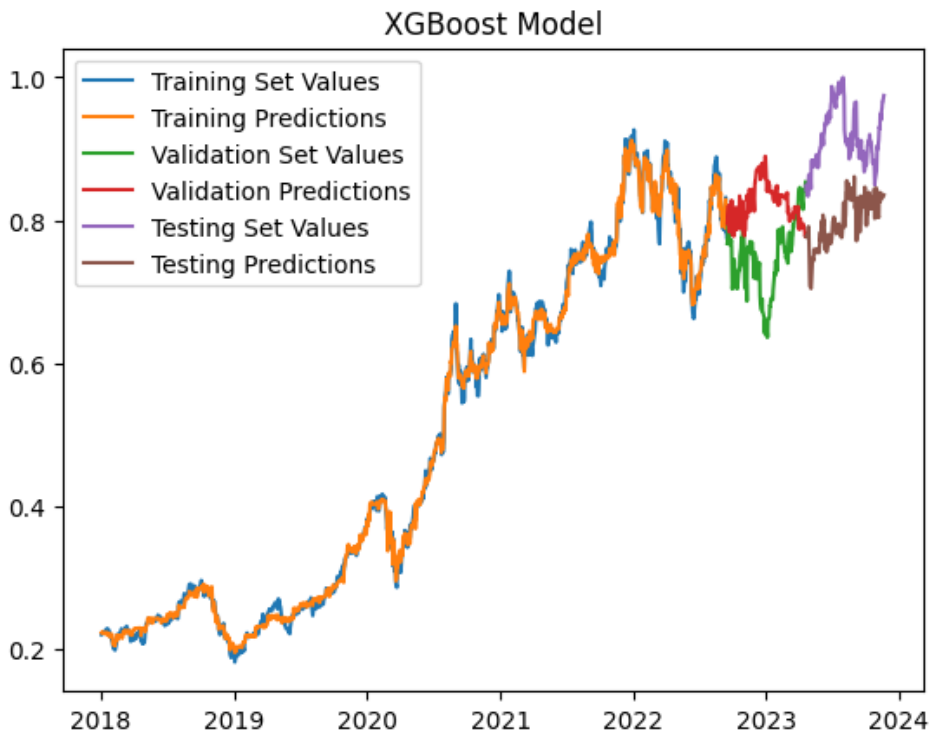


This model seems better than the random forest model, but it is still not generalizing the validation set well. Let us check how it performs on the testing set:

```
[160]: # testing it with the test set
xgb_test_predicted = xgboost_regressor.predict(test_X)
xgb_close_prices_test = pd.DataFrame(data={'real':test_y, 'pred':xgb_test_predicted})

# Evaluating rf_regressor2 model on test data
xgb_mean_sq_err = mean_squared_error(test_y, xgb_test_predicted)
r2 = r2_score(test_y, xgb_test_predicted)
print('Evaluating XGBoost model on test set:')
print(f"The Mean Squared Error is {xgb_mean_sq_err}")
print(f"The Root Mean Squared Error is {np.sqrt(xgb_mean_sq_err)}")
print(f"The R2 score is {r2}")
```

```
Evaluating XGBoost model on test set:
The Mean Squared Error is 0.01597999311720448
The Root Mean Squared Error is 0.1264119975208227
The R2 score is -8.494319620324132
```

Finally plotting all the graphs of the training, validation, and testing predictions:



Deep Learning Model – LSTM Regressor

Long-Term Short-Term Memory (LSTM) models are a type of recurrent neural network models. This type of neural network has the capability to learn better from past sequence of data. Therefore, these are a good type of models while working with time-series data.

```
[145]: # Build the LSTM model
lstm_model = Sequential()
lstm_model.add(Input((train_X.shape[1], 1))) # train_X.shape[1] returns the number of attributes in the train set
lstm_model.add(LSTM(64))
lstm_model.add(Dense(32, activation='relu'))
lstm_model.add(Dense(32, activation='relu'))
lstm_model.add(Dense(1))

# Compiling the model
lstm_model.compile(optimizer='adam', loss='mean_squared_error')
```

```
[146]: lstm_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_1 (LSTM)               (None, 64)                16896

 dense (Dense)               (None, 32)                2080

 dense_1 (Dense)             (None, 32)                1056

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 20065 (78.38 KB)
Trainable params: 20065 (78.38 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Training the LSTM model:

```
[151]: # Compiling the model
lstm_model.compile(optimizer='adam', loss='mean_squared_error')
```

```
[152]: # Train the model
lstm_model.fit(train_X, train_y, validation_data=(val_X, val_y), epochs=10, callbacks=[check_point_lstm])
```

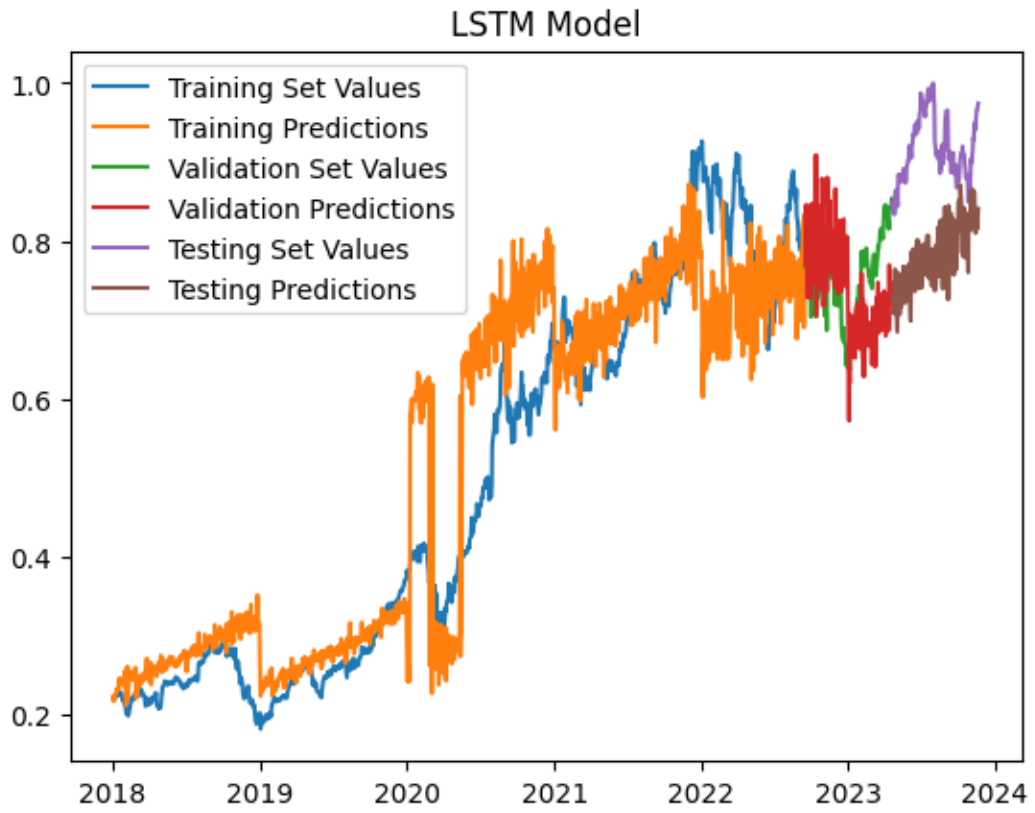
```
Epoch 1/10
18/38 [=============>.............] - ETA: 0s - loss: 0.1421 INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 2s 35ms/step - loss: 0.0888 - val_loss: 0.0508
Epoch 2/10
19/38 [==============>.............] - ETA: 0s - loss: 0.0257INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 1s 33ms/step - loss: 0.0252 - val_loss: 0.0341
Epoch 3/10
21/38 [===============>..............] - ETA: 0s - loss: 0.0198INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 1s 25ms/step - loss: 0.0186 - val_loss: 0.0208
Epoch 4/10
21/38 [===============>..............] - ETA: 0s - loss: 0.0157INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 1s 26ms/step - loss: 0.0148 - val_loss: 0.0154
Epoch 5/10
19/38 [==============>.............] - ETA: 0s - loss: 0.0131INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 1s 27ms/step - loss: 0.0121 - val_loss: 0.0109
Epoch 6/10
38/38 [==============================] - ETA: 0s - loss: 0.0114INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 1s 26ms/step - loss: 0.0114 - val_loss: 0.0105
Epoch 7/10
37/38 [============================>.] - ETA: 0s - loss: 0.0089INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 1s 33ms/step - loss: 0.0089 - val_loss: 0.0075
Epoch 8/10
38/38 [==============================] - 0s 3ms/step - loss: 0.0082 - val_loss: 0.0144
Epoch 9/10
23/38 [=================>............] - ETA: 0s - loss: 0.0114INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 1s 25ms/step - loss: 0.0102 - val_loss: 0.0061
Epoch 10/10
38/38 [==============================] - 0s 3ms/step - loss: 0.0080 - val_loss: 0.0124
[152]: <keras.src.callbacks.History at 0x31d5da580>
```

Let us calculate the same metrics for this LSTM model too:

[170]:
```
# Evaluating LSTM model on test data
lstm_mean_sq_err = mean_squared_error(train_y, lstm_train_preds)
r2 = r2_score(train_y, lstm_train_preds)
print('Evaluating the LSTM model on train set:')
print(f"The Mean Squared Error is {lstm_mean_sq_err}")
print(f"The Root Mean Squared Error is {np.sqrt(lstm_mean_sq_err)}")
print(f"The R2 score is {r2}")
print()

# Evaluating LSTM model on validation data
lstm_mean_sq_err = mean_squared_error(val_y, lstm_val_preds)
r2 = r2_score(val_y, lstm_val_preds)
print('Evaluating the LSTM model on validation set:')
print(f"The Mean Squared Error is {lstm_mean_sq_err}")
print(f"The Root Mean Squared Error is {np.sqrt(lstm_mean_sq_err)}")
print(f"The R2 score is {r2}")
print()

# Evaluating LSTM model on test data
lstm_mean_sq_err = mean_squared_error(test_y, lstm_test_preds)
r2 = r2_score(test_y, lstm_test_preds)
print('Evaluating the LSTM model on train set:')
print(f"The Mean Squared Error is {lstm_mean_sq_err}")
print(f"The Root Mean Squared Error is {np.sqrt(lstm_mean_sq_err)}")
print(f"The R2 score is {r2}")
```

```
Evaluating the LSTM model on train set:
The Mean Squared Error is 0.008012394646228717
The Root Mean Squared Error is 0.0895119804619958
The R2 score is 0.8551670511825595

Evaluating the LSTM model on validation set:
The Mean Squared Error is 0.006121489358902203
The Root Mean Squared Error is 0.07823994733447999
The R2 score is -1.5350329257245385

Evaluating the LSTM model on train set:
The Mean Squared Error is 0.020631585631008417
The Root Mean Squared Error is 0.14363699255765702
The R2 score is -11.258007047824574
```

After training the model let us plot the training and validation predictions of the model:



Evaluation of All Models

Model	Training Set			Validation Set			Testing Set		
	MSE	RMSE	R2	MSE	RMSE	R2	MSE	RMSE	R2
Random Forest Regressor #1	3.96×10^{-10}	1.99×10^{-5}	0.999	0.00396	0.063	-2.35	-	-	-
Random Forest Regressor #2	0.000467	0.0216	0.992	0.0118	0.109	-3.88	0.025	0.159	-13.95
XGBoost Regressor	9.47×10^{-5}	0.00973	0.998	0.0102	0.101	-3.23	0.016	0.126	-8.49
LSTM Regressor	0.0080	0.0895	0.855	0.00612	0.078	-1.53	0.021	0.143	-11.25

Based on the above table, we can see that the best model in terms of the evaluation metrics was the XGBoost Regressor. It has the lowest score for the mean square error (MSE) and root mean square errors (RMSE). This model has almost perfect R2 score of 0.998 which fits the training data almost perfectly, and still keeps the R2 score low (as expected) for the validation and testing data.

Based on the metrics, the second-best model is the LSTM model. This model generalizes the data well on both training and validation. However, it falls short behind the XGBoost model based on the R squared score.

Conclusion

In this supervised learning assignment, I decided to work on time-series prediction of the S&P 500 stock index with a large dataset of more than 8 million rows of stock market data ranging from 1980 to 2023. I performed data cleaning on all these rows and performed feature engineering – which included a mathematical approach to adding new features as well as the unsupervised learning approach. Before performing clustering using K-Means and GMMs, I also performed dimensionality reduction using PCA. Running a random forest in this large dataset was too long and after 20 hours I had to abandon this approach. Therefore, I focused the rest of the assignment specifically on predicting the closing price of Apple stock. Using classical machine learning techniques - I trained 2 random forest regression models, and an XGBoost regressor model. I also used a deep learning technique to train the model using LSTMs. Finally, I evaluated each of the models based on the MSE, RMSE and R squared metrics.

References

1. Korstanje, J. (2023, July 31). *How to select a model for Your time series prediction task [guide]*. neptune.ai. https://neptune.ai/blog/select-model-for-time-series-prediction-task
2. Levy, E. (2023, September 28). *What is the parquet file format? use cases & benefits*. Upsolver. https://www.upsolver.com/blog/apache-parquet-why-use
3. Korstanje, J. (2023, July 31). *How to select a model for Your time series prediction task [*guide*]*. neptune.ai. https://neptune.ai/blog/select-model-for-time-series-prediction-task
4. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sort_values.html
5. https://towardsdatascience.com/tips-on-working-with-datetime-index-in-pandas-2bcedf956d70#:~:text=By%20default%20pandas%20will%20use,your%20datetime%20data%20during%20import.
6. https://www.investopedia.com/terms/s/sp500.asp
7. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html
8. https://www.analyticsvidhya.com/blog/2020/08/types-of-categorical-data-encoding/
9. https://www.kdnuggets.com/2022/08/implementing-dbscan-python.html
10. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html
11. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
12. https://www.analyticsvidhya.com/blog/2021/05/know-the-best-evaluation-metrics-for-your-regression-model/
13. . https://keras.io/api/callbacks/model_checkpoint/