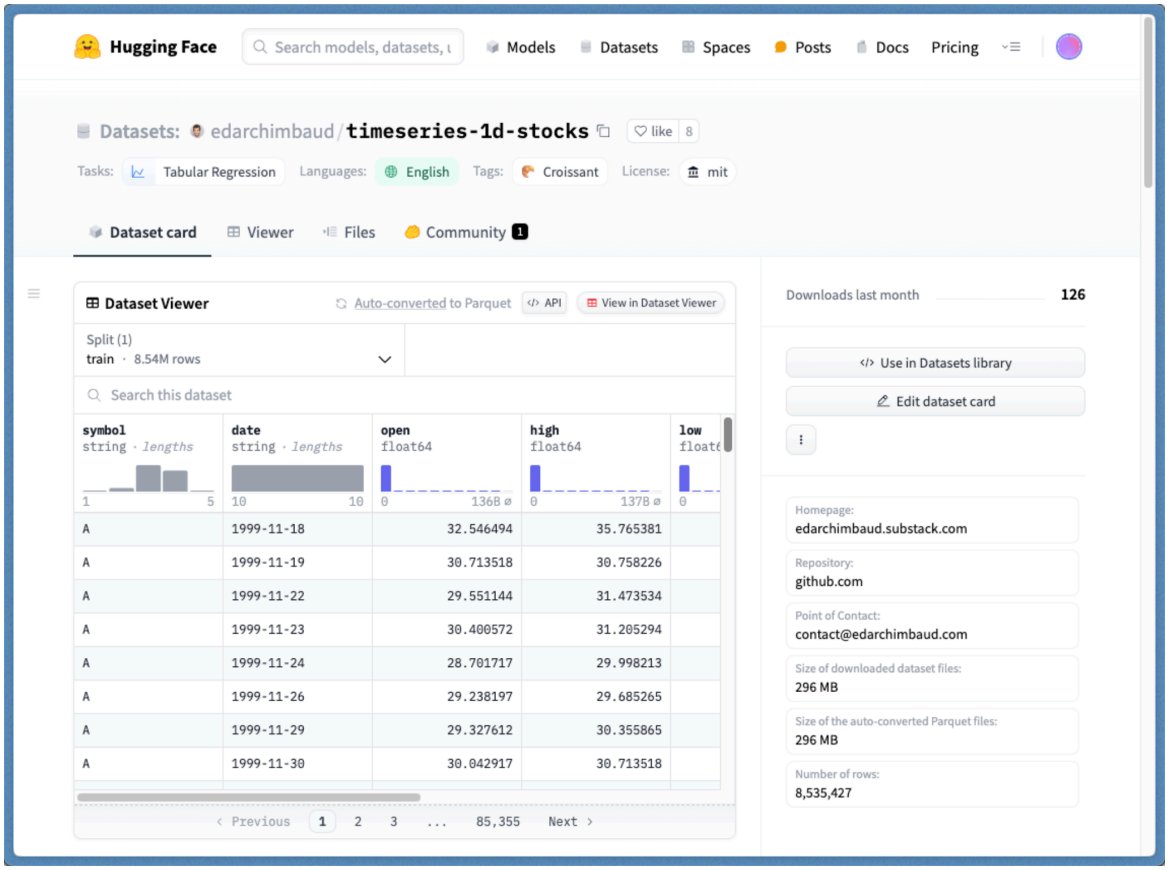
Stock Forecasting through
Machine Learning & Deep Learning

Reeaa Rana
School of Technology, Management & Engineering
SVKNM's NMIMS Indore
21 March 2024

Abstract

For this project, I utilised time-series data to forecast the closing price of Apple stock. I conducted feature engineering using mathematical techniques and unsupervised clustering methods. Additionally, I compared and assessed the performance of traditional machine learning models, including random forest regressors and XGBoost regressors, alongside a deep learning LSTM model.

Dataset



Dataset	Name	Dataset Character-istics	Attribute Character-istics	Associate Task	Number of Instances	Number of attributes
1	Time Series Forecastin g of S&P 500 stocks	Multivariate	Real	Regression/ Forecasting	85,35,427	7

I have selected a dataset from Hugging Face containing time-series data on the S&P 500 stocks. What makes this dataset intriguing is its comprehensive coverage of the S&P 500, a weighted stock market index comprising 500 prominent public companies in the United States across various sectors such as technology, healthcare, and finance. Spanning from the 1980s to 2023, the dataset includes details on each company's opening and closing prices, as well as trading volume. Analyzing this dataset allows for insights into individual company performance and facilitates forecasting, aiding in investment decision-making within the S&P 500 stock market index.

Before delving into machine learning or deep learning models, traditional statistical methods were employed for data trend analysis and prediction. Given that predicting the S&P 500 stock index involves identifying patterns and trends within the data, it lends itself well to machine learning and deep learning approaches.

To load this dataset, I utilized the huggingface_hub library. The dataset is stored in Apache Parquet (.parquet) format, which organizes data in a column-wise manner, differing from CSV files that arrange data row-wise.

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

Upon examining the dataset, it becomes apparent that it contains a substantial volume of data. Consequently, I have applied a filtering process to retain only the data pertaining to the top 10 stocks. This allows for a more focused analysis, enabling a comprehensive understanding of the various data types and trends across all features throughout the entire timeline of the top 10 companies.

```python
# 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)
```

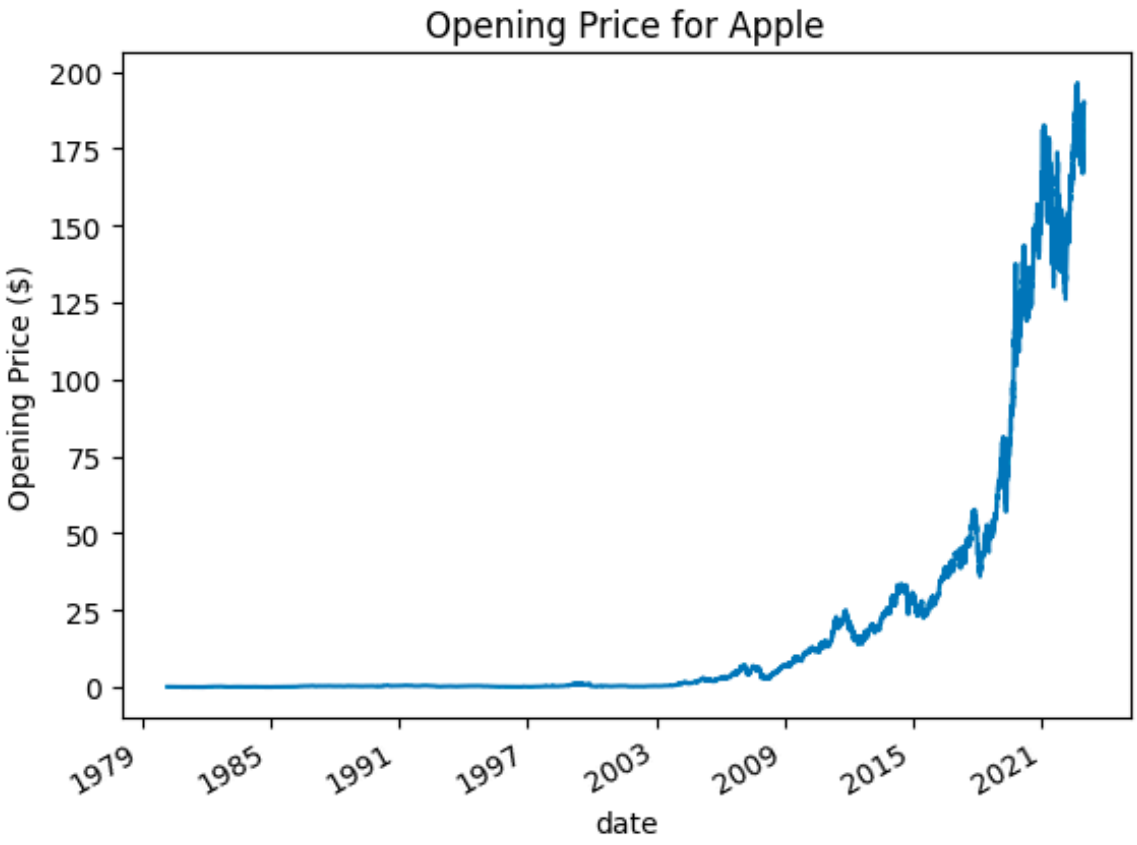
[17]

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

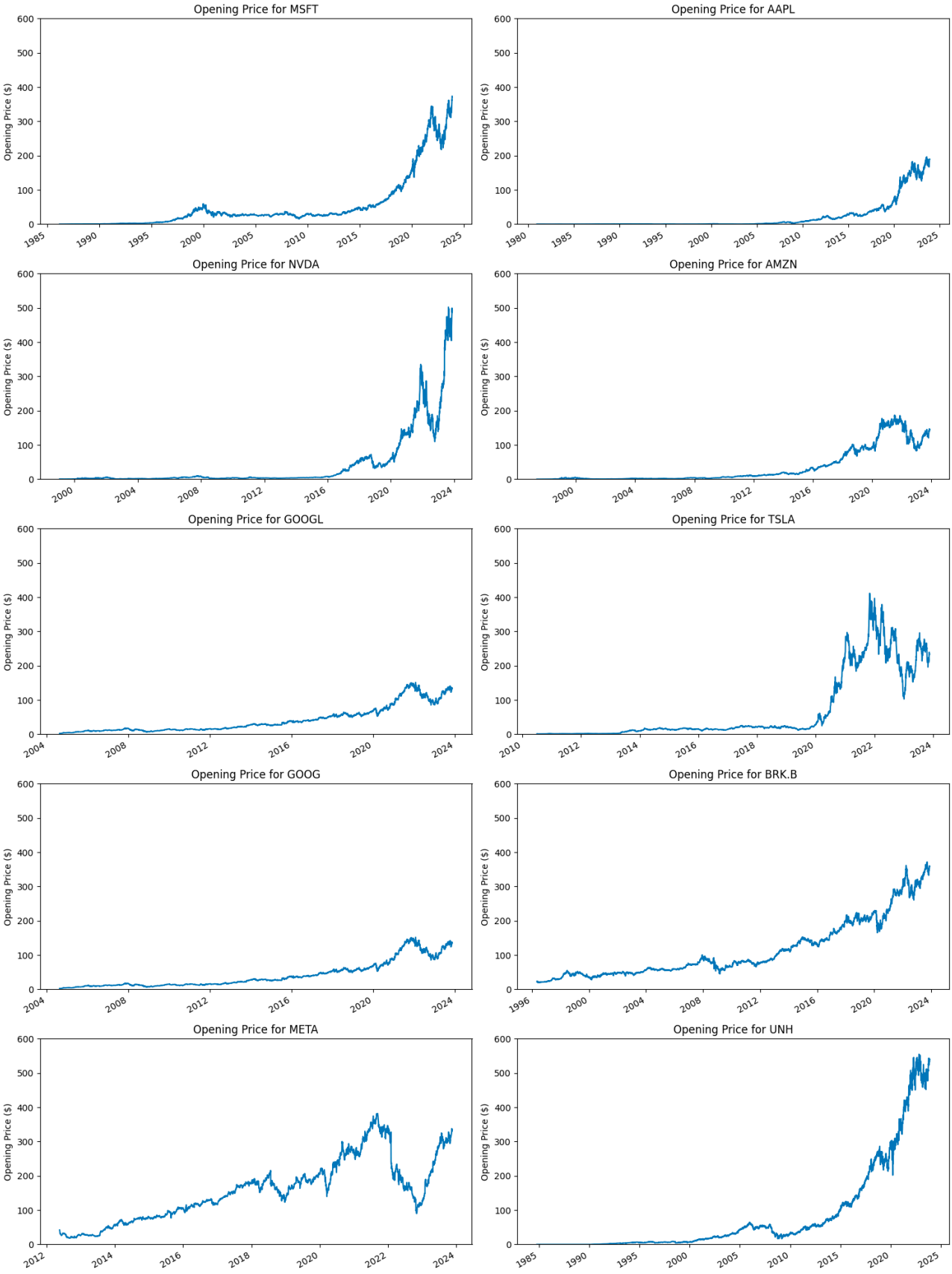
65983 rows × 7 columns

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

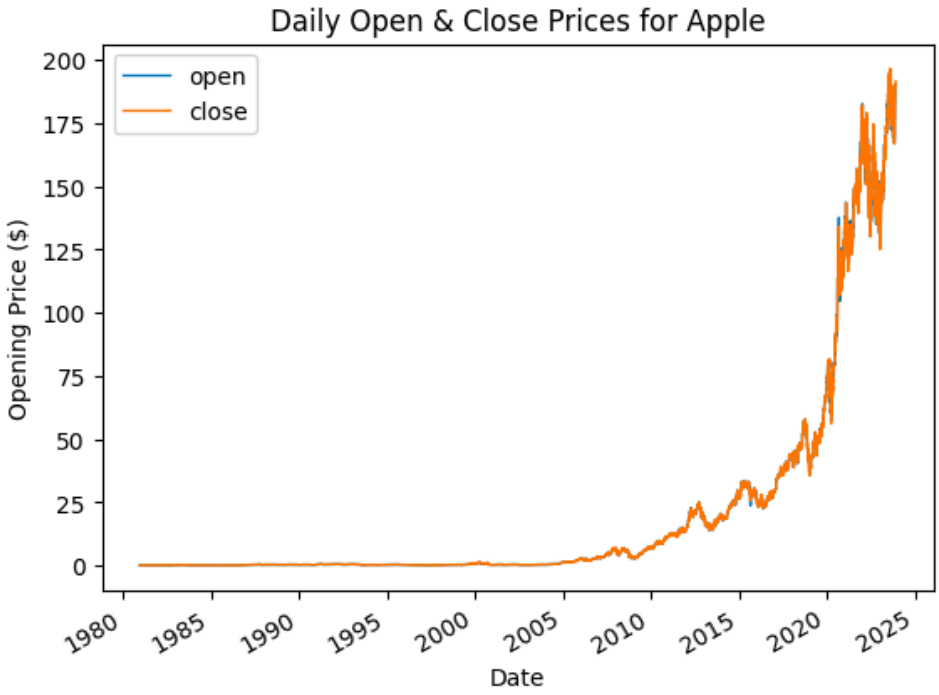
To begin, let's visualize the opening prices specifically for Apple stocks over time. As depicted below, we observe a notable increase in the open price starting from 2018 onwards.



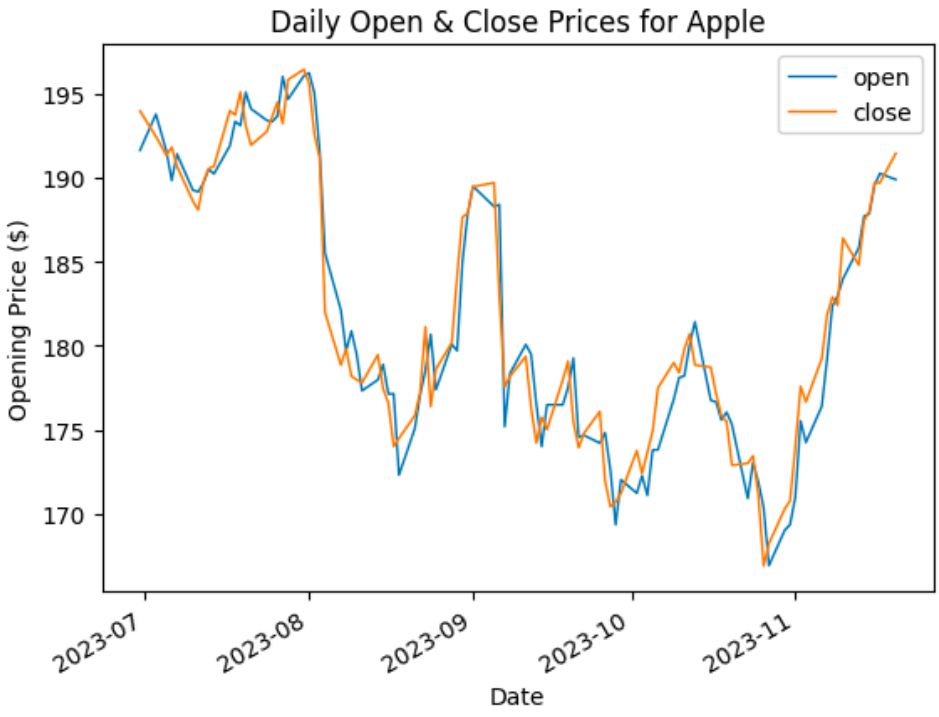
Likewise, let's examine the opening prices for the top 10 stocks in the S&P 500:



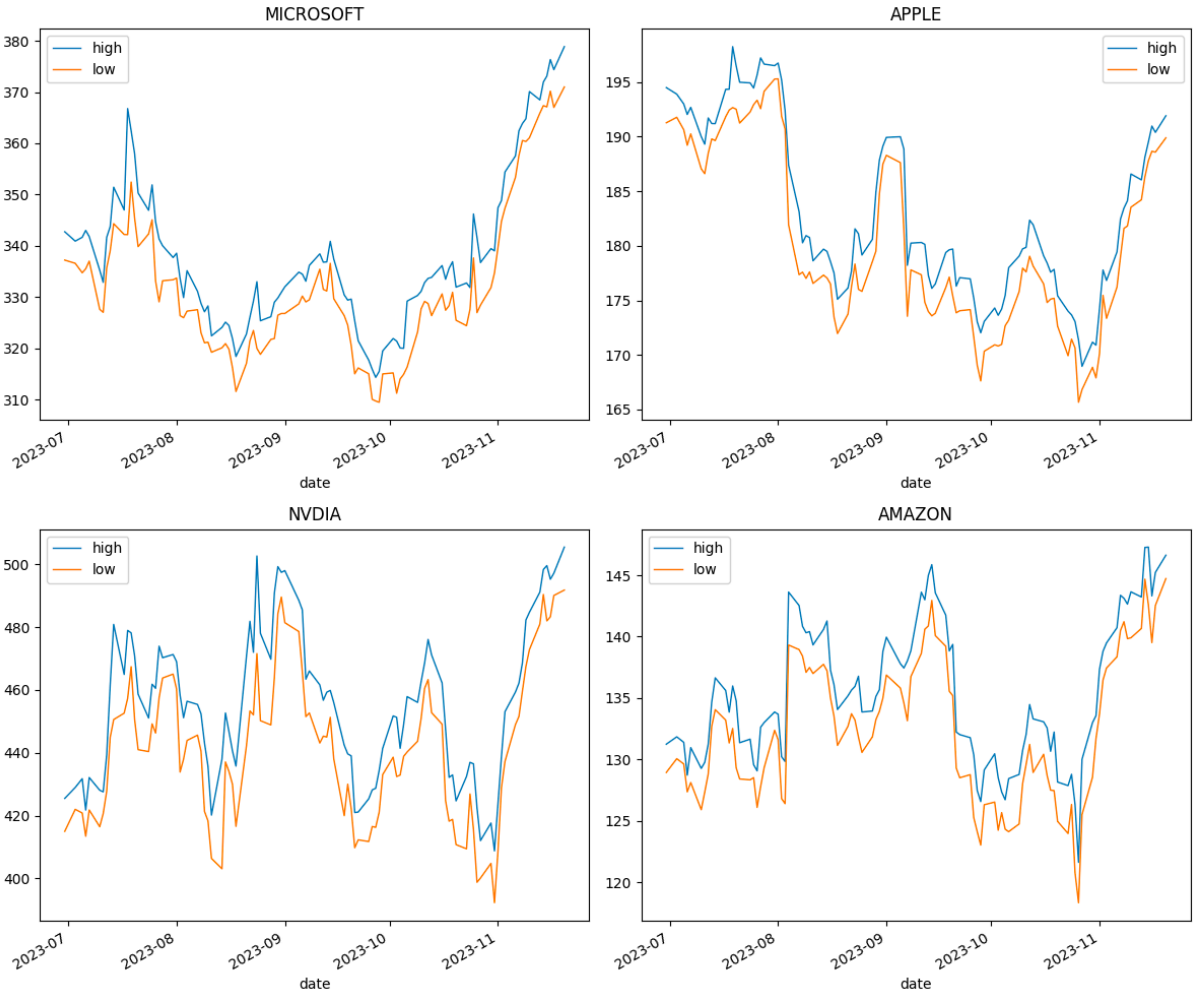
Plotting the open and close prices for Apple, we observe that the price change is not significantly different. The close price typically aligns closely with or occasionally exceeds the open price.



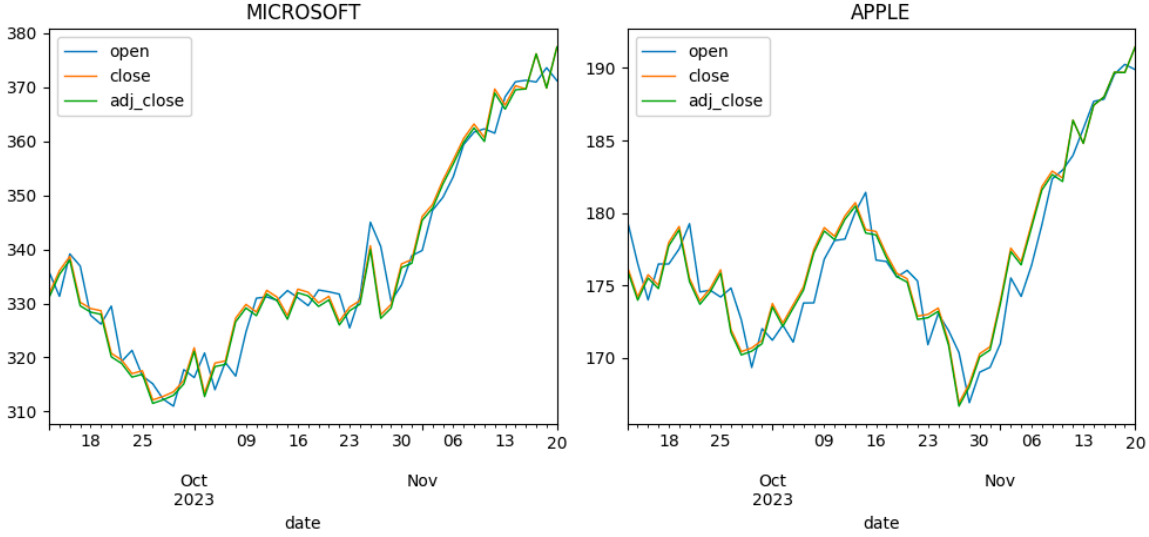
When zooming in on the same graph above, we gain a clearer understanding of the data:



Let's analyze the differences based on the high and low prices of different stocks by plotting a graph for the top 4 companies, as illustrated below. Similar to the previous graph, we observe that the daily high and low prices exhibit minimal discrepancies.



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



From the data above, it's evident that Microsoft exhibited a lower adjusted closing price (depicted by the green line) in comparison to the actual closing price (represented by the orange line).

Correlation Matrix

Upon examining the data types, it becomes apparent that all columns exhibit high correlation with each other. This observation is further supported by the graphs plotted earlier, indicating that the values of the high, low, open, close, and adj_close columns mutually influence each other.

```
[27]  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

Addressing the issue of highly correlated data is imperative, and we'll tackle it during the feature engineering phase of this project.

Defining The Problem

In this project, my objective is to forecast the closing price for the stocks. This decision is motivated by the belief that having accurate predictions of the closing price, along with other relevant parameters, enables informed investment decisions. By analyzing the closing price data over time and projecting it into the future, we aim to enhance our ability to assess the potential of investing in specific stocks.

Data Pre-Processing

Handling null Values

There are total 202 null values found in this dataset

```
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)
```

```
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)
```

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 comprising 8 million rows, we've identified 202 rows containing missing values. Despite the presence of these missing values, we can opt to employ the forward fill (fill) method for imputation. This approach entails using the last valid value of the respective columns to fill the non-null values. To ensure the integrity of the imputation process, we'll first group the data based on the 'symbol' column and then apply the fill() method within

each group. This ensures that the last valid value used for imputation

```
[38]
clean_df = dataframe.groupby('symbol')[['symbol','open', 'high', 'low', 'close', 'adj_close', 'volume']].ffill()

[39]
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)

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

corresponds to the same stock value.

Encoding Text Column

In our dataset, we have only one column labeled 'symbol', which represents the code of the company. With at least 500 unique values, each corresponding to a different company, our dataset may expand further if new companies are added to the S&P 500 index.

```
[41]
aapl_df['symbol'].value_counts()

...  symbol
     AAPL    10826
     Name: count, dtype: int64
```

Given the nature of the 'symbol' column, neither label encoders nor one-hot encoders are suitable for encoding the data. Label encoders are inappropriate because there's no inherent ordinal relationship among the symbols. Additionally, employing one-hot encoders would result in a highly complex model due to the large number of unique data values.

```
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
```

date	symbol_0	symbol_1	symbol_2	symbol_3	symbol_4	symbol_5	symbol_6	symbol_7	symbol_8	symbol_9	symbol_10
1999-11-18	0	0	0	0	0	0	0	0	0	0	1
1999-11-19	0	0	0	0	0	0	0	0	0	0	1
1999-11-22	0	0	0	0	0	0	0	0	0	0	1
1999-11-23	0	0	0	0	0	0	0	0	0	0	1
1999-11-24	0	0	0	0	0	0	0	0	0	0	1
...	...	...	...	...	...	...	...	...	...	...	...
2023-11-14	1	0	1	1	1	0	1	0	0	1	0
2023-11-15	1	0	1	1	1	0	1	0	0	1	0
2023-11-16	1	0	1	1	1	0	1	0	0	1	0
2023-11-17	1	0	1	1	1	0	1	0	0	1	0
2023-11-20	1	0	1	1	1	0	1	0	0	1	0

8535427 rows × 11 columns

Binary Encoding for the Symbol Attribute

After dropping the old 'Symbol' column and adding the dataframe containing the encoded values, I observed that the encoded values were stored as int64 datatype. Given that we only have 0 or 1 as values, utilizing such a large datatype is unnecessary. Therefore, I adjusted the datatype to int8, resulting in significant memory savings of over 500 MB.

```
[45]  clean_df.info()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 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
```

```
[48]  clean_df.info()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 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   int8
 8   symbol_1   int8
 9   symbol_2   int8
 10  symbol_3   int8
 11  symbol_4   int8
 12  symbol_5   int8
 13  symbol_6   int8
 14  symbol_7   int8
 15  symbol_8   int8
 16  symbol_9   int8
 17  symbol_10  int8
dtypes: float64(6), int8(11), object(1)
memory usage: 610.5+ 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

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

2. Daily Price Percentage Change

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

The moving average is a key indicator in market analysis used to discern trends. It involves calculating the average of a specified window of data. In my case, I will compute moving averages for 7 days, 30 days, and 60 days. Here's how it works:

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

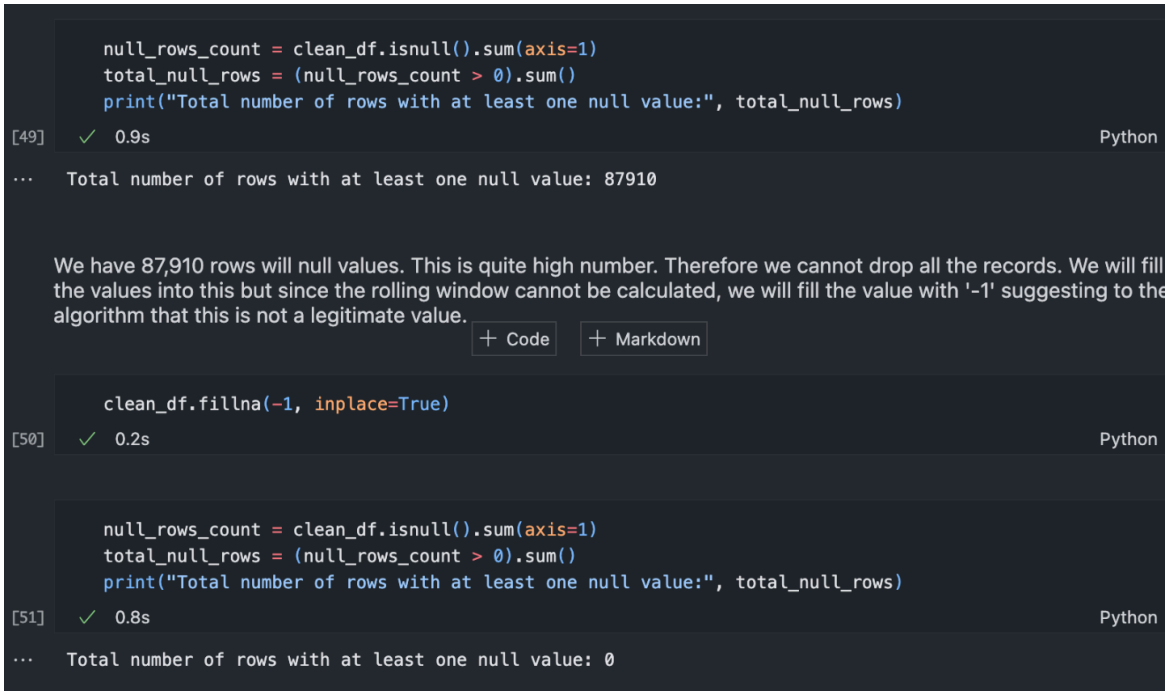
```
new_df.head(10)
```

[59] ✓ 0.0s Python

symbol	date	Moving Avg 7 Days	Moving Avg 30 Days	Moving Avg 60 Days
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

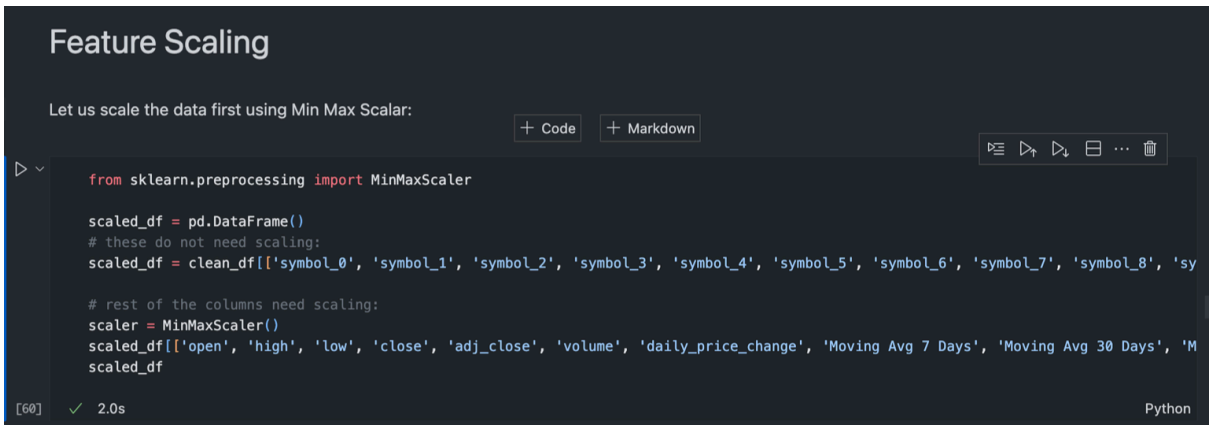
We have computed the rolling average, but it's evident that there are NaN values present in the resulting columns. Specifically, the 'Moving Avg 7 Days' column has NaN values for the first 7 records, while the last two

columns have 30 and 60 NaNs, respectively. Additionally, it's worth noting that the first 59 rows for each company will contain NaN values in the 'Moving Avg 60 Days' column. To address this, let's check the total number of rows containing NaN values.



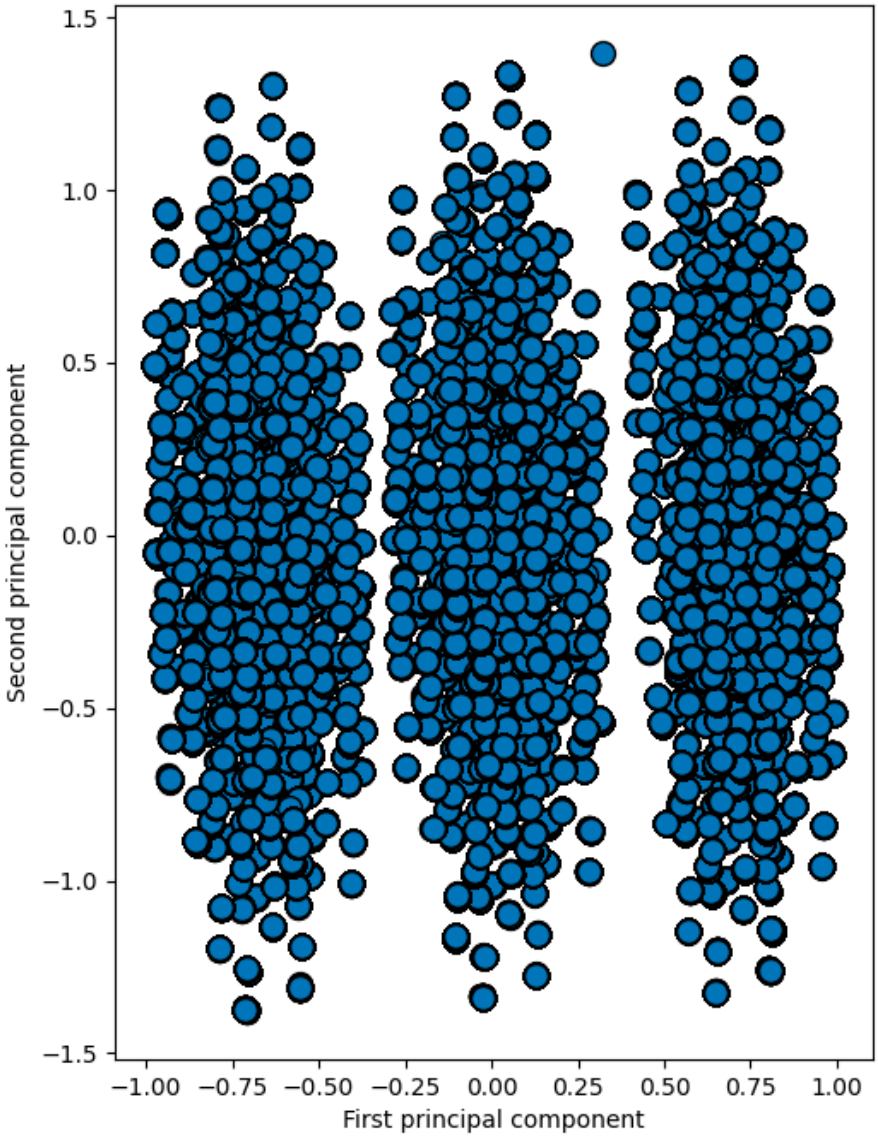
Unsupervised Learning Approach

Feature Scaling using Min Max Scalar



Performing Dimensionality Reduction using PCA

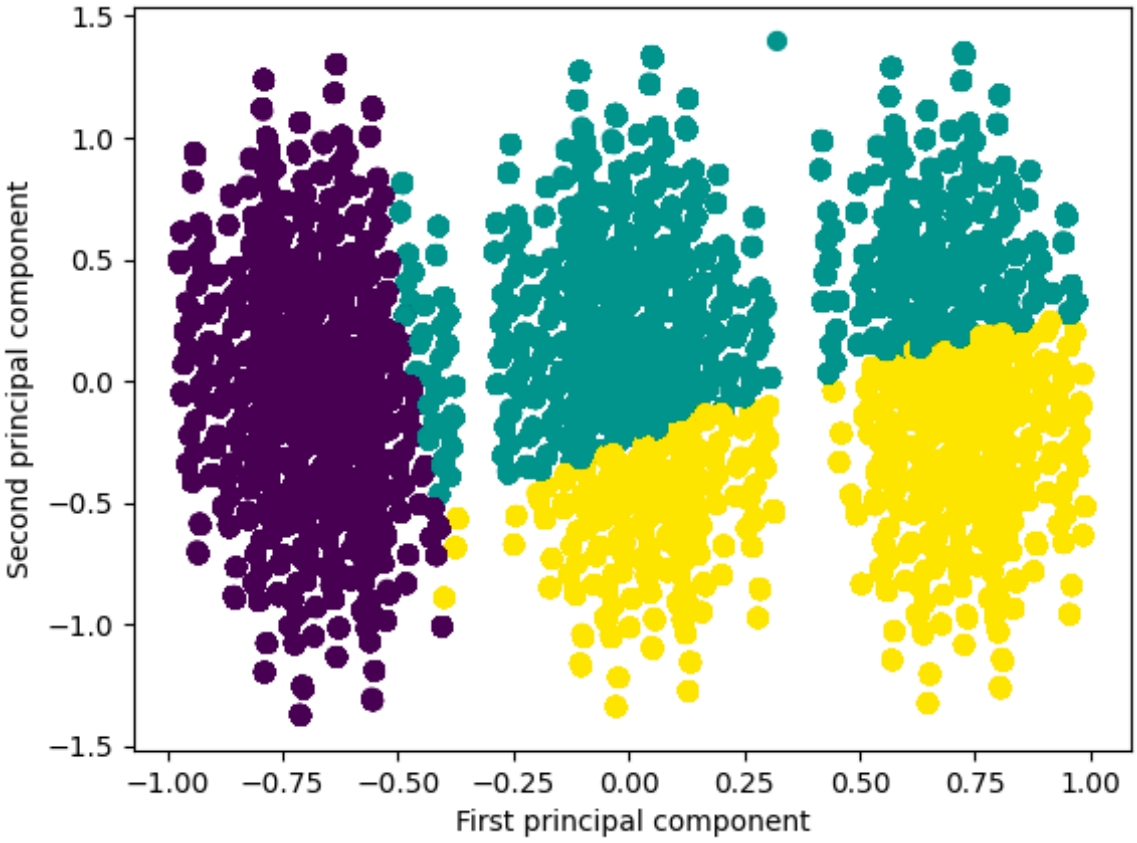
Prior to running the clustering algorithm, I conducted dimensionality reduction and visualized the data against the first and second principal components.



Based on the image provided, it appears that the data forms three distinct clusters.

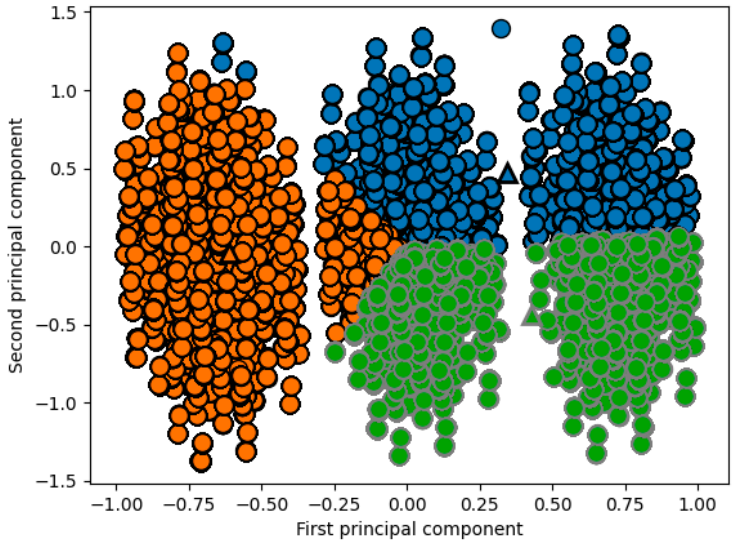
Using Gaussian Mixture Model (GMMs) for Clustering

I opted to utilize the Gaussian Mixture Model (GMM) because unlike K-Means, it considers the variance or standard deviation in the data. However, despite experimenting with various hyperparameters for the initial cluster centers, the GMM model did not correctly identify the three clusters as anticipated. The resulting graph is depicted below.

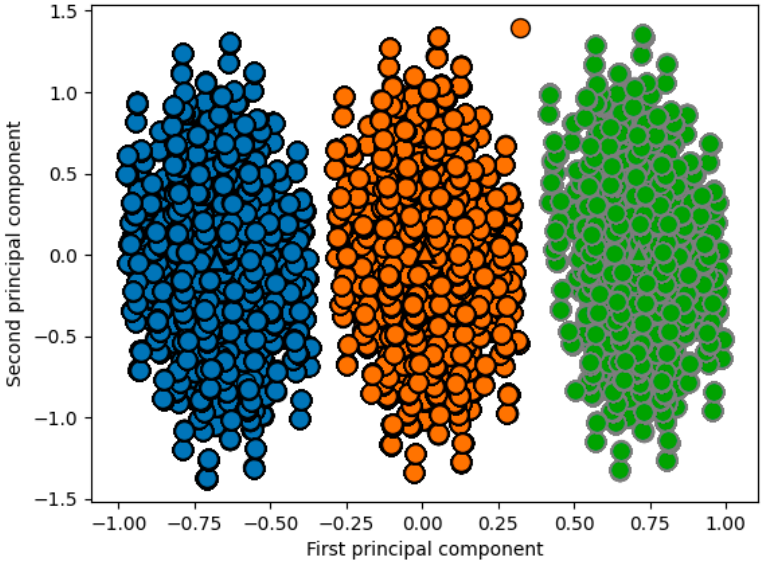


K-Means Clustering

I attempted to utilize K-Means clustering as well, but it also failed to accurately identify the visible clusters. It seems to be converging to local minima and terminating prematurely.



In our scenario, it's evident that the cluster centers should approximate around (-0.75, 0), (0.0, 0.0), and (0.75, 0). Consequently, I re-ran the K-Means algorithm with these specified cluster center values, and it successfully converged to the centers of the three clusters, as illustrated below.

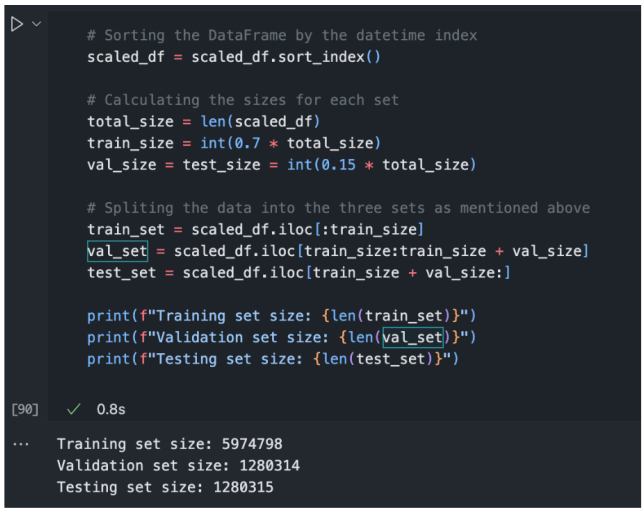


Splitting the Data

In this project, three sets were created:

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)

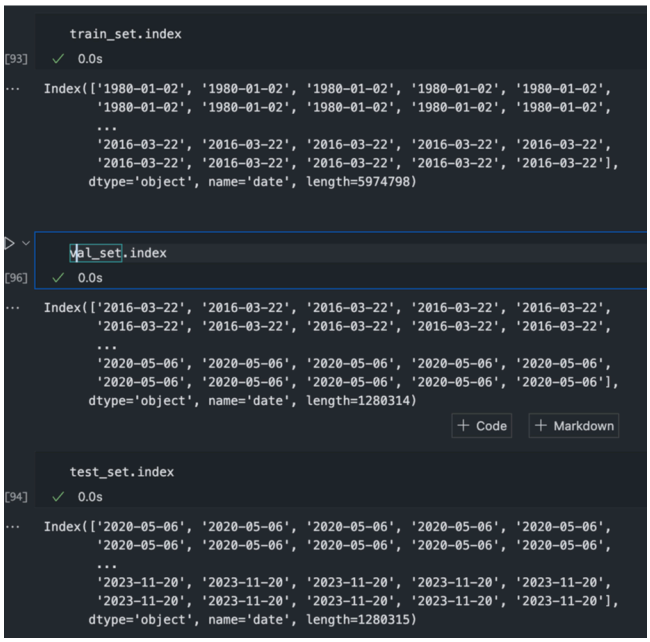
I have made the decision to allocate 15% of the dataset for the validation set. Each model will be evaluated based on its performance on this validation set. Subsequently, after determining the best-performing model, I will conduct testing on the remaining test set to assess its performance.



```
# 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 indeed an ensemble learning method that utilizes the Bootstrap Aggregation (Bagging) technique.

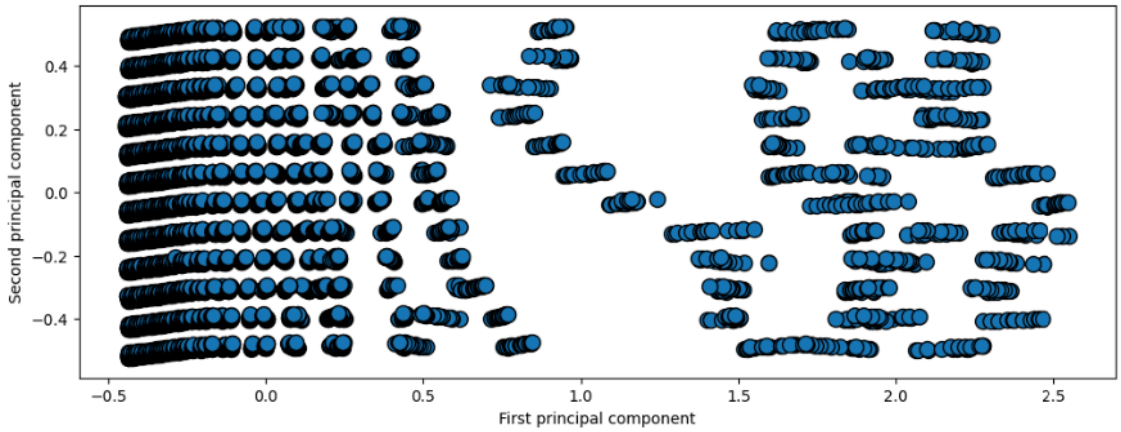
Classical Machine Learning Model

Random Forest Regressor

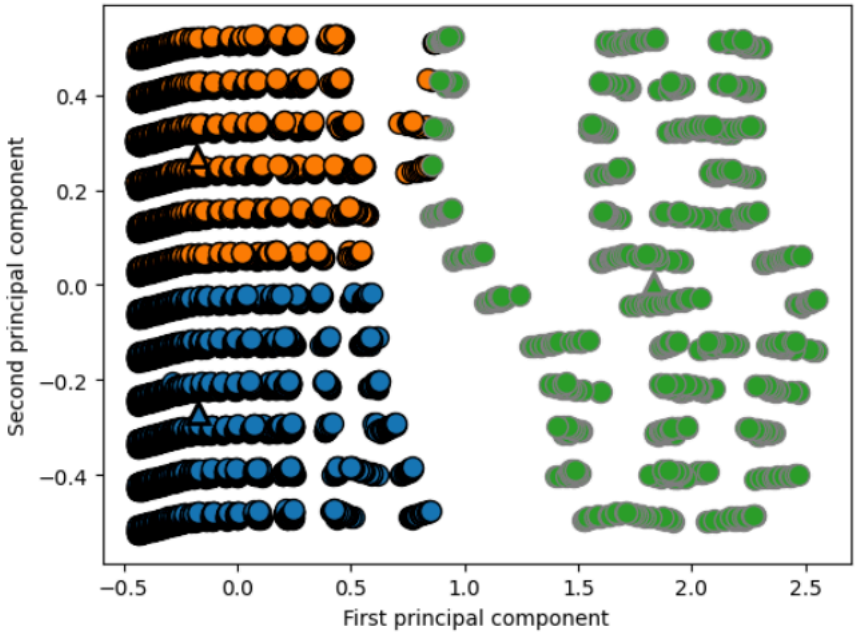
```
# from sklearn.ensemble import RandomForestRegressor

# rf_regressor = RandomForestRegressor(n_estimators=10, random_state=23, oob_score=True)
# rf_regressor.fit(train_X, train_y)
```

After more than 2800 minutes of training the model (equivalent to 45 hours), I have reached the decision to discontinue training the model on the entire dataset due to time constraints. Consequently, I will attempt to train the models on a subset of the data, focusing solely on one company, Apple.

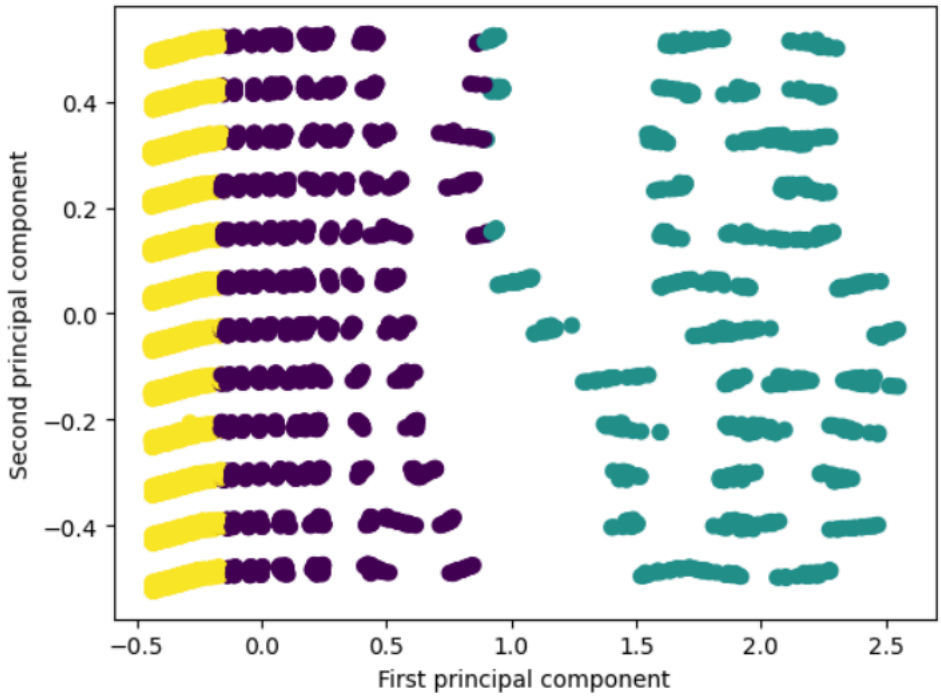
Dimensionality Reduction Again using PCA (on Apple's PCA Data Points



K-Means Custering Again on (Apple's PCA Data points)



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



Given the superior performance of the Gaussian Mixture Model (GMM) compared to the K-Means model, let's proceed by incorporating the Cluster ID from the GMM model as a new feature for our supervised learning model.

Following this update, we observe a reduction in the dataset size to 34,219 records. Additionally, we have included timestamps ranging from January 2010 to November 2023, encompassing the entirety of available data.

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's proceed by splitting the data into train, validation, and test sets again, with the following proportions: 7,536 rows for the train set, 1,615 rows for the validation set, and 1,616 rows for the test set.

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 removed columns with high correlation between each other, including:

1. Open
2. High
3. Low
4. Adj_Close
5. MovingAvg7Days
6. MovingAvg30Days

After this removal, the correlation matrix appears as follows. While there still exists high correlation between some columns, we'll proceed with this configuration for now.

Re-Running 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's contrast the actual values with the forecasted 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

Based on the image provided, it's evident that the forecasted values closely align with the actual data.

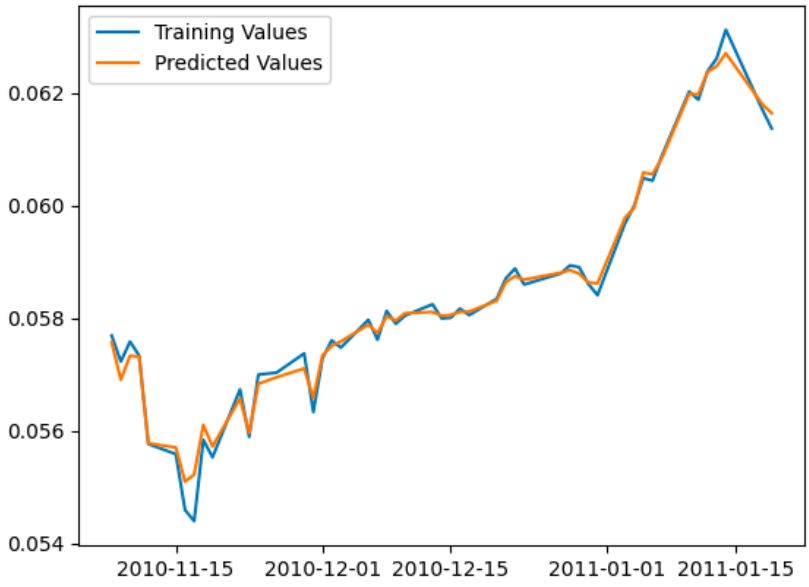
Metrics to Evaluate the Random Forest Model

To gauge the performance of this model, I've employed metrics such as mean squared error, root mean squared error, as well as r-squared and adjusted r-squared values. Assessing the predictions from training:

```
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 9.337708743226205e-08
The Root Mean Squared Error for Training Set is 0.0003055766473935174
The R2 score for Training Set is 0.9992413956473362
```

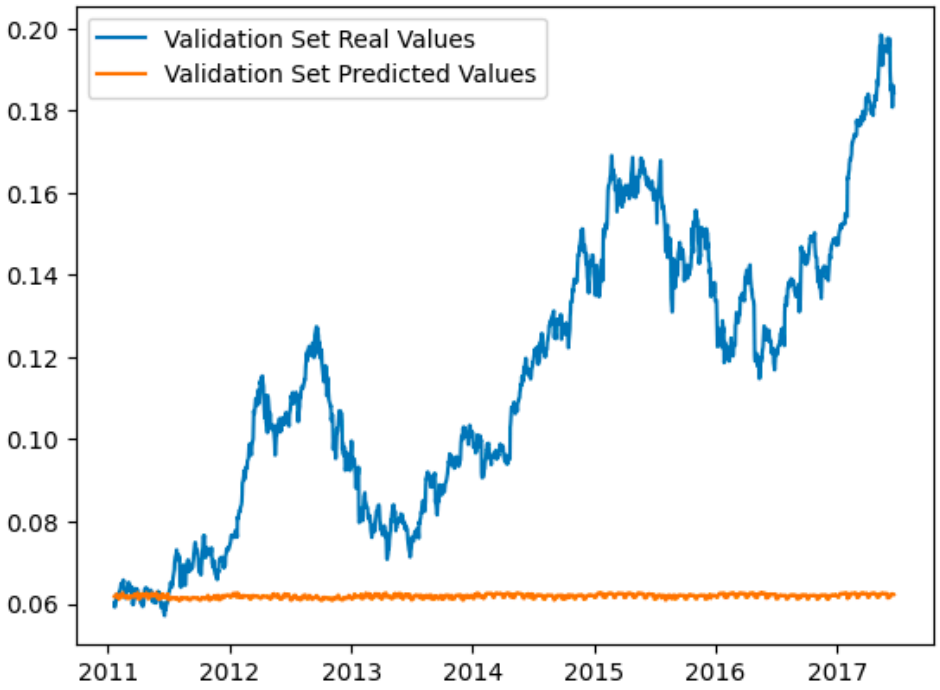
The training set exhibits a remarkably low mean squared error of 3.96 × 10^(-4), alongside a correspondingly low root mean squared error of 1.99 × 10^(-2). Additionally, the R-squared score stands at 0.999, indicating that the training model can elucidate 99% of the variance in the data. However, it's essential to note that an R-squared score nearing 1 suggests potential overfitting on the data. Consequently, there's a considerable likelihood that the model has overfit. Let's visualize the training and predicted values on a graph:



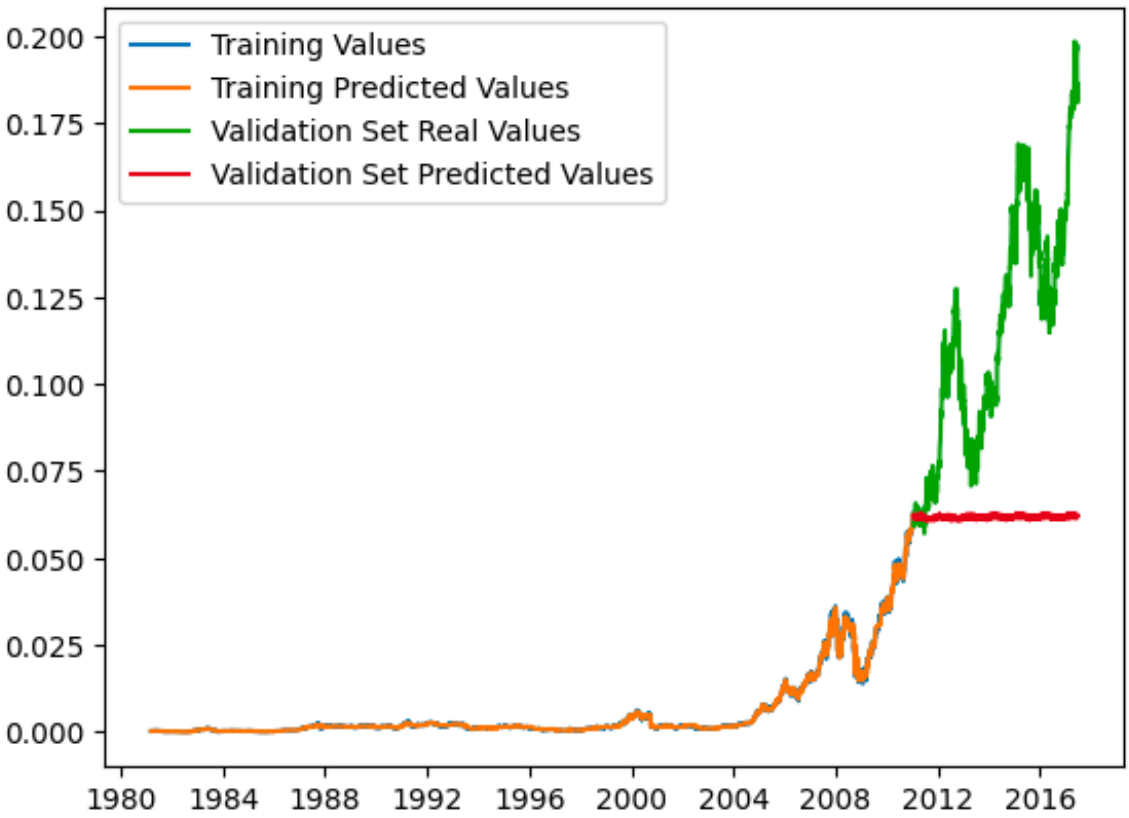
Evaluating validation set predictions:

```
# 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 0.0040067895924012584
The Root Mean Squared Error for Validation Set is 0.06329920688603656
The R2 score for Validation Set is -2.386277635353037
[Parallel(n_jobs=1)]: Done  49 tasks      | elapsed:    0.0s
```



Initially, the model performed well on the training set; however, its performance on the prediction set was notably poor. Let's visualize both the training and validation line plots on a single graph:



I have a hunch that the model's inability to generalize well to the validation dataset stems from a significant difference between the characteristics of the training and validation data. Specifically, the closing price data from 1980 to around 2011 demonstrates a relatively flat trend, punctuated by occasional sharp increases. However, starting from 2012, we observe a distinct upward trajectory in the closing price of Apple stock, as depicted by the green line.

Random Forest Regression Model 2

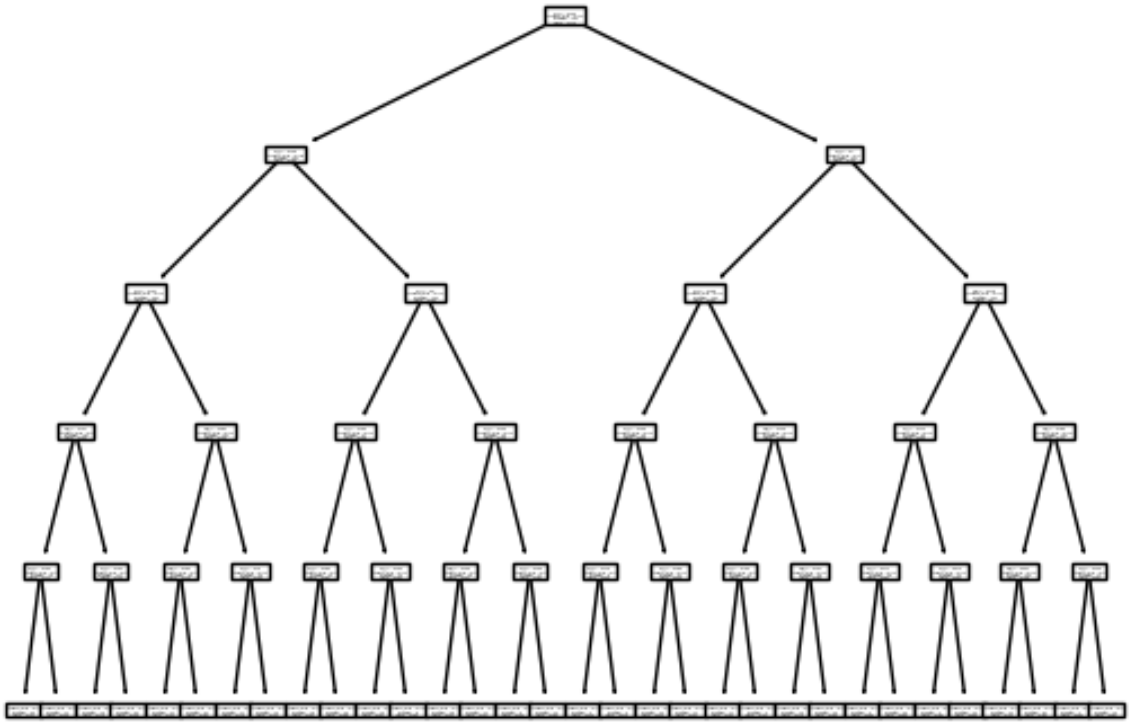
To rectify the shortcomings of the previous random forest model, we'll revisit the process by redividing our data into training, validation, and test sets. However, this time, we'll exclusively utilise data from 2018 onwards.

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

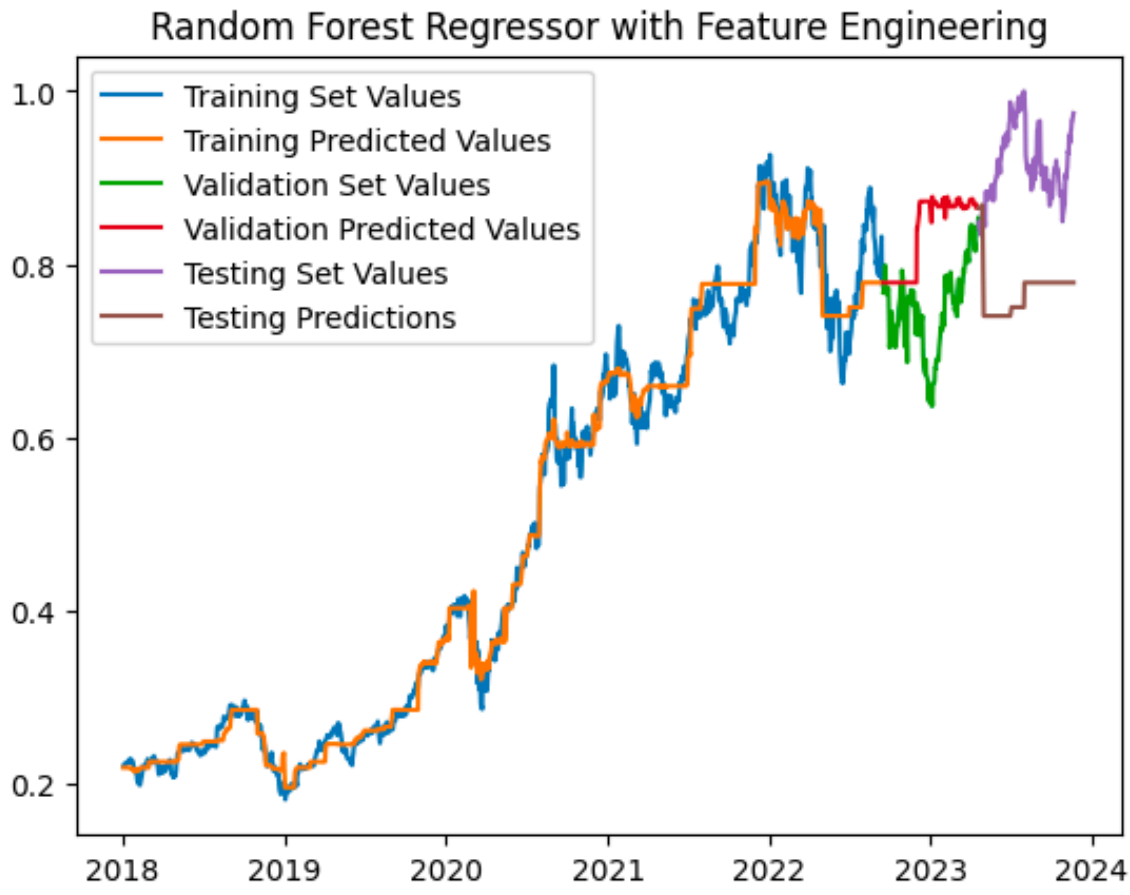
date	daily_price_change_percent	close	volume	daily_price_change	Moving Avg 60 Days	year	month	day	Cluster_ID
2018-01-02	0.012191	0.219021	0.013774	0.441125	0.225159	0.880952	0.000000	0.033333	0
2018-01-03	-0.001742	0.218983	0.015909	0.406322	0.225538	0.880952	0.000000	0.066667	0
2018-01-04	0.002832	0.220001	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



```
rf2_close_prices_test = pd.DataFrame(data={'real': test_y, 'pred': rf2_test_predicted})

# plotting training & training predictions
plt.plot(rf2_close_prices.index, rf2_close_prices['real'])
plt.plot(rf2_close_prices.index, rf2_close_prices['pred'])

# plotting validation & validation predictions
plt.plot(rf2_close_prices_val.index, rf2_close_prices_val['real'])
plt.plot(rf2_close_prices_val.index, rf2_close_prices_val['pred'])

# plotting testing & testing predictions
plt.plot(rf2_close_prices_test.index, rf2_close_prices_test['real'])
plt.plot(rf2_close_prices_test.index, rf2_close_prices_test['pred'])

plt.title("Random Forest Regressor with Feature Engineering")
plt.legend(['Training Set Values', 'Training Predicted Values', 'Validation Set Values', 'Validation Predicted Values', 'Testing Set Values', 'Testing Predictions'])
```



While this model shows improvement over the previous one, it still exhibits signs of overfitting on the training data, resulting in inaccurate predictions in both the validation and test sets.

Deep Learning Model - LSTM Regressor

LSTM (Long Short-Term Memory) models belong to the category of recurrent neural networks (RNNs). Unlike traditional RNNs, LSTMs possess the ability to effectively learn from sequences of data over extended periods. As a result, they are particularly well-suited for analyzing and making predictions with time-series data.

```
# 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))
```

```
lstm_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (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:

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

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

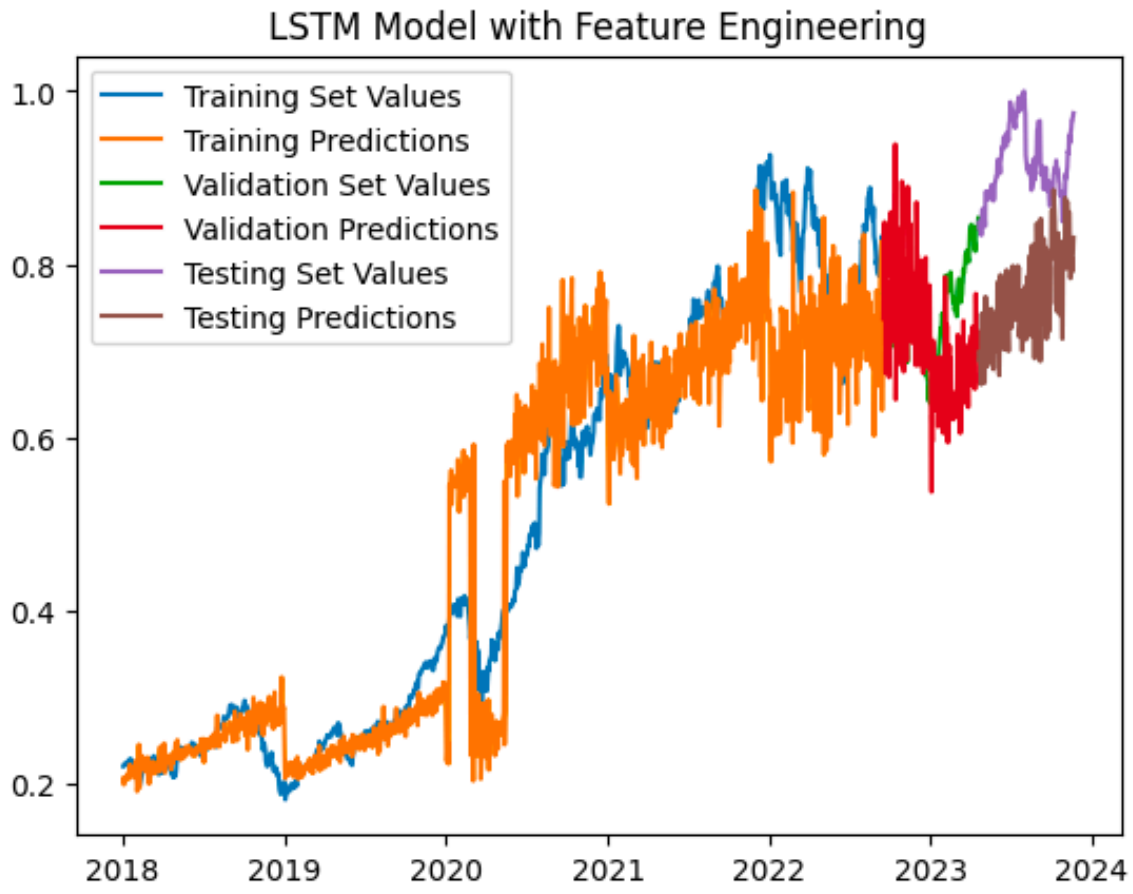
```
Epoch 1/10
18/38 [=============>............] - ETA: 0s - loss: 0.2219 INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 2s 40ms/step - loss: 0.1358 - val_loss: 0.0236
Epoch 2/10
38/38 [==============================] - 0s 3ms/step - loss: 0.0286 - val_loss: 0.0266
Epoch 3/10
38/38 [==============================] - 0s 3ms/step - loss: 0.0232 - val_loss: 0.0263
Epoch 4/10
38/38 [==============================] - 0s 3ms/step - loss: 0.0185 - val_loss: 0.0247
Epoch 5/10
21/38 [===============>..............] - ETA: 0s - loss: 0.0165INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 1s 28ms/step - loss: 0.0157 - val_loss: 0.0170
Epoch 6/10
21/38 [===============>..............] - ETA: 0s - loss: 0.0147INFO:tensorflow:Assets written to: lstm_model/assets
INFO:tensorflow:Assets written to: lstm_model/assets
38/38 [==============================] - 1s 28ms/step - loss: 0.0141 - val_loss: 0.0144
```

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

```
    # 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 0.006920894654378087
The Root Mean Squared Error is 0.08319191459738191
The R2 score is 0.8748971280739102

Evaluating the LSTM model on validation set:
The Mean Squared Error is 0.008140661463144063
The Root Mean Squared Error is 0.09022561422979654
The R2 score is -2.371213055567249

Evaluating the LSTM model on train set:
The Mean Squared Error is 0.026763483874246995
The Root Mean Squared Error is 0.16359548855102024
The R2 score is -14.901200219036385
```

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

According to the provided table, the XGBoost Regressor emerges as the top-performing model, exhibiting the lowest scores for both mean squared error (MSE) and root mean squared error (RMSE). With an impressive R-squared score of 0.998, it demonstrates nearly perfect fit to the training data while maintaining low R-squared scores, as expected, for the validation and testing datasets.

Following closely is the LSTM model, which demonstrates good generalization across both training and validation datasets. However, it lags behind the XGBoost model in terms of R-squared score, positioning it as the second-best model based on the metrics evaluated.

Conclusion

In this supervised learning project, I tackled the task of time-series prediction for the S&P 500 stock index using a substantial dataset comprising over 8 million rows of stock market data spanning from 1980 to 2023. My approach involved meticulous data cleaning across the entire dataset, followed by comprehensive feature engineering. This included both mathematical methods for generating new features and leveraging unsupervised learning techniques.

Before delving into clustering using K-Means and Gaussian Mixture Models (GMMs), I conducted dimensionality reduction using Principal Component Analysis (PCA). However, due to the computational demands of running a random forest on such a large dataset, which exceeded 20 hours without completion, I opted to pivot the focus of the assignment exclusively to predicting the closing price of Apple stock.

For this purpose, I employed classical machine learning techniques, training two random forest regression models alongside an XGBoost regressor model. Additionally, I explored deep learning methodologies by training a Long Short-Term Memory (LSTM) model.

Subsequently, I assessed the performance of each model based on metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared.

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/