AY23/24 SEM 2

Forecasting Fortunes

Stocks Price Prediction

PRESENTED BY

Group 07

Table of Content

1. Introduction

1.1. Motivation

Investing in stocks offers the potential to grow savings, shield funds from inflation and taxes, and optimize investment returns. As such, knowing when to buy which stocks is important.

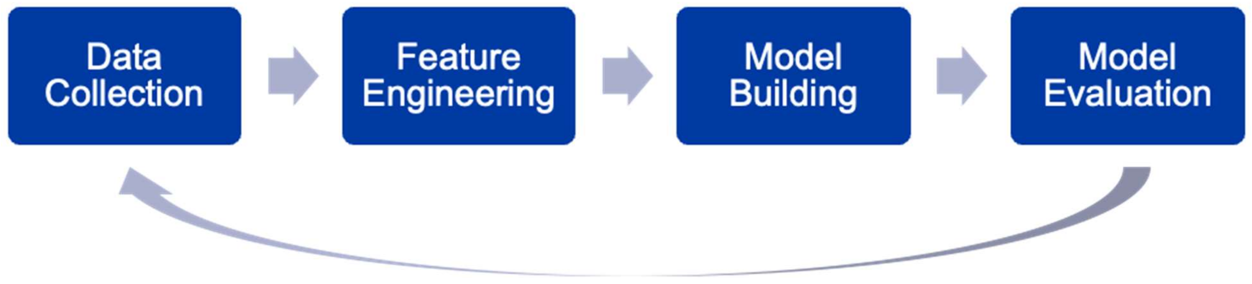
Our project endeavors to help stock investors make informed decisions by developing a predictive model that leverages current data to predict future movements in stock prices.

1.2. Description

In this project, our primary focus is on the GOOGL stock.

Utilizing historical data of the stock, we aim to predict the stock price for the subsequent day.

This process encompasses data collection, feature engineering, model construction, and model evaluation. Naturally, this process is iterative and non-linear. It often involves revisiting earlier steps based on the results obtained in later stages.



2. Data Collection

Refer to the `Data_Collection.ipynb` file.

- Supabase Set Up
- Supabase Helper Functions
- Scraping Data from Yahoo Finance (Table 1)

Refer to the `Data_Preprocessing.ipynb` file.

- Preprocessing (Table 2 and Table 3)

2.1. Data Source

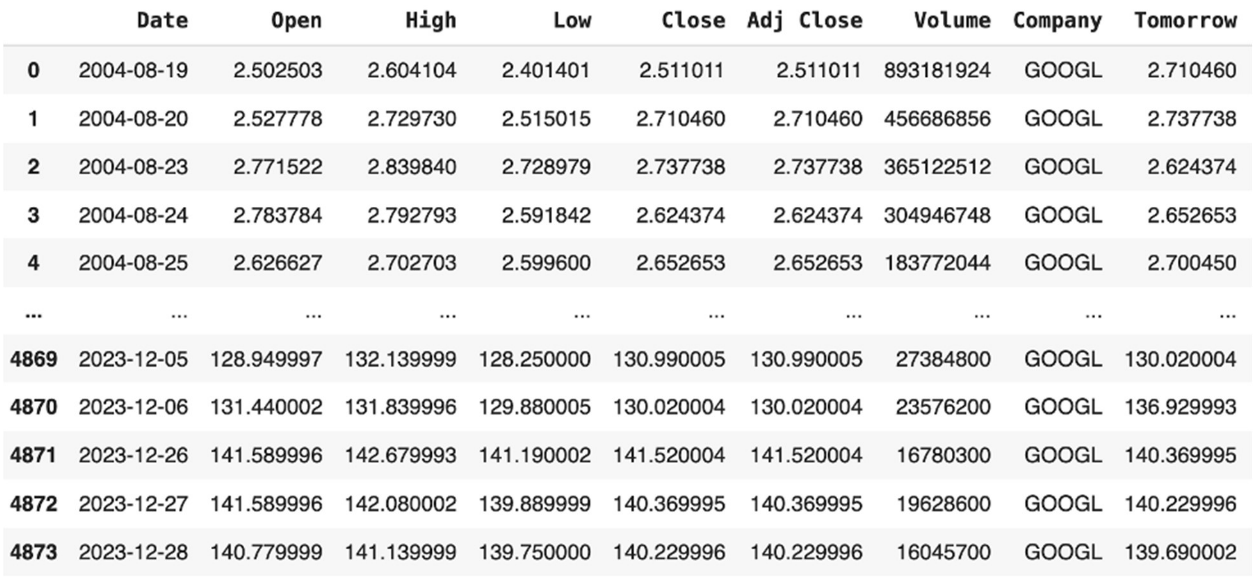
We developed a custom scraper to collect historical stock data using finance.yahoo.com API.

The scraper scrapes data starting from early 2000 to end 2023.

2.2. Table 1

This table contains raw data scraped from the API.

The most important columns are Date, Adj Close, and Tomorrow.



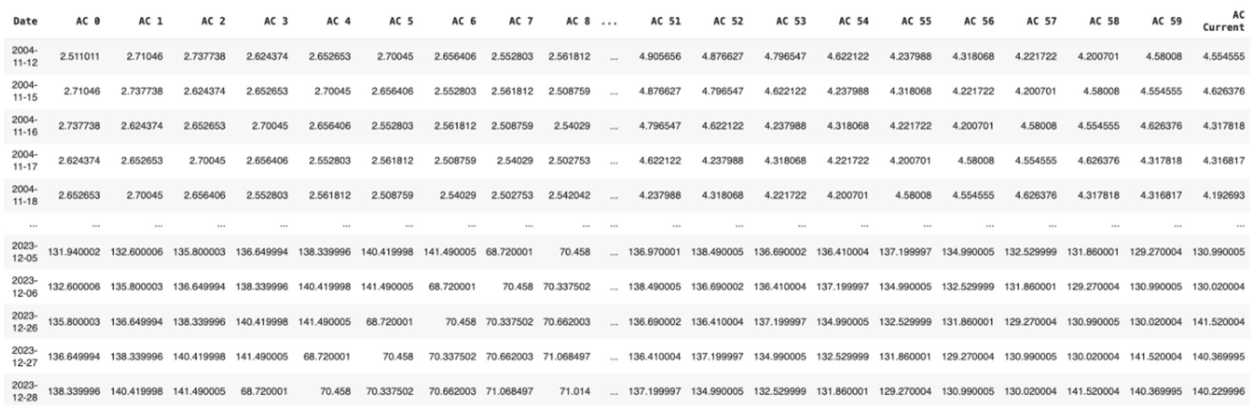
	Date	Open	High	Low	Close	Adj Close	Volume	Company	Tomorrow
0	2004-08-19	2.502503	2.604104	2.401401	2.511011	2.511011	893181924	GOOGL	2.710460
1	2004-08-20	2.527778	2.729730	2.515015	2.710460	2.710460	456686856	GOOGL	2.737738
2	2004-08-23	2.771522	2.839840	2.728979	2.737738	2.737738	365122512	GOOGL	2.624374
3	2004-08-24	2.783784	2.792793	2.591842	2.624374	2.624374	304946748	GOOGL	2.652653
4	2004-08-25	2.626627	2.702703	2.599600	2.652653	2.652653	183772044	GOOGL	2.700450
...	...	...	...	...	...	...	...	...	...
4869	2023-12-05	128.949997	132.139999	128.250000	130.990005	130.990005	27384800	GOOGL	130.020004
4870	2023-12-06	131.440002	131.839996	129.880005	130.020004	130.020004	23576200	GOOGL	136.929993
4871	2023-12-26	141.589996	142.679993	141.190002	141.520004	141.520004	16780300	GOOGL	140.369995
4872	2023-12-27	141.589996	142.080002	139.889999	140.369995	140.369995	19628600	GOOGL	140.229996
4873	2023-12-28	140.779999	141.139999	139.750000	140.229996	140.229996	16045700	GOOGL	139.690002

Feature	**Description**
Date	Specific date on which trading activity occurred. Weekends are skipped as the stock market is closed. First recorded date is on 2004-08-19 as GOOGL went public in 2004.
Open	Price at which the first trade of the day occurs.
High	Highest price at which stock is traded for the day.
Low	Lowest price at which stock is traded for the day.
Close	Price at which the last trade of the day occurs.
Adj Close	Close price that accounts for any corporate actions that may affect the stock price, such as dividends, stock splits, or mergers.
Volume	Total number of shares or contracts traded for the day.
Company	Specific corporation or entity. For the project, we fixed the company as GOOGL as we are only focused on the GOOGL stock.
Tomorrow	Tomorrow is the price of the stock on the subsequent day.

2.3. Table 2

As we are doing time series classification of stock prices, it is not sufficient to only take one day's data as an input. One day's data may not provide enough context or historical information for the model to learn meaningful patterns in the data. Furthermore, daily data can be noisy and exhibit significant variability, especially for high-frequency data. Using longer time periods can help smooth out some of this noise and capture more stable trends.

This table shows the Adj Close price of the last 60 days and that of the current date.

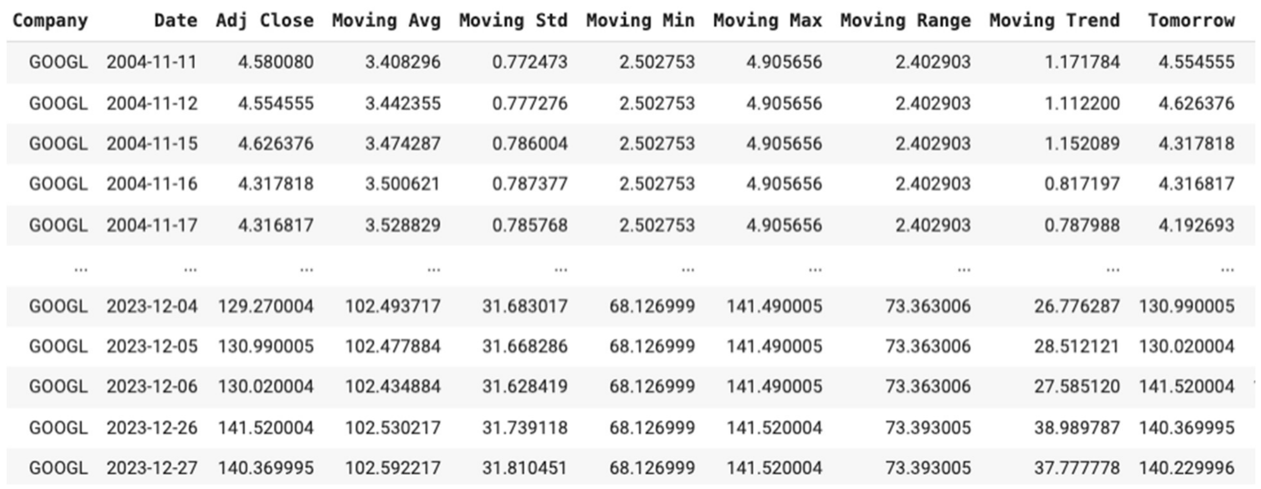


Date	AC 0	AC 1	AC 2	AC 3	AC 4	AC 5	AC 6	AC 7	AC 8	...	AC 51	AC 52	AC 53	AC 54	AC 55	AC 56	AC 57	AC 58	AC 59	AC Current
2004-11-12	2.511011	2.71046	2.737738	2.624374	2.652653	2.70045	2.656406	2.552803	2.561812	...	4.905656	4.876627	4.796547	4.622122	4.237988	4.318068	4.221722	4.200701	4.58008	4.554555
2004-11-15	2.71046	2.737738	2.624374	2.652653	2.70045	2.656406	2.552803	2.561812	2.508759	...	4.876627	4.796547	4.622122	4.237988	4.318068	4.221722	4.200701	4.58008	4.554555	4.626376
2004-11-16	2.737738	2.624374	2.652653	2.70045	2.656406	2.552803	2.561812	2.508759	2.54029	...	4.796547	4.622122	4.237988	4.318068	4.221722	4.200701	4.58008	4.554555	4.626376	4.317818
2004-11-17	2.624374	2.652653	2.70045	2.656406	2.552803	2.561812	2.508759	2.54029	2.502753	...	4.622122	4.237988	4.318068	4.221722	4.200701	4.58008	4.554555	4.626376	4.317818	4.316817
2004-11-18	2.652653	2.70045	2.656406	2.552803	2.561812	2.508759	2.54029	2.502753	2.542042	...	4.237988	4.318068	4.221722	4.200701	4.58008	4.554555	4.626376	4.317818	4.316817	4.192693
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2023-12-05	131.940002	132.600006	135.800003	136.649994	138.339996	140.419998	141.490005	68.720001	70.458	...	136.970001	138.490005	136.690002	136.410004	137.199997	134.990005	132.529999	131.860001	129.270004	130.990005
2023-12-06	132.600006	135.800003	136.649994	138.339996	140.419998	141.490005	68.720001	70.458	70.337502	...	138.490005	136.690002	136.410004	137.199997	134.990005	132.529999	131.860001	129.270004	130.990005	130.020004
2023-12-26	135.800003	136.649994	138.339996	140.419998	141.490005	68.720001	70.458	70.337502	70.662003	...	136.690002	136.410004	137.199997	134.990005	132.529999	131.860001	129.270004	130.990005	130.020004	141.520004
2023-12-27	136.649994	138.339996	140.419998	141.490005	68.720001	70.458	70.337502	70.662003	71.068497	...	136.410004	137.199997	134.990005	132.529999	131.860001	129.270004	130.990005	130.020004	141.520004	140.369995
2023-12-28	138.339996	140.419998	141.490005	68.720001	70.458	70.337502	70.662003	71.068497	71.014	...	137.199997	134.990005	132.529999	131.860001	129.270004	130.990005	130.020004	141.520004	140.369995	140.229996

Feature	**Description**
Date	Specific date on which trading activity occurred.
AC [x] x in range [0, 59]	AC = Adj Close AC [x] = the Adj Close price (60-x) days before the current date Adj Close is the price at which the last trade of the day occurs and it also accounts for any corporate actions that may affect the stock price, such as dividends, stock splits, or mergers.
AC current	Adj Close price of the current day.

2.4. Table 3

This table consists of various moving metrics of AC prices with a window period of 60 days.



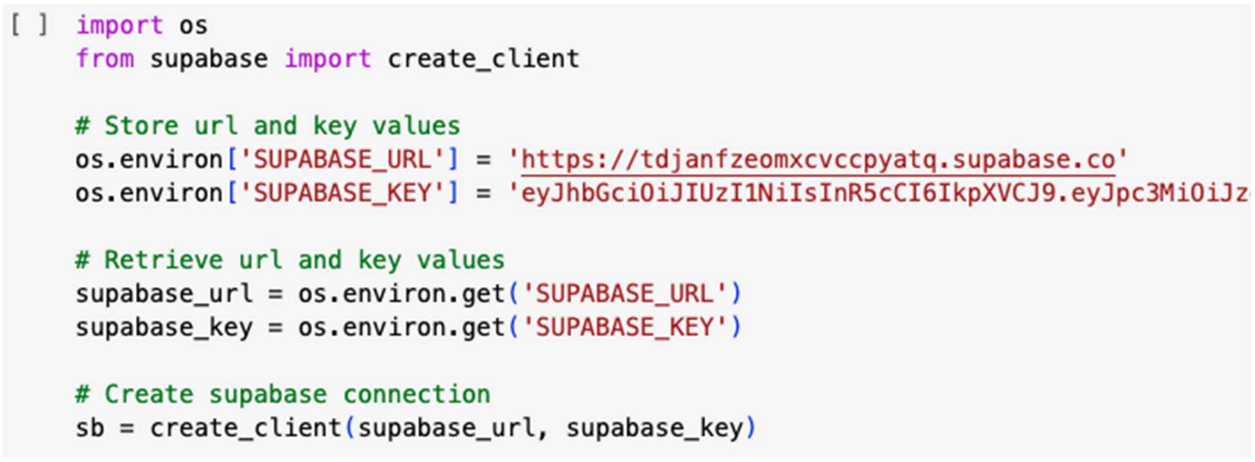
Company	Date	Adj Close	Moving Avg	Moving Std	Moving Min	Moving Max	Moving Range	Moving Trend	Tomorrow
GOOGL	2004-11-11	4.580080	3.408296	0.772473	2.502753	4.905656	2.402903	1.171784	4.554555
GOOGL	2004-11-12	4.554555	3.442355	0.777276	2.502753	4.905656	2.402903	1.112200	4.626376
GOOGL	2004-11-15	4.626376	3.474287	0.786004	2.502753	4.905656	2.402903	1.152089	4.317818
GOOGL	2004-11-16	4.317818	3.500621	0.787377	2.502753	4.905656	2.402903	0.817197	4.316817
GOOGL	2004-11-17	4.316817	3.528829	0.785768	2.502753	4.905656	2.402903	0.787988	4.192693
...	...	...	...	...	...	...	...	...	...
GOOGL	2023-12-04	129.270004	102.493717	31.683017	68.126999	141.490005	73.363006	26.776287	130.990005
GOOGL	2023-12-05	130.990005	102.477884	31.668286	68.126999	141.490005	73.363006	28.512121	130.020004
GOOGL	2023-12-06	130.020004	102.434884	31.628419	68.126999	141.490005	73.363006	27.585120	141.520004
GOOGL	2023-12-26	141.520004	102.530217	31.739118	68.126999	141.520004	73.393005	38.989787	140.369995
GOOGL	2023-12-27	140.369995	102.592217	31.810451	68.126999	141.520004	73.393005	37.777778	140.229996

Feature	**Description**
Company	Specific corporation or entity. For the project, we fixed the company as GOOGL as we are only focused on the GOOGL stock.
Date	Specific date on which trading activity occurred.
Adj Close	Close price that accounts for any corporate actions that may affect the stock price, such as dividends, stock splits, or mergers.
Moving Avg	Average of close prices of the past 60 days.
Moving Std	Standard deviation of close prices in the past 60 days.
Moving Min	Minimum of close prices in the past 60 days.
Moving Max	Maximum of close prices in the past 60 days.
Moving Range	Range of close prices in the past 60 days (Moving Max - Moving Min).
Moving Trend	Adj Close - Moving Avg.
Tomorrow	Tomorrow is the price of the stock on the subsequent day.

2.5. Data Storage and Retrieval

Data is stored in Supabase. Supabase is chosen as it is easy to set up databases and APIs with it. Additionally, Supabase can handle large amounts of data should we choose to expand our project in the future.

In each Colab file for this project, we start with a "Fetch Data" segment. This segment is where we set up a Supabase instance using the same Supabase URL and key. This approach ensures that data access is streamlined and consistent across all files.



```
[ ]  import os
     from supabase import create_client

     # Store url and key values
     os.environ['SUPABASE_URL'] = 'https://tdjanfzeomxcvccpyatq.supabase.co'
     os.environ['SUPABASE_KEY'] = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJz

     # Retrieve url and key values
     supabase_url = os.environ.get('SUPABASE_URL')
     supabase_key = os.environ.get('SUPABASE_KEY')

     # Create supabase connection
     sb = create_client(supabase_url, supabase_key)
```

There are also two helper functions written to help insert and retrieve data.

3. Models

3.1. Linear Regression

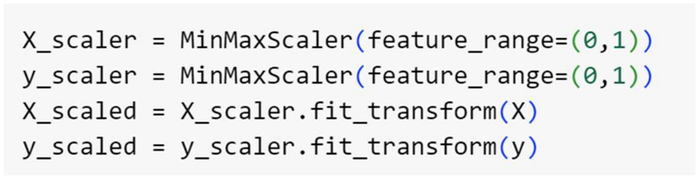
Refer to the `Linear_Reg.ipynb` file.

As a baseline, we used the linear regression model on our data. Briefly, a linear model is defined by the following equation:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^{\top}\,\mathbf{x} = \overbrace{\theta_d x_d + \cdots + \theta_2 x_2 + \theta_1 x_1}^{\text{Slopes of hyper-plane}} + \underbrace{\theta_0}_{\text{bias (offset)}}$$

where the thetas are linear and the 2 parameters are the slope and the bias.

3.1.1. Scaling

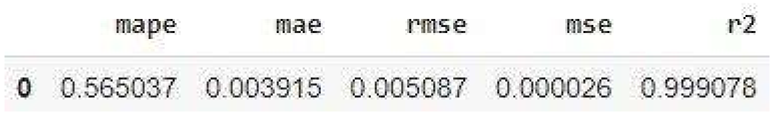


```
X_scaler = MinMaxScaler(feature_range=(0,1))
y_scaler = MinMaxScaler(feature_range=(0,1))
X_scaled = X_scaler.fit_transform(X)
y_scaled = y_scaler.fit_transform(y)
```

Figure: Scaling of X and y features

First, we used MinMaxScaler to scale the X and y features, so that they will be positive. Then, we fitted our model using features 'Open', 'High', 'Low' and 'Volume' to predict 'Adj Close'.

<u>**Initial Results**</u>



	mape	mae	rmse	mse	r2
0	0.565037	0.003915	0.005087	0.000026	0.999078

Figure: Evaluation metrics of Vanilla Linear Regression

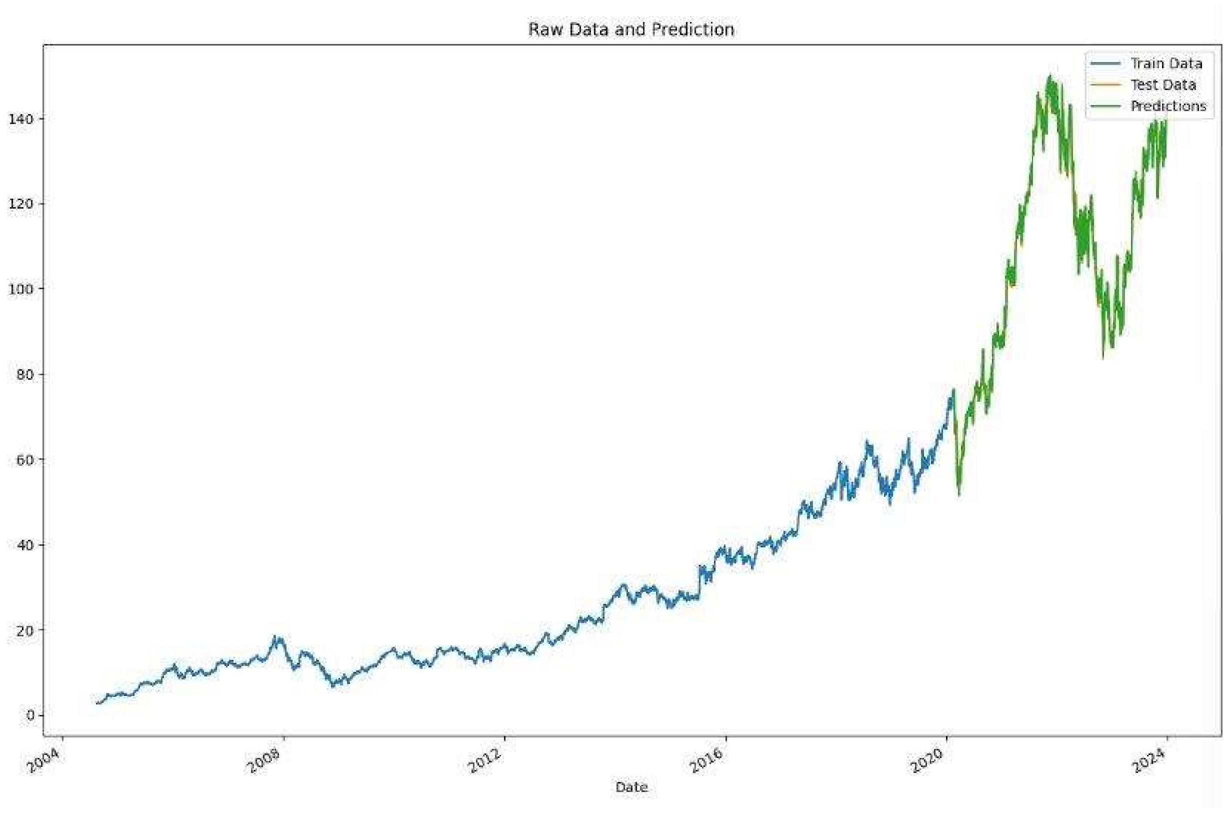
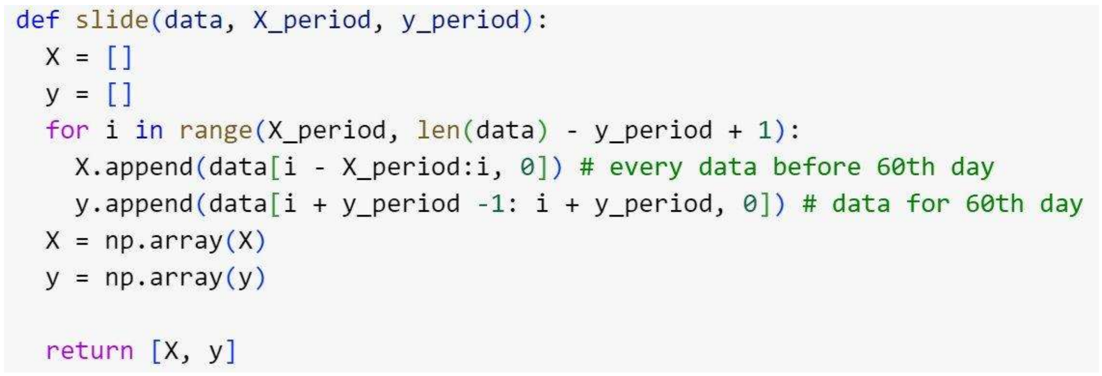


Figure: Plot of Actual Stock Price data against Prediction

The evaluation metrics are very low and when we plot predictions of the adjusted close on the last 20% of the data, they show that the linear model is overfitting as shown us not being able to distinguish the predictions from the test data above.

3.1.2. Time-series Feature Engineering via Sliding Window



```
def slide(data, X_period, y_period):
  X = []
  y = []
  for i in range(X_period, len(data) - y_period + 1):
    X.append(data[i - X_period:i, 0]) # every data before 60th day
    y.append(data[i + y_period -1: i + y_period, 0]) # data for 60th day
  X = np.array(X)
  y = np.array(y)

  return [X, y]
```

Figure: Implementation of Sliding Window

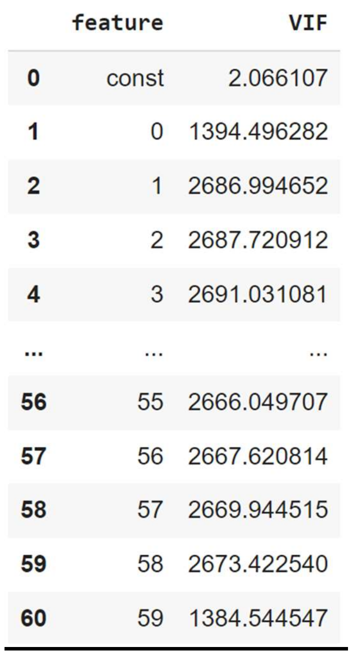
Next, we engineered the linear regression model to use the 'Adj Close' of the past 60 days by implementing a sliding window.

3.1.3. Variance Inflation Factors

We check for multicollinearity by using Variance Inflation Factors (VIF), which has the following equation:

$$VIF_i = \frac{1}{1 - R_i^2}$$

, where Ri is the multiple R for the regression of Xi on the other covariates.

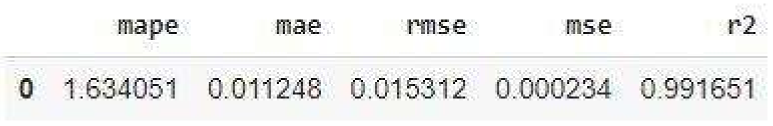


	feature	VIF
0	const	2.066107
1	0	1394.496282
2	1	2686.994652
3	2	2687.720912
4	3	2691.031081
...	...	...
56	55	2666.049707
57	56	2667.620814
58	57	2669.944515
59	58	2673.422540
60	59	1384.544547

Figure: Variance Inflation Factors on Time-series features

The VIF values are much higher than 10 for all the variables, which indicates that there is high multicollinearity. This is because we are using time-series data, one day's stock prices is likely to be dependent on the previous days' stock prices.

<u>Results after Time-series Feature Engineering</u>



	mape	mae	rmse	mse	r2
0	1.634051	0.011248	0.015312	0.000234	0.991651

Figure: Evaluation metrics of Linear Regression with Time-series Feature Engineering

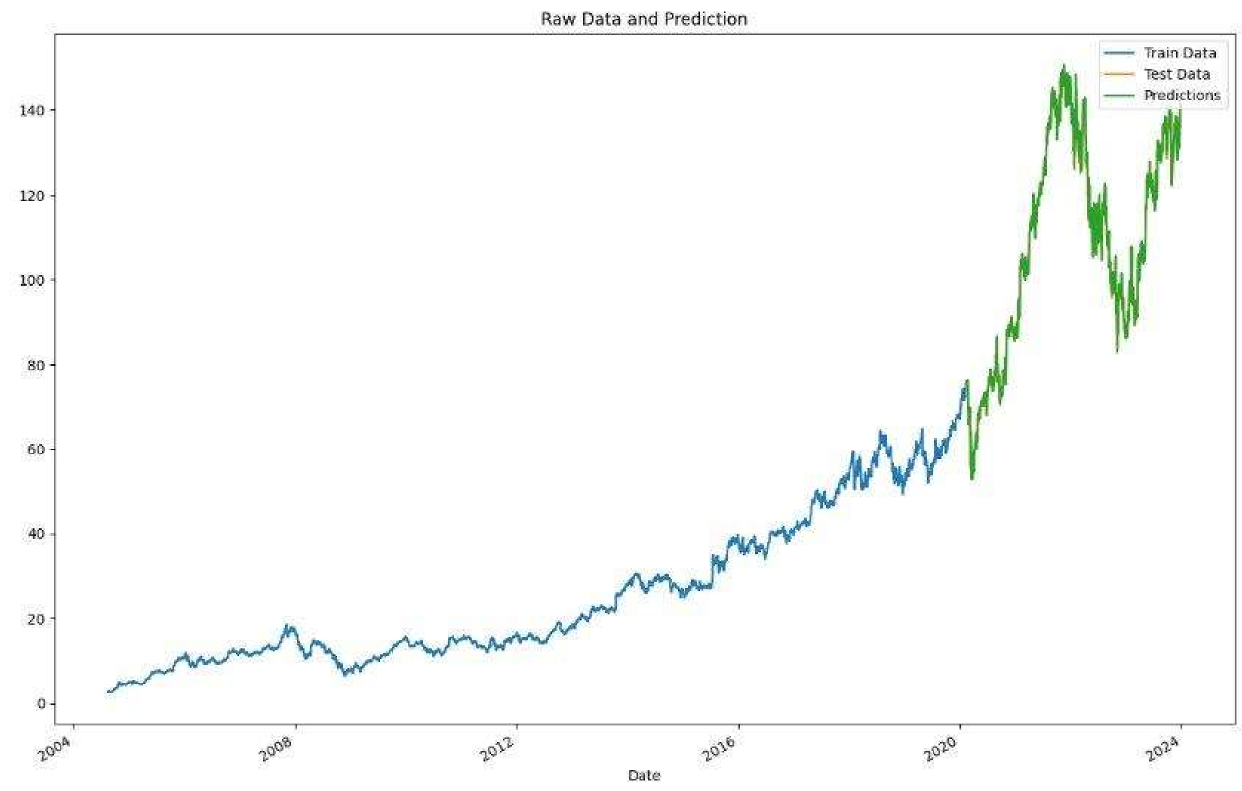


Figure: Plot of Actual Stock Price data against Prediction of Linear Regression with Time-series Feature Engineering

The metrics show a slight increase in the errors, but the overall plot still shows the overfitting of the model.

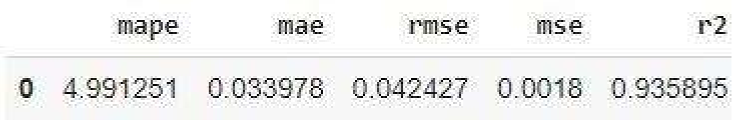
3.1.4. Ridge Regression

Lastly, to combat the overfitting problem, we performed ridge regression to shrink all coefficients toward zero, so that the model would be less sensitive to noise.

$$min_{\theta} L_{task}(\theta) \text{ s.t.} ||\theta||_2^2 \leq C \Leftrightarrow min_{\theta} L_{task}(\theta) + \lambda||\theta||_2^2$$

The penalty term, lambda, is proportional to the square of the magnitude of the coefficients.

<u>Ridge Regression Results</u>



	mape	mae	rmse	mse	r2
0	4.991251	0.033978	0.042427	0.0018	0.935895

Figure: Evaluation metrics of Ridge Regression

Using the same Time-series features, the metrics show that the errors of the ridge regression model are still low, but larger than that of the previous 2 models. This is due to the bias-variance trade-off as Ridge Regression tends to reduce the variance, which will increase the bias. The overfitting is reduced but it still seems that there is some overfitting.

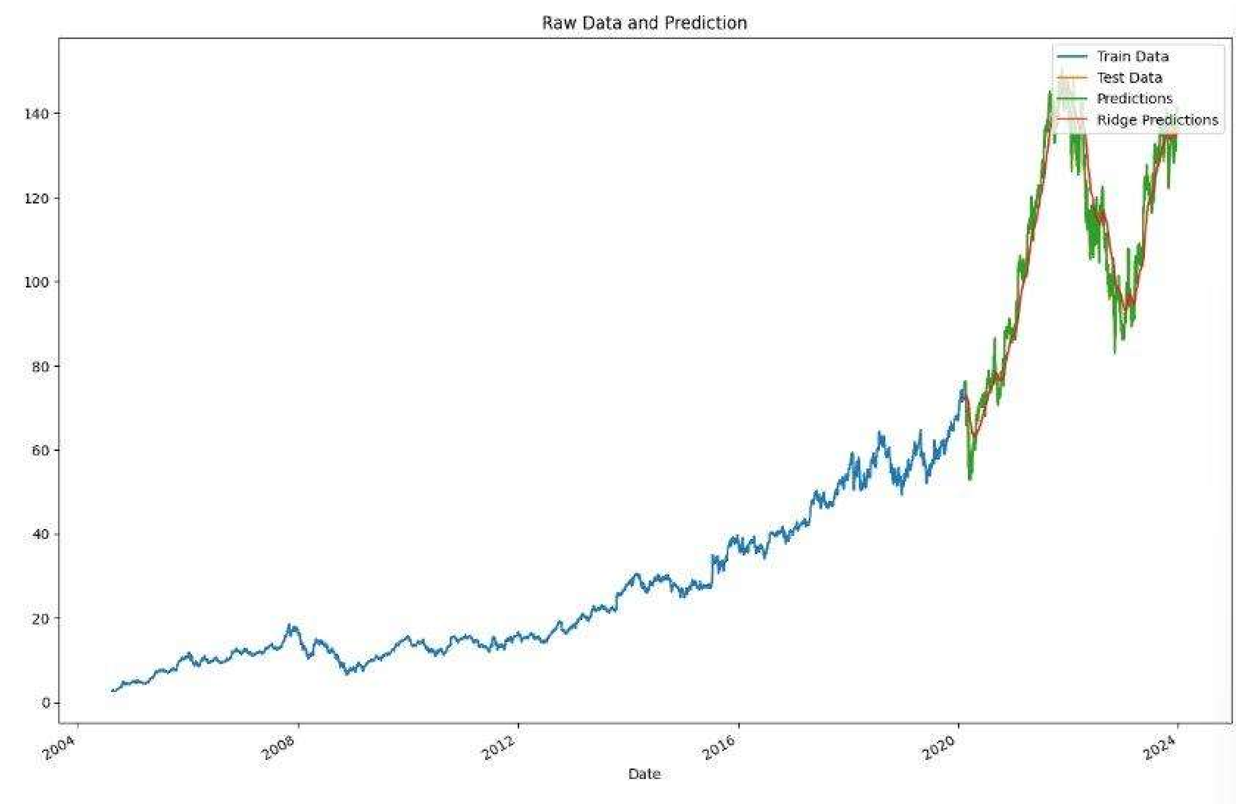


Figure: Plot of Actual Stock Price data against Prediction of Ridge Regression

The plot shows a smoother red curve but some portions still overfit. This is caused by high multicollinearity.

3.2. XGBoost

Refer to the `XGBoost.ipynb` file.

We used the XGBRegressor to create the regression model. XGBoost utilizes decision trees as base learners and employs regularization techniques to enhance model generalization. The equation for an XGBoost regression model can be written as:

$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$$

- $\hat{y}$ is the predicted value for the i-th instance
- x_i is the feature vector for the i-th instance
- K is the number of trees (boosting rounds)
- f_k is the prediction function of the k-th tree

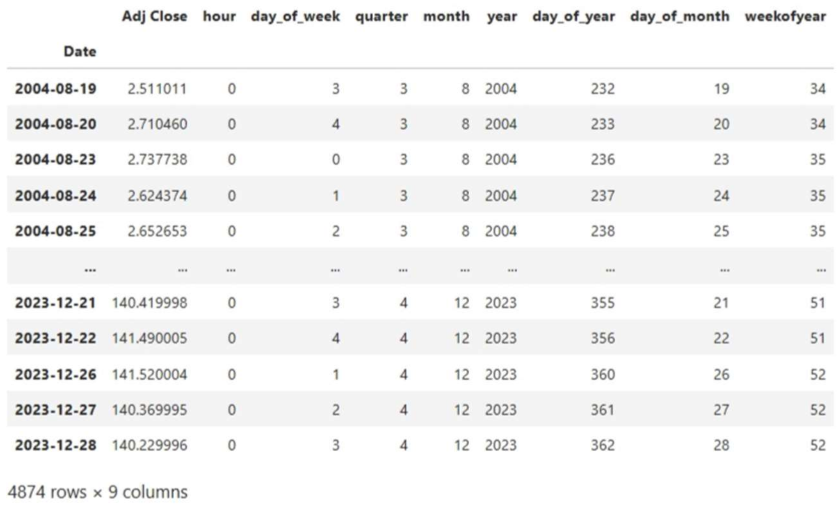
It has various parameters that can be tuned to customise the model, such as base_score, n_estimators, max_depth, learning_rate, etc.

3.2.1 Parameters

- **base_score:** The initial prediction score of all instances, which serves as a baseline.
- **booster:** The type of boosting algorithm to use. Here, 'gbtree' indicates that decision trees are used as base learners.
- **n_estimators:**The number of boosting rounds (trees) to build. In this case, 1000 trees will be built.
- **early_stopping_rounds:** The number of rounds with no improvement in the validation score to stop boosting. Here, if there is no improvement for 50 rounds, the boosting process will stop.

- **objective:** The loss function to be optimized. 'reg:squarederror' indicates that the model will optimise mean squared error for regression.
- **max_depth:** The maximum depth of the decision trees.
- **learning_rate:** The step size shrinkage used in each boosting step to prevent overfitting.

To give the decision trees some predictors we decompose the time/date variable into multiple features:



	Adj Close	hour	day_of_week	quarter	month	year	day_of_year	day_of_month	weekofyear
Date									
2004-08-19	2.511011	0	3	3	8	2004	232	19	34
2004-08-20	2.710460	0	4	3	8	2004	233	20	34
2004-08-23	2.737738	0	0	3	8	2004	236	23	35
2004-08-24	2.624374	0	1	3	8	2004	237	24	35
2004-08-25	2.652653	0	2	3	8	2004	238	25	35
...	...	...	...	...	...	...	...	...	...
2023-12-21	140.419998	0	3	4	12	2023	355	21	51
2023-12-22	141.490005	0	4	4	12	2023	356	22	51
2023-12-26	141.520004	0	1	4	12	2023	360	26	52
2023-12-27	140.369995	0	2	4	12	2023	361	27	52
2023-12-28	140.229996	0	3	4	12	2023	362	28	52

4874 rows × 9 columns

Figure: Dataset with varying time features

3.2.2 Feature Relationships

We then visualize the relationship between the price and time periods:

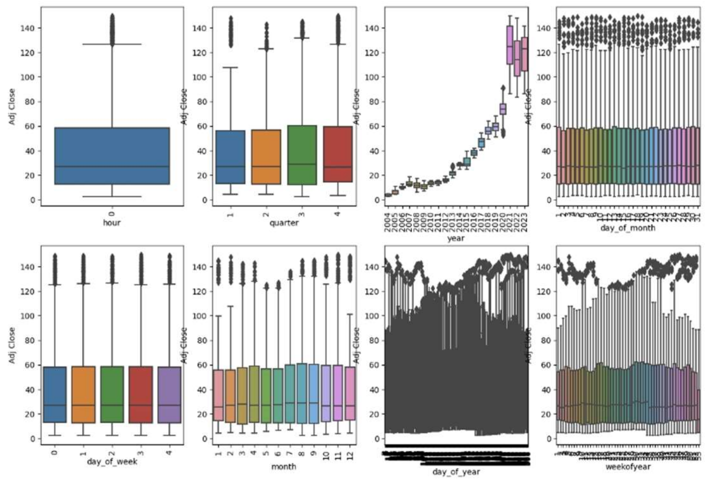


Figure: relationship between the Price and Time features

3.2.3 Time Series/Randomized Split

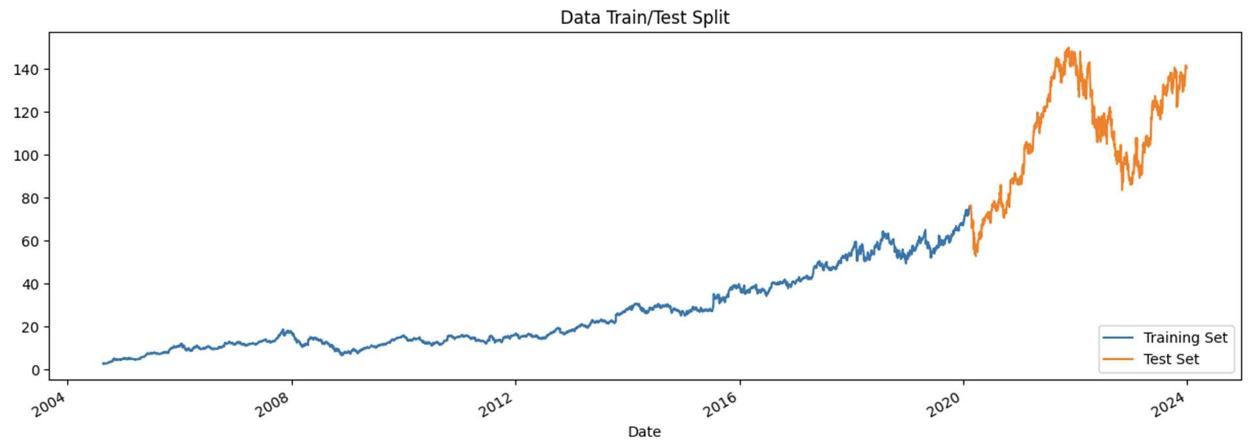


Figure: Plot of Time Series Split

We noticed that there was a possible issue with how to split the dataset.

Initially, we wanted to use the dataset first 80 percent of the data to predict the latest 20 percent data in line with the nature of the time series data. However, we can see that the value of the stock price in test data has a range that exceeds the training data's price range. This means that the decision tree in the XGBoost model may not be able to handle the time series split. Hence, we attempted to split the data by time series and by randomized sampling and observe the outcome.

3.2.4 Feature Importance

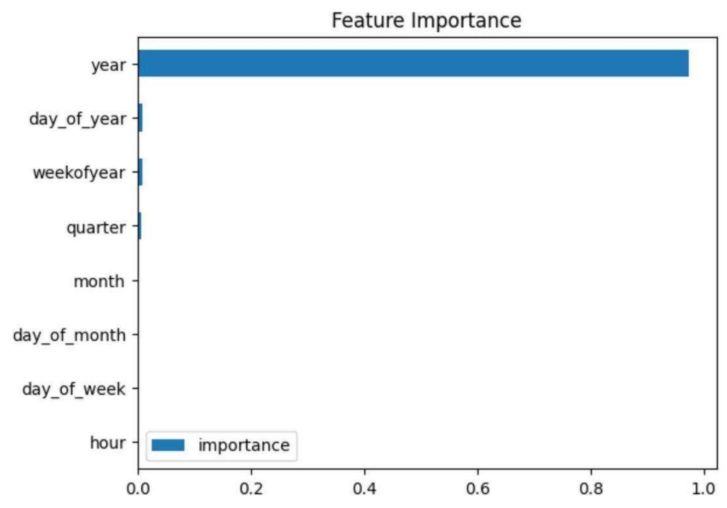
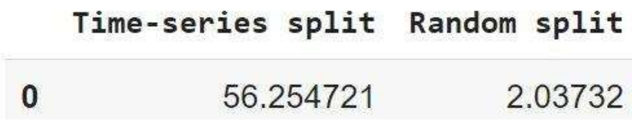


Figure: Measure of Feature Importance

We then plot the Feature Importance of the model fit and note that year is the most important feature to predict Adjusted Close in both the time series and randomized split.

Initial Results



	Time-series split	Random split
0	56.254721	2.03732

Figure: Root Mean Square Error on test data

We find that the performance of the model using random split is significantly higher than time series split.

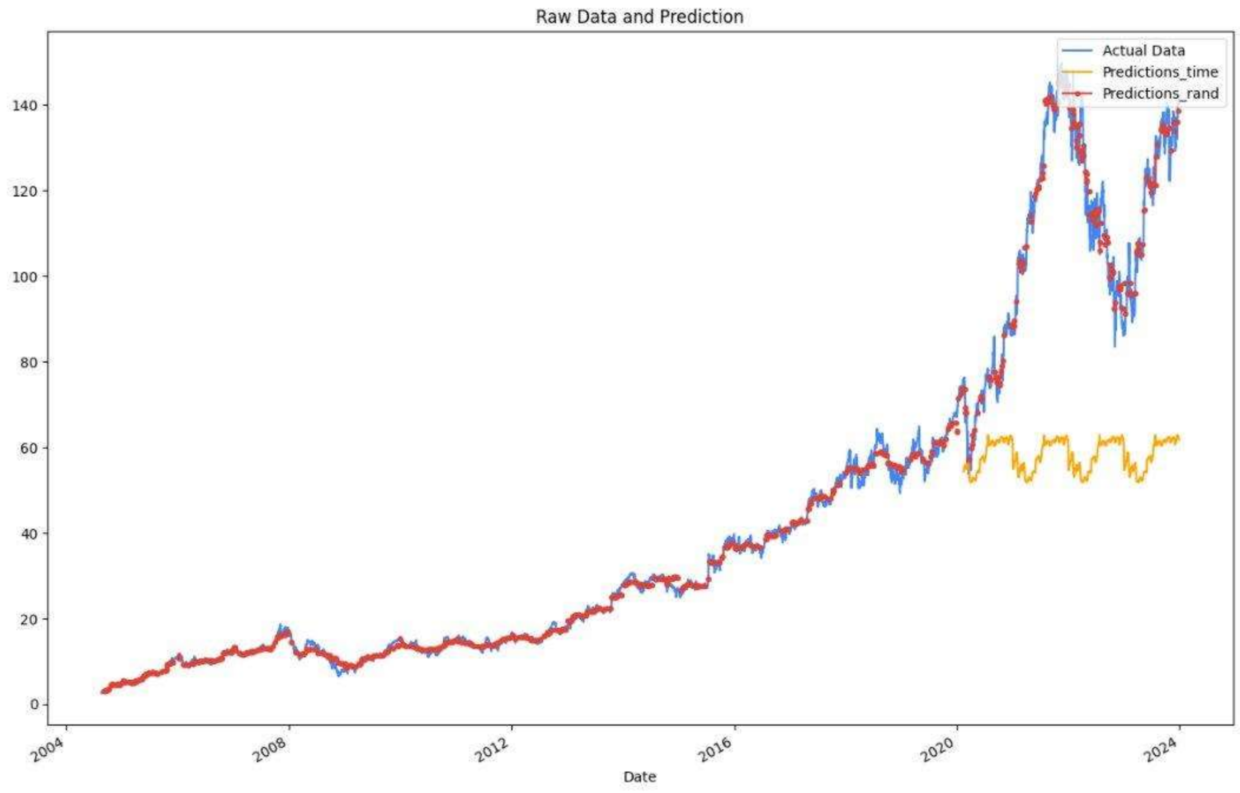


Figure: Plot of Actual Stock Price data against Prediction

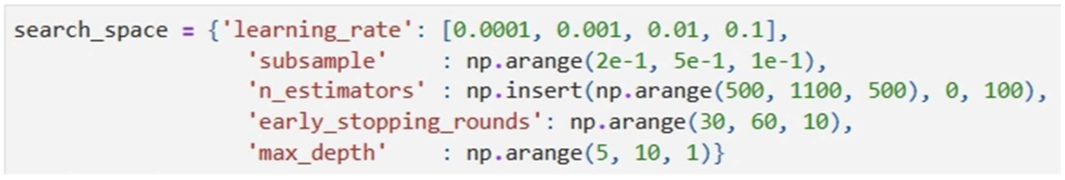
In this plot, the yellow line represents the predictions made by the XGBoost model using time series split, while the red line represents the predictions made by the model using randomized split. And we can observe that time series split predictions are unable to predict properly along the test data set, while the randomized split predictions are able to. We then proceeded with randomized splitting.

3.2.5 Hypertuning

To improve the performance of the model, we performed grid search.

GridSearchCV can help find the optimal combination of hyperparameters for the XGBoost regressor. It performs an exhaustive search over a specified parameter grid to find the best parameters. The objective is to minimize the loss value over the validation dataset.

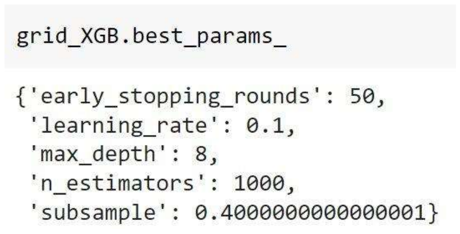
The search space is as follows:



```
search_space = {'learning_rate': [0.0001, 0.001, 0.01, 0.1],
                'subsample'    : np.arange(2e-1, 5e-1, 1e-1),
                'n_estimators' : np.insert(np.arange(500, 1100, 500), 0, 100),
                'early_stopping_rounds': np.arange(30, 60, 10),
                'max_depth'    : np.arange(5, 10, 1)}
```

Figure: Grid Search Search Space

When we executed the grid search, we have



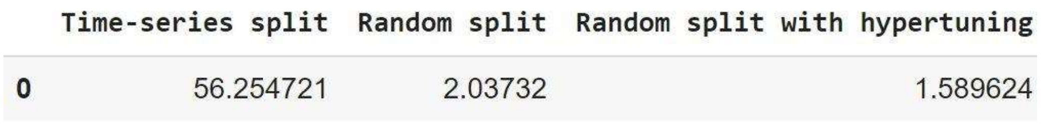
```
grid_XGB.best_params_
```

```
{'early_stopping_rounds': 50,
 'learning_rate': 0.1,
 'max_depth': 8,
 'n_estimators': 1000,
 'subsample': 0.4000000000000001}
```

Figure: Hyperparameters of Hypertuned Model

3.2.6 Hypertuned Results

Repeating the training and prediction process with the hypertuned model, we have;



	Time-series split	Random split	Random split with hypertuning
0	56.254721	2.03732	1.589624

Figure: Root Mean Square Error on test data

Here we can see that with hypertuning, the RMSE is lower.

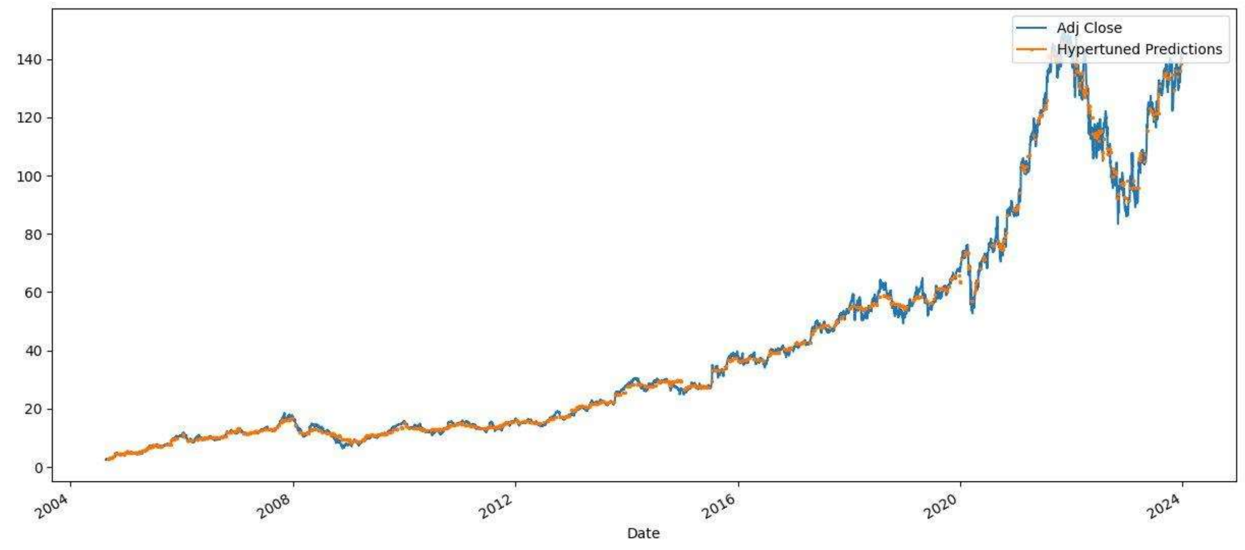


Figure: Plot of Actual Stock Price data against Hypertuned Prediction

In this plot, the yellow line represents the predictions made by the Hypertuned XGBoost model using time series split. We note that there is no significant shift besides the smaller RMSE value.

3.3. SARIMA

Refer to the `Arima model.ipynb` file.

3.3.1. Motivation

As our data is time-series data with some seasonal patterns, we used Seasonal Autoregressive Integrated Moving Average (SARIMA) model to predict the adjusted close price. SARIMA is an extension of the Autoregressive Integrated Moving Average (ARIMA) model with a seasonality component. It works by modeling the link between past and current values of a time series and recognizing patterns in the data.

For the SARIMA model, the parameters are typically denoted as:

ARIMA(p, d, q) x (P, D, Q)S

where:

- p: The number of autoregressive (AR) terms for the non-seasonal component.
- d: The degree of differencing for the non-seasonal component.
- q: The number of moving average (MA) terms for the non-seasonal component.
- P : The number of seasonal autoregressive (SAR) terms.
- D : The degree of seasonal differencing
- Q : The number of seasonal moving average (SMA) terms.
- S : The seasonal period

3.3.2. Analysis of Time-Series Data

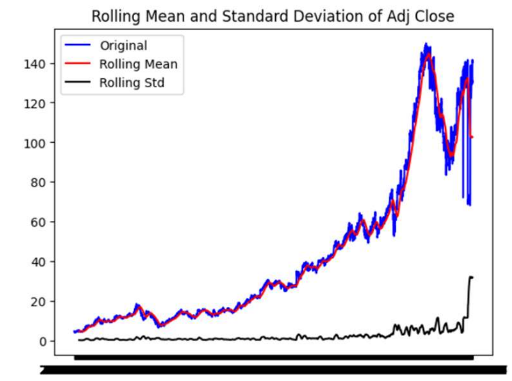
Augmented Dickey Fuller (ADF) Test

As time-series analysis only works for stationary data, we will first determine whether the Adj Close is stationary using the Augmented Dickey-Fuller (ADF) test. The series is stationary if the mean and variance are relatively constant. The test's null and alternate hypothesis are:

H0 : The series has a unit root.

H1 : The series has no unit root.

If the null hypothesis is not rejected, the series is said to be non-stationary.



```
Results of dickey fuller test
Test Statistics                  -0.055389
p-value                           0.953705
No. of lags used                 32.000000
Number of observations used    4781.000000
critical value (1%)              -3.431719
critical value (5%)              -2.862145
critical value (10%)             -2.567092
dtype: float64
```

Figure: Plot of ADF test

From the plot, we can see that the series is not stationary as the mean and standard deviation is increasing. Furthermore the p-value is > 0.05, which means we do not reject the null hypothesis.

<u>Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) Test</u>

We used ACF and PACF to analyze the autocorrelation of data in order to understand the patterns between time steps.

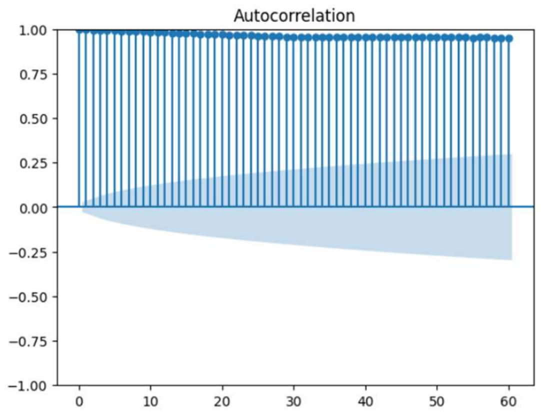


Figure: ACF Plot

The ACF plot is gradually decreasing. This indicates that there is a long-range dependence or persistence in the time series data. This means that future values are highly correlated or affected by past values. The presence of a strong positive autocorrelation, especially with a gradual decrease, could also indicate the presence of a trend or seasonality in the data.

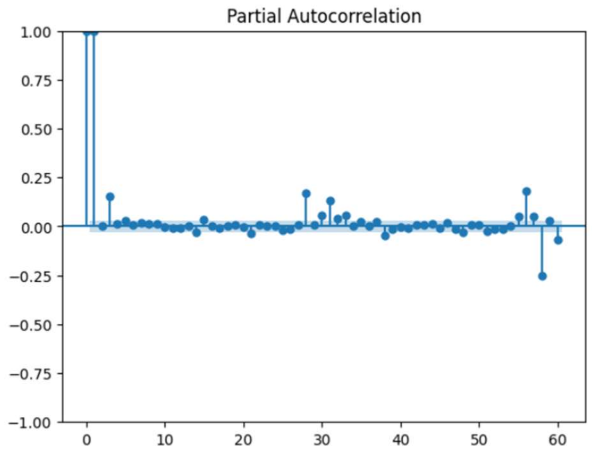


Figure: PACF Plot

The constant PACF at lag 1 and 2 suggests that there is a strong, consistent relationship between the current value of the time series and the values one and two time periods ago. This implies that the series may exhibit a strong AR(2) pattern.The significant drop in PACF at lag 2 indicates that once the first two lagged values are accounted for, the correlation between the observation and the second lagged value becomes insignificant. This drop suggests that the higher-order autoregressive relationship is primarily driven by the first two lags.

The combination of a constant PACF at lag 1 and 2, a significant drop at lag 2, and a gradually decreasing ACF suggests that the time series may exhibit a seasonal pattern that can be captured using a SARIMA model.

Isolating the Time Series from Trend and Seasonality

We used multiplicative decomposition to decompose the time series into its components. The equation for multiplicative decomposition is:

$y_t = S_t \text{ x } T_t \text{ x } R_t$

where:

- y_t is the data
- S_t is the seasonal component
- T_t is the trend-cycle component
- R_t is the remainder component, all at period t

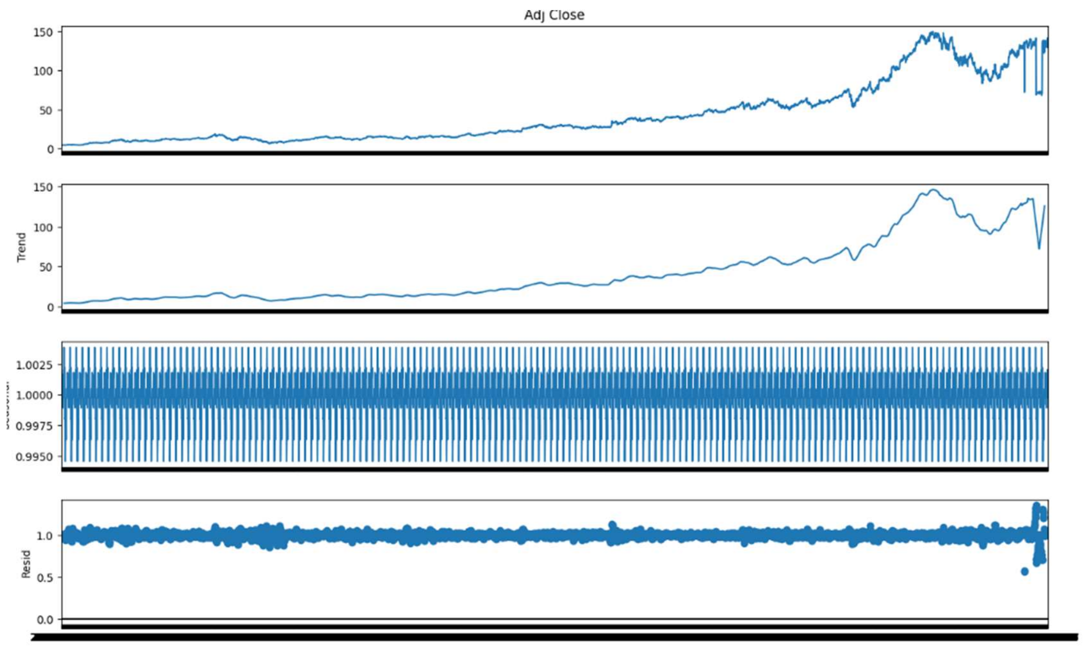


Figure: Multiplicative Decomposition

From the seasonal plot, there is a seasonal pattern to the data.

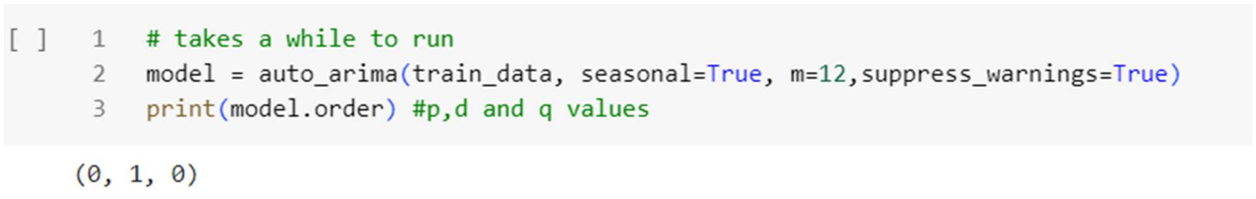
3.3.3. Initial Implementation

We can see that Adjusted Close follows an exponential relationship with respect to time from the ADF test. We log Adjusted Close to make it more linear. This stabilizes the variance to make the series more stationary.

We used auto_arima() from the 'pmdarima' library to find the optimal p,d and q automatically. We set m=12 for computational limitation reasons. if we put m=365 , it would be computationally expensive. By default, auto_arima() uses Information Criteria as AIC where:

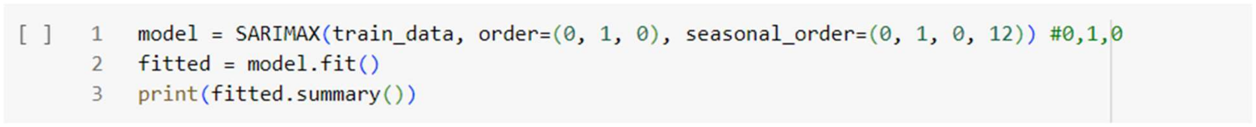
$\mathrm{AIC} = -2\log(L) + 2(p+q+k+1)$ where $k = 1$ if $c! = 0$ and $k = 0$ if $c = 0$.

auto_arima() uses the Hyndman-Khandakar algorithm to obtain a SARIMA model.

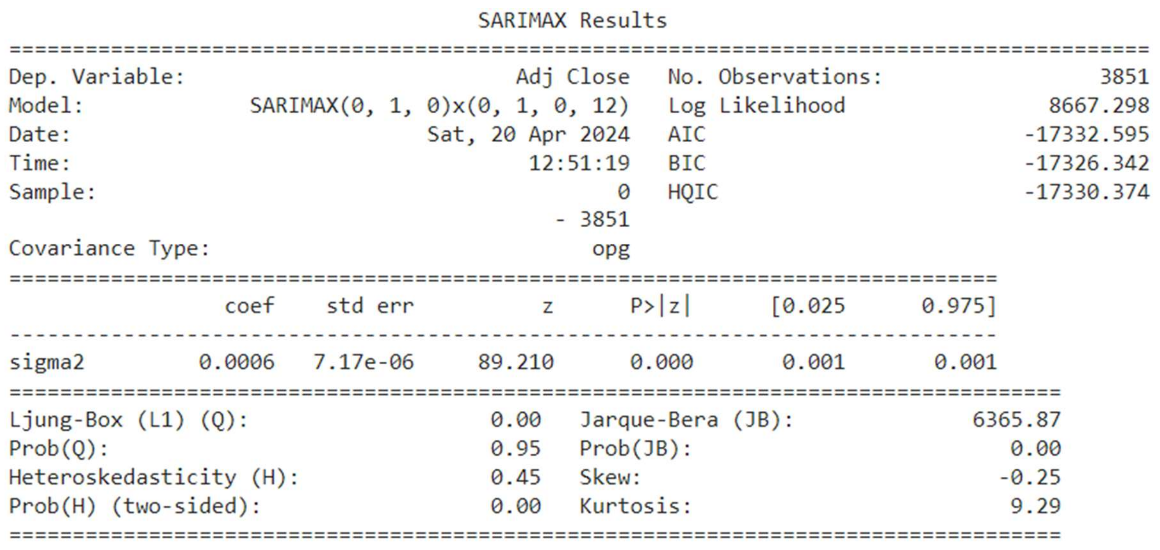


```
[ ]  # takes a while to run
     model = auto_arima(train_data, seasonal=True, m=12,suppress_warnings=True)
     print(model.order) #p,d and q values

  (0, 1, 0)
```

We then fit the model using the p,d, q values from auto_arima().



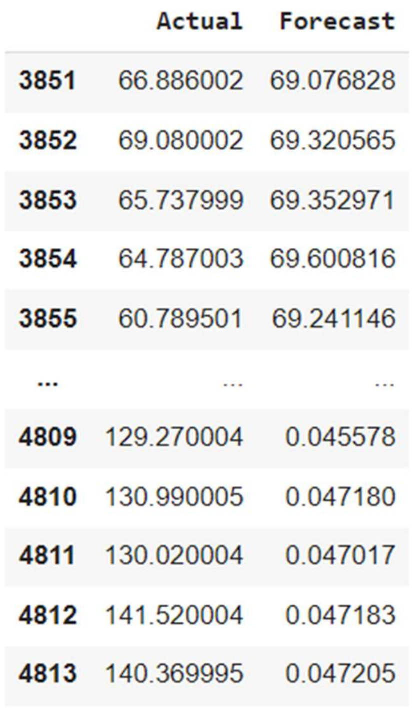
```
[ ]  model = SARIMAX(train_data, order=(0, 1, 0), seasonal_order=(0, 1, 0, 12)) #0,1,0
     fitted = model.fit()
     print(fitted.summary())
```



```
                                     SARIMAX Results
==========================================================================================
Dep. Variable:                          Adj Close   No. Observations:                 3851
Model:             SARIMAX(0, 1, 0)x(0, 1, 0, 12)   Log Likelihood                8667.298
Date:                            Sat, 20 Apr 2024   AIC                         -17332.595
Time:                                    12:51:19   BIC                         -17326.342
Sample:                                         0   HQIC                        -17330.374
                                           - 3851
Covariance Type:                              opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
sigma2         0.0006   7.17e-06     89.210      0.000       0.001       0.001
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):              6365.87
Prob(Q):                              0.95   Prob(JB):                         0.00
Heteroskedasticity (H):               0.45   Skew:                            -0.25
Prob(H) (two-sided):                  0.00   Kurtosis:                         9.29
===================================================================================
```

As we log the data initially, we have to take the inverse transform data to get the original scale of adjusted close.

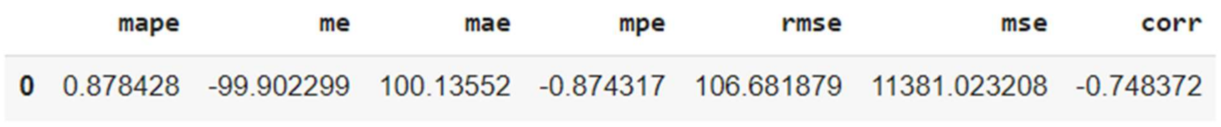
3.3.4. Initial Results



	Actual	Forecast
3851	66.886002	69.076828
3852	69.080002	69.320565
3853	65.737999	69.352971
3854	64.787003	69.600816
3855	60.789501	69.241146
...	...	...
4809	129.270004	0.045578
4810	130.990005	0.047180
4811	130.020004	0.047017
4812	141.520004	0.047183
4813	140.369995	0.047205

Figure: Predictions on test data

Here we see that initially the predictions are close to the actual value, but it becomes very far off for future values.



	mape	me	mae	mpe	rmse	mse	corr
0	0.878428	-99.902299	100.13552	-0.874317	106.681879	11381.023208	-0.748372

Figure: Evaluation metrics on test data

This also results in a very large MSE.

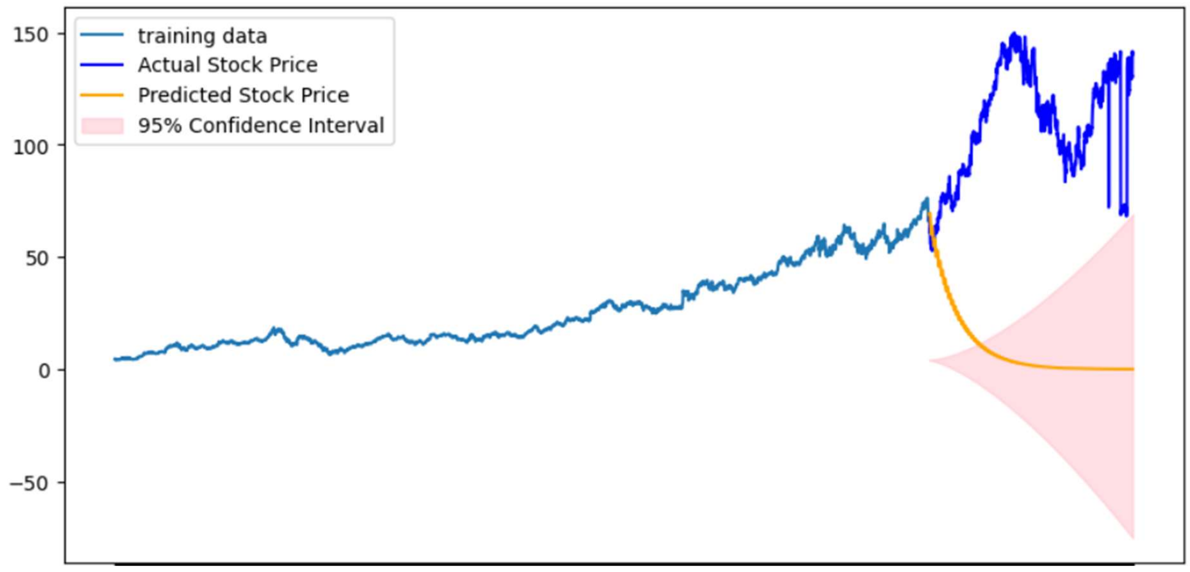
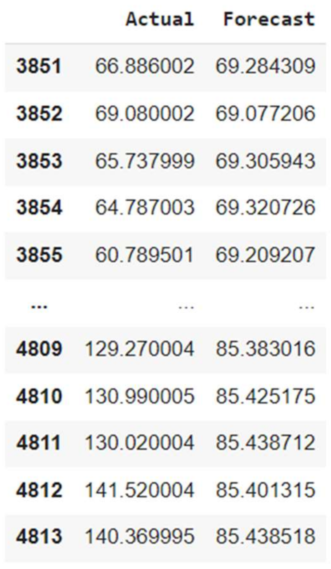


Figure: Plot of predictions vs actual

From the plot, none of the actual values lie in the 95% confidence interval of the predicted values. This shows that we should not take the log of the data. This could be because doing back-transformation may introduce errors, especially for very small values.

3.3.5. SARIMA without Transforming

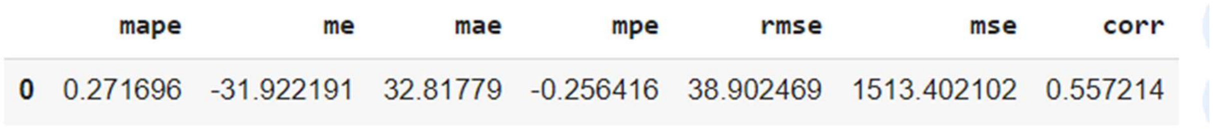
We repeated the steps from 3.4.3 without applying log to the data and here are the results.



	Actual	Forecast
3851	66.886002	69.284309
3852	69.080002	69.077206
3853	65.737999	69.305943
3854	64.787003	69.320726
3855	60.789501	69.209207
...	...	...
4809	129.270004	85.383016
4810	130.990005	85.425175
4811	130.020004	85.438712
4812	141.520004	85.401315
4813	140.369995	85.438518

Figure: Predictions on test data

Similar to the previous results, the predictions were close at the start but become more far off for later data points. The predictions are closer to the actual values compared to the previous results, which results in a lower MSE as shown below.



	mape	me	mae	mpe	rmse	mse	corr
0	0.271696	-31.922191	32.81779	-0.256416	38.902469	1513.402102	0.557214

Figure: Evaluation metrics on test data

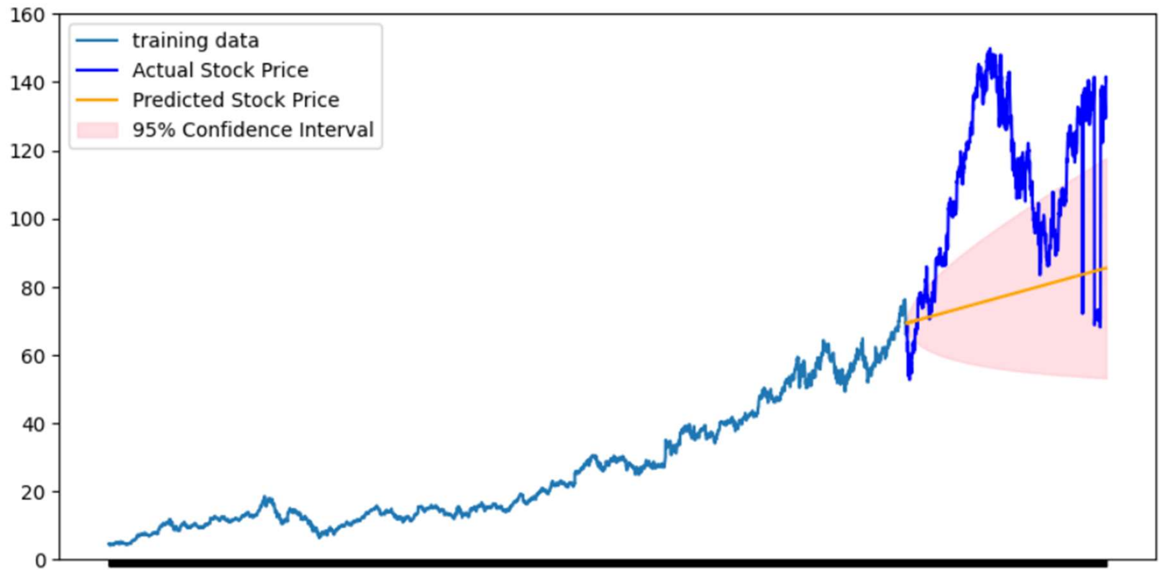


Figure: Plot of predictions against actual values

The plot is more accurate than the previous plot. Differencing is sufficient. The plot is able to predict that the stock prices are increasing but is unable to predict the extent of the increase accurately. Some parts of the actual values lie within the 95% confidence interval of the predicted values.

However, some parts still lie outside of the confidence interval. This could be due to the 'm' component being too small and hence it is unable to capture the seasonal pattern of the data accurately.

3.4. LSTM

Refer to the `LSTM model.ipynb` file.

The components of the LSTM unit are as follows:

- **Cell**: Stores the state of a sequence, so it has the ability to either keep or forget certain information.
- **Input Gate**: It decides the extent of information to be stored in the cell.
- **Output Gate**: It determines what the next hidden state should be.
- **Forget Gate**: It decides what information should be thrown away or kept.

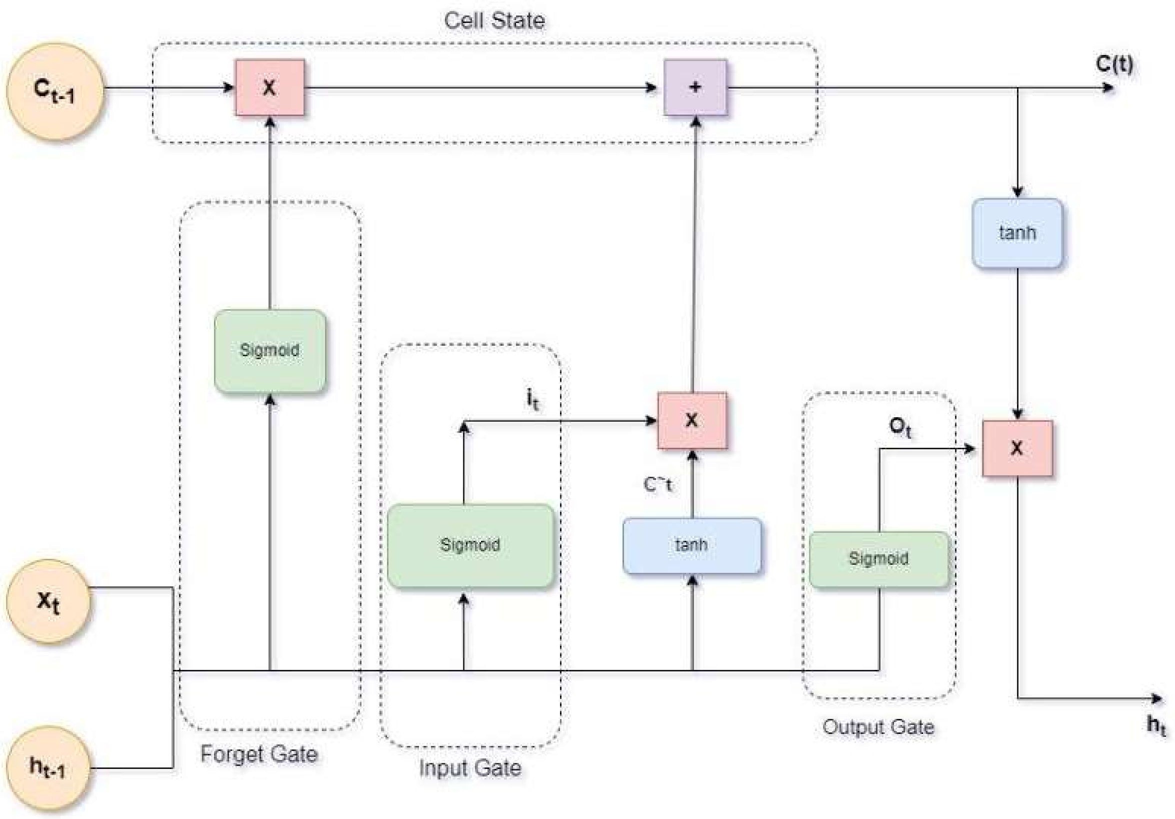


Figure: LSTM Structure (Thomas, 2023)

3.4.1. LSTM Architecture

1. Given input vector, x_t and previous state, h_{t-1}.
   Forget gate decides what information to discard using sigmoid activation.
   A value closer to 0 means to "get rid of it", while closer to 1 means to "keep it":

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t])$$

2. Given input vector, x_t and previous state, h_{t-1}.
   Input gate decides what information to update using sigmoid activation.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t])$$

   and creates a vector of candidate values, C'_t that could be added to the cell state

$$C'_t = tanh(W_i \cdot [h_{t-1}, x_t])$$

3. Given the previous cell state C_{t-1}, forget gate output f_t, input gate output i_t and candidate state C'_t,
   the Cell State, C_t is updated as follows:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot C'_t$$

4. Given input vector, x_t and previous state, h_{t-1}.
   Output gate decides what information to output using sigmoid activation.

$$O_t = \sigma(W_o \cdot [h_{t-1}, x_t])$$

   and hidden state is given as h_t

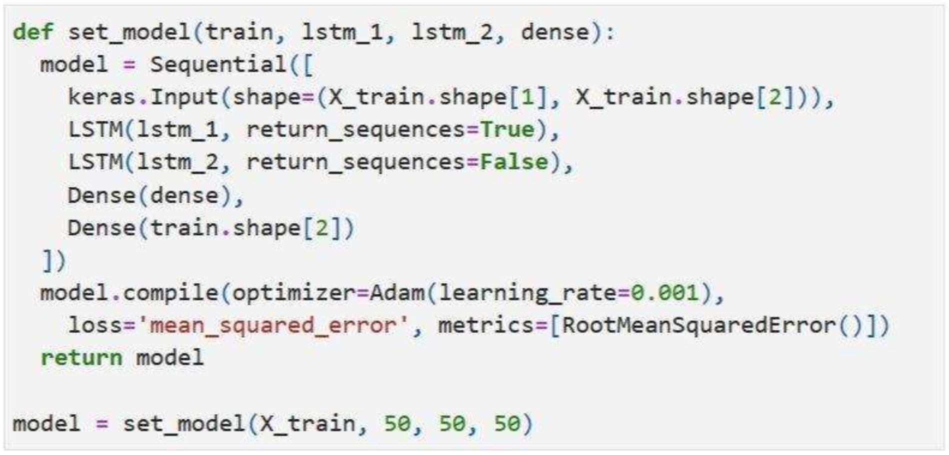
$$h_t = O_t \cdot tanh(C_t)$$

Hence, we choose LSTM for the following reasons:

1. **Long-term Dependencies**: LSTM is capable of learning long-term dependencies.
2. **Handling of Sequential Data**: It can process data with time steps of different lengths, unlike traditional neural networks.

3. **Avoiding Vanishing Gradient Problem**: LSTM networks are designed to avoid the long-term dependency problem or the vanishing gradient problem.

3.4.2. Implementation

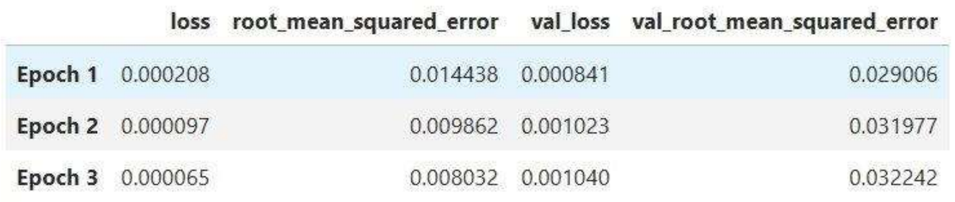


```
def set_model(train, lstm_1, lstm_2, dense):
  model = Sequential([
    keras.Input(shape=(X_train.shape[1], X_train.shape[2])),
    LSTM(lstm_1, return_sequences=True),
    LSTM(lstm_2, return_sequences=False),
    Dense(dense),
    Dense(train.shape[2])
  ])
  model.compile(optimizer=Adam(learning_rate=0.001),
    loss='mean_squared_error', metrics=[RootMeanSquaredError()])
  return model

model = set_model(X_train, 50, 50, 50)
```

Using Keras library, the neural network layers are set as follows:

1. **Input Layer**: **keras.Input(shape=(X_train.shape[1] == 60, X_train.shape[2] == 1))**
    a. Takes in an object dimensions: (No. of rows, No. of columns = 60, No. of features = 1)
2. **LSTM layer 1 : LSTM(lstm_1 == 50 units, return_sequences=True)**
    a. Takes in input layer object
    b. Outputs **all** the hidden states (h) of each time step, t.
3. **LSTM layer 2: LSTM(lstm_2 == 50 units, return_sequences=True)**
    a. Outputs the **last** hidden state (h_T) at the final step.
4. **Dense Layer: Dense(dense == 50 units)**
    a. Takes in hidden state (h_T)
    b. Outputs layer a^(4)
5. **Dense Layer: Dense(X_train.shape[2] == 1)**
    a. Takes in layer a^(4)
    b. Outputs a^(5), which is a single element representing a single day's prediction

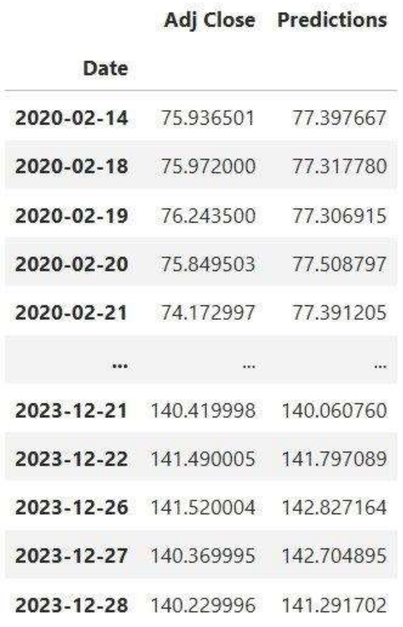
3.4.3. Initial Results



	loss	root_mean_squared_error	val_loss	val_root_mean_squared_error
Epoch 1	0.000208	0.014438	0.000841	0.029006
Epoch 2	0.000097	0.009862	0.001023	0.031977
Epoch 3	0.000065	0.008032	0.001040	0.032242

Figure: Loss values on Training and Validation Data:

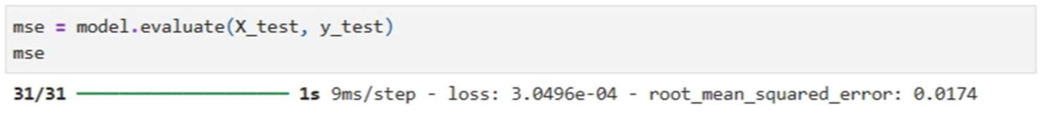
Here we see that the model achieve low loss values which indicates that the model is highly fitting over the training and validation data set.



Date	Adj Close	Predictions
2020-02-14	75.936501	77.397667
2020-02-18	75.972000	77.317780
2020-02-19	76.243500	77.306915
2020-02-20	75.849503	77.508797
2020-02-21	74.172997	77.391205
...	...	...
2023-12-21	140.419998	140.060760
2023-12-22	141.490005	141.797089
2023-12-26	141.520004	142.827164
2023-12-27	140.369995	142.704895
2023-12-28	140.229996	141.291702

Figure: Predictions on Test data

Here we can see the values for the prediction are close to the "Adj Close" price, indicating highly accurate predictions.



```
mse = model.evaluate(X_test, y_test)
mse
```

```
31/31 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - loss: 3.0496e-04 - root_mean_squared_error: 0.0174
```

Figure: Loss values for Test data

We also observe that the loss values on the Test data is also quite low, which thus indicates that the predictions are very close to the actual values.

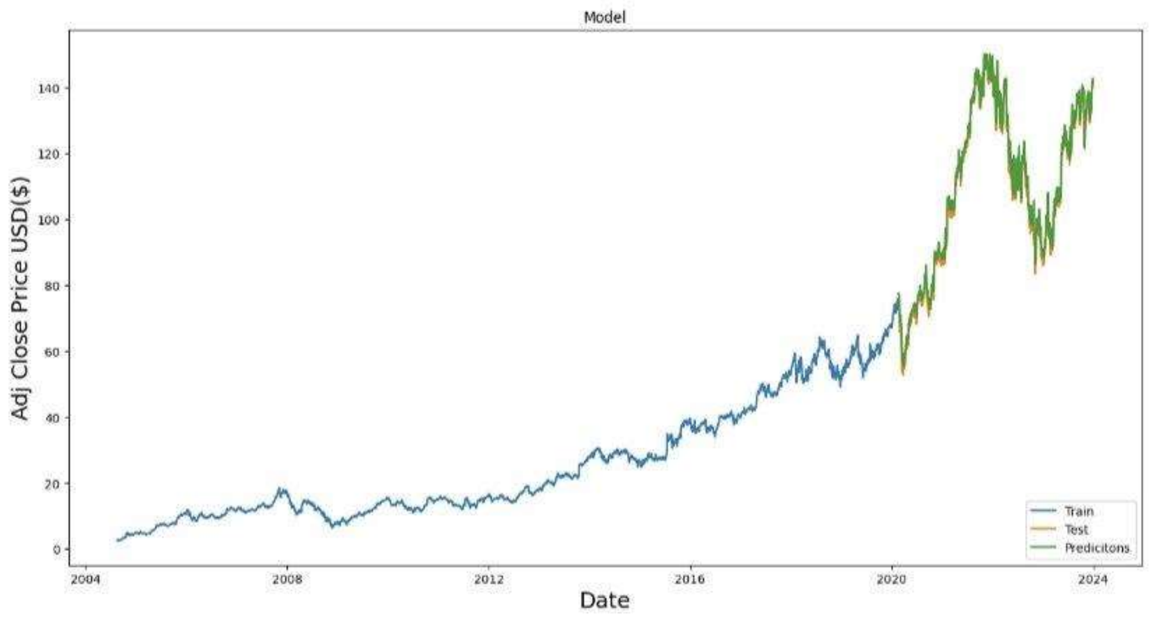
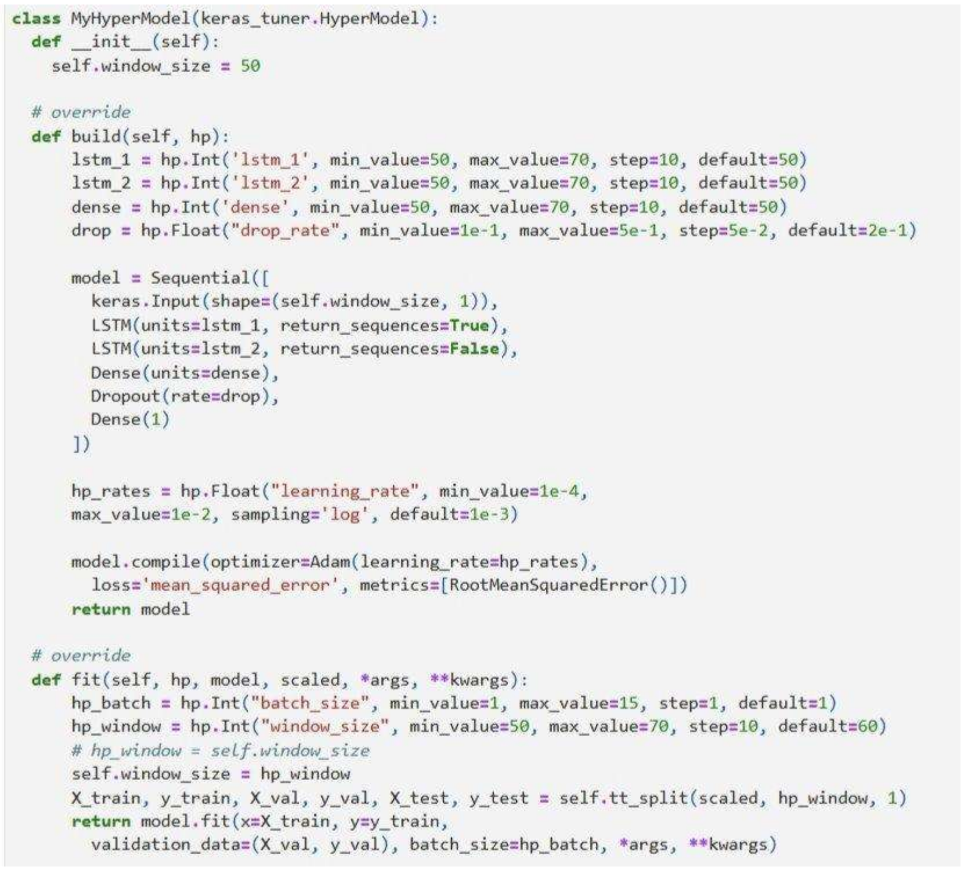


Figure: Plot of predictions with actual data

Here we can see that the predictions are indeed very close to the test data values. The predictions are able to capture the noticeable spikes in the price trends as well. However, the almost perfect fitting implies a certain level of overfitting. This is addressed in the hypertuning step.

3.4.4. Hypertuning



```python
class MyHyperModel(keras_tuner.HyperModel):
  def __init__(self):
    self.window_size = 50

  # override
  def build(self, hp):
      lstm_1 = hp.Int('lstm_1', min_value=50, max_value=70, step=10, default=50)
      lstm_2 = hp.Int('lstm_2', min_value=50, max_value=70, step=10, default=50)
      dense = hp.Int('dense', min_value=50, max_value=70, step=10, default=50)
      drop = hp.Float("drop_rate", min_value=1e-1, max_value=5e-1, step=5e-2, default=2e-1)

      model = Sequential([
        keras.Input(shape=(self.window_size, 1)),
        LSTM(units=lstm_1, return_sequences=True),
        LSTM(units=lstm_2, return_sequences=False),
        Dense(units=dense),
        Dropout(rate=drop),
        Dense(1)
      ])

      hp_rates = hp.Float("learning_rate", min_value=1e-4,
      max_value=1e-2, sampling='log', default=1e-3)

      model.compile(optimizer=Adam(learning_rate=hp_rates),
        loss='mean_squared_error', metrics=[RootMeanSquaredError()])
      return model

  # override
  def fit(self, hp, model, scaled, *args, **kwargs):
      hp_batch = hp.Int("batch_size", min_value=1, max_value=15, step=1, default=1)
      hp_window = hp.Int("window_size", min_value=50, max_value=70, step=10, default=60)
      # hp_window = self.window_size
      self.window_size = hp_window
      X_train, y_train, X_val, y_val, X_test, y_test = self.tt_split(scaled, hp_window, 1)
      return model.fit(x=X_train, y=y_train,
        validation_data=(X_val, y_val), batch_size=hp_batch, *args, **kwargs)
```

Figure: Code redefining Methods from keras_tuner's class

Using the Keras_tuner library, we attempted to perform a grid search.

First we need to define the `build` and `fit`methods from the parent class `keras_tuner.Hypermodel`. We define those methods in our child class as `MyHyperModel`.

These methods include lines of code that creates an array of choices that assist in grid search.

For example in the line:

lstm_1 = hp.Int('lstm_1', min_value=50, max_value=70, step=10, default=50)

We allow for the number of units in the first LSTM layer chosen to include the values [50, 60, 70] units.

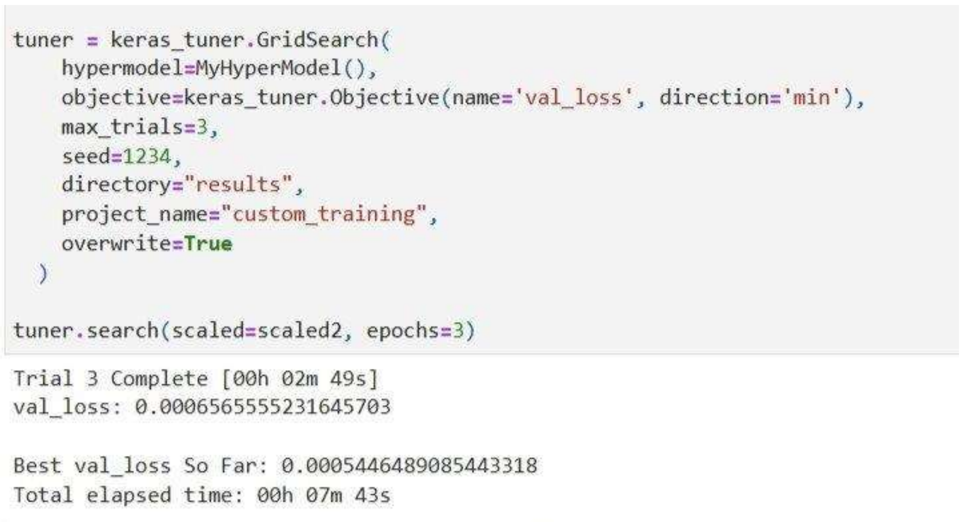
In addition, we also added a Dropout Layer which randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by 1 / (1 - dropout rate) such that the sum over all inputs is unchanged.

The following list shows the hyperparameters we wish to tune:

- **lstm_1**: No. of neurons in first LSTM layer
- **lstm_2**: No. of neurons in second LSTM layer
- **dense**: No. of neurons in dense layer
- **drop**: rate of dropout in Dropout layer
- **learning_rate**: momentum coefficient in Adam optimizer
- **batch_size**: Number of samples propagated
- **window_size**: No. of days used to predict next day's price

3.4.5. Hypertuned Results

The Grid Search will cover all combinations of the hyperparameters and by pursuing on the objective of minimizing the validation loss value, we have

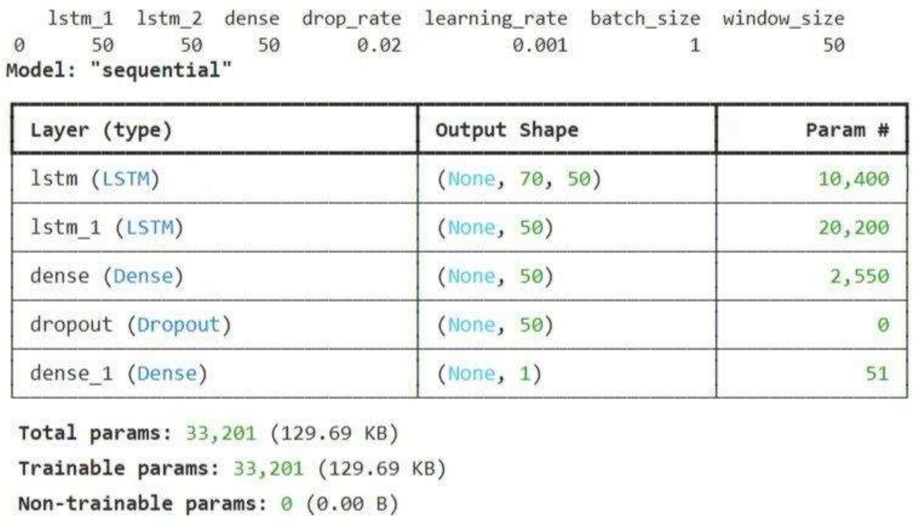


```
tuner = keras_tuner.GridSearch(
    hypermodel=MyHyperModel(),
    objective=keras_tuner.Objective(name='val_loss', direction='min'),
    max_trials=3,
    seed=1234,
    directory="results",
    project_name="custom_training",
    overwrite=True
  )

tuner.search(scaled=scaled2, epochs=3)
```

```
Trial 3 Complete [00h 02m 49s]
val_loss: 0.0006565555231645703

Best val_loss So Far: 0.0005446489085443318
Total elapsed time: 00h 07m 43s
```

Figure: Executing GridSearch

Where the lowest validation loss value is approximately **0.00065655**.



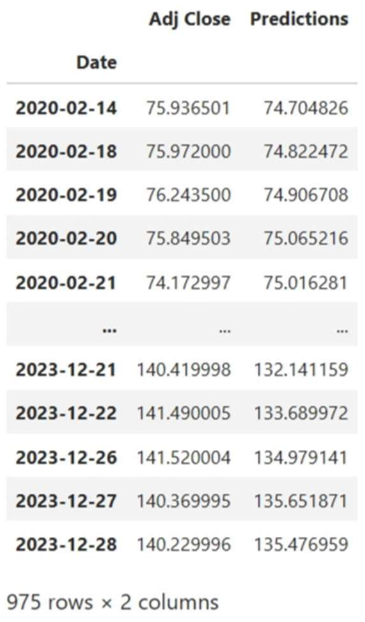
```
   lstm_1  lstm_2  dense  drop_rate  learning_rate  batch_size  window_size
0      50      50     50       0.02          0.001           1           50
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━┓
┃ Layer (type)                            ┃ Output Shape               ┃        Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━┩
│ lstm (LSTM)                             │ (None, 70, 50)             │         10,400 │
├─────────────────────────────────────────┼────────────────────────────┼────────────────┤
│ lstm_1 (LSTM)                           │ (None, 50)                 │         20,200 │
├─────────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dense (Dense)                           │ (None, 50)                 │          2,550 │
├─────────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dropout (Dropout)                       │ (None, 50)                 │              0 │
├─────────────────────────────────────────┼────────────────────────────┼────────────────┤
│ dense_1 (Dense)                         │ (None, 1)                  │             51 │
└─────────────────────────────────────────┴────────────────────────────┴────────────────┘
 Total params: 33,201 (129.69 KB)
 Trainable params: 33,201 (129.69 KB)
 Non-trainable params: 0 (0.00 B)
```

Figure: Best Hyperparameters and Model

And from this figure, the hyperparameters and model that gave the lowest validation loss is found.

We then repeat the following with the hypertuned model:

1. Predict the results with the hypertuned model



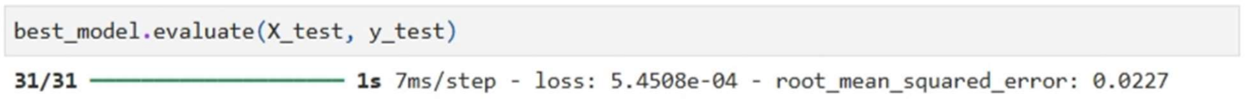
Date	Adj Close	Predictions
2020-02-14	75.936501	74.704826
2020-02-18	75.972000	74.822472
2020-02-19	76.243500	74.906708
2020-02-20	75.849503	75.065216
2020-02-21	74.172997	75.016281
...	...	...
2023-12-21	140.419998	132.141159
2023-12-22	141.490005	133.689972
2023-12-26	141.520004	134.979141
2023-12-27	140.369995	135.651871
2023-12-28	140.229996	135.476959

975 rows × 2 columns

Figure: Hypertuned model's Predictions

And again we see that the results are close albeit not as much as the initial results.

2. Evaluate the loss value on the test data



```
best_model.evaluate(X_test, y_test)
31/31 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 5.4508e-04 - root_mean_squared_error: 0.0227
```

Figure: Loss values on test data by Hypertuned model

And again we see that the loss value over the test data is low but not as much as the initial results.

3. Plot the hypertuned model's predictions with actual data

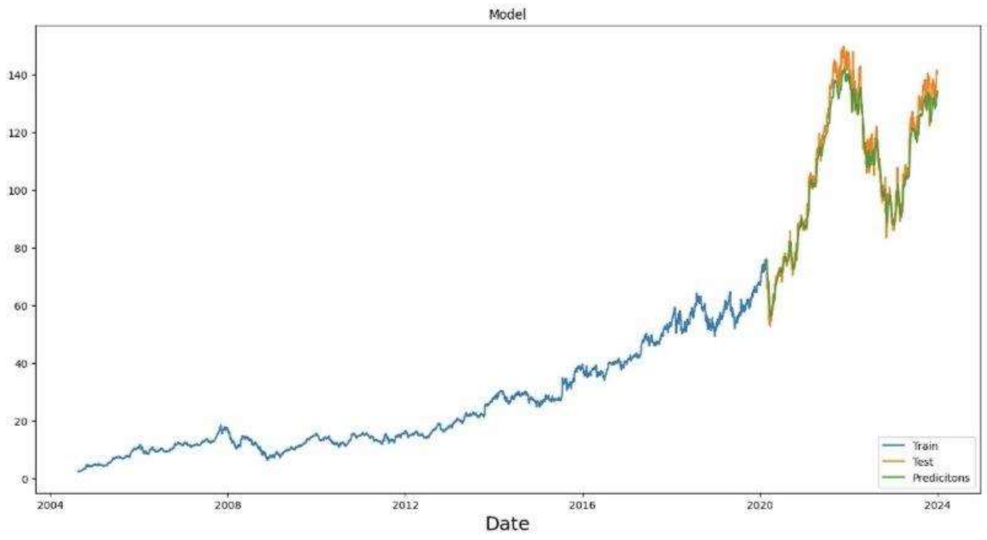


Figure: Figure: Plot of Hypertuned predictions with actual data

Here we can see that the predictions are still close, able to capture spikes in trends and are not as overfitting as the initial model.

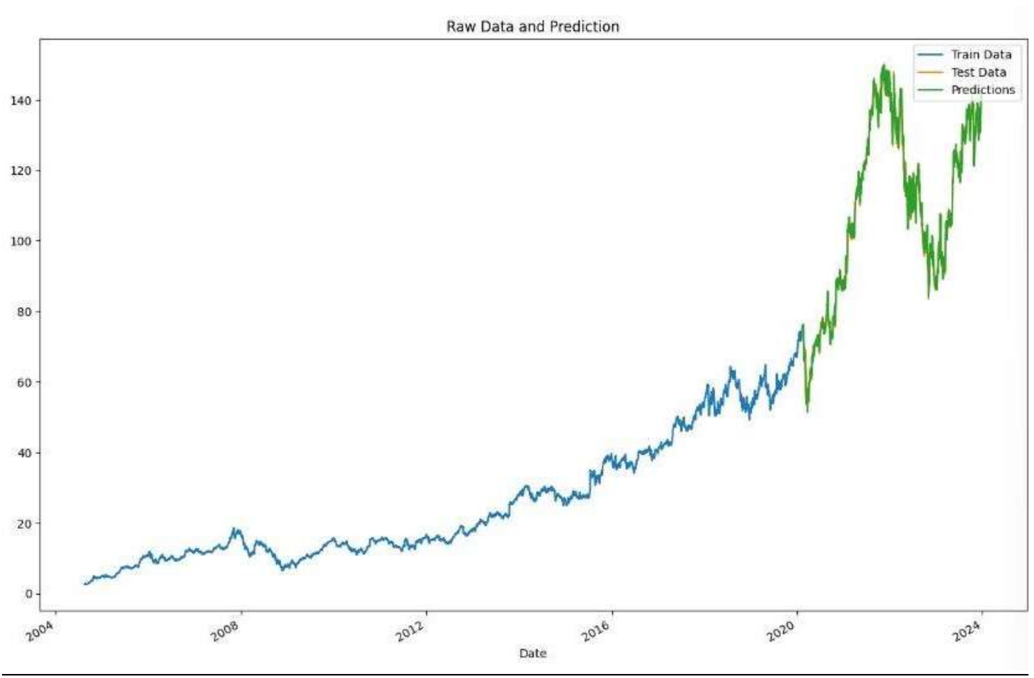
4. Conclusion

4.1. Results Analysis

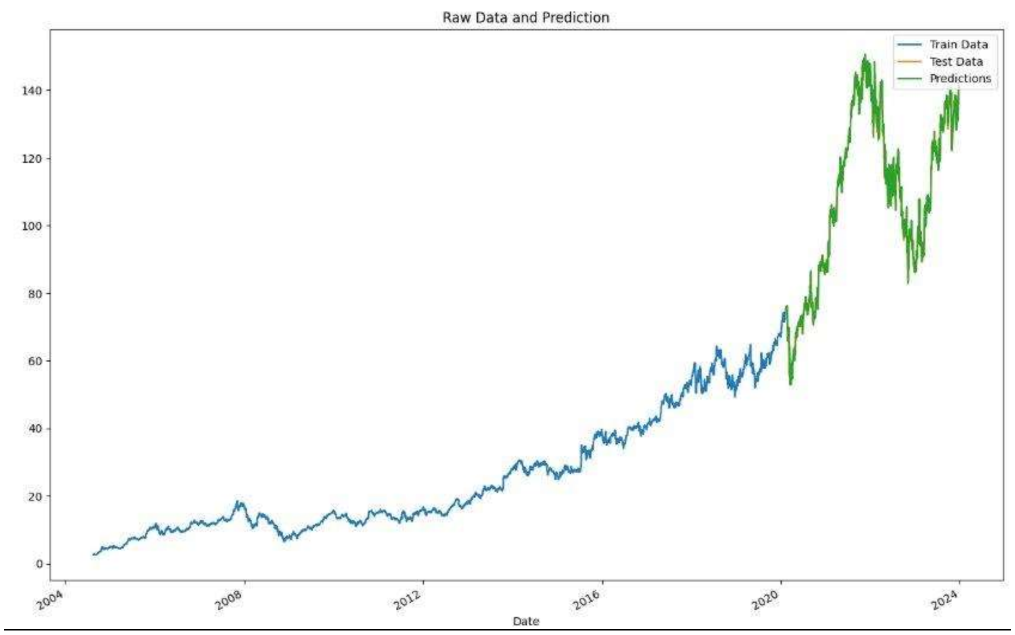
Linear Regression

There is multicollinearity in the data. This may be a reason why linear regression generally overfits, as no multicollinearity is one of the assumptions that linear regression has.

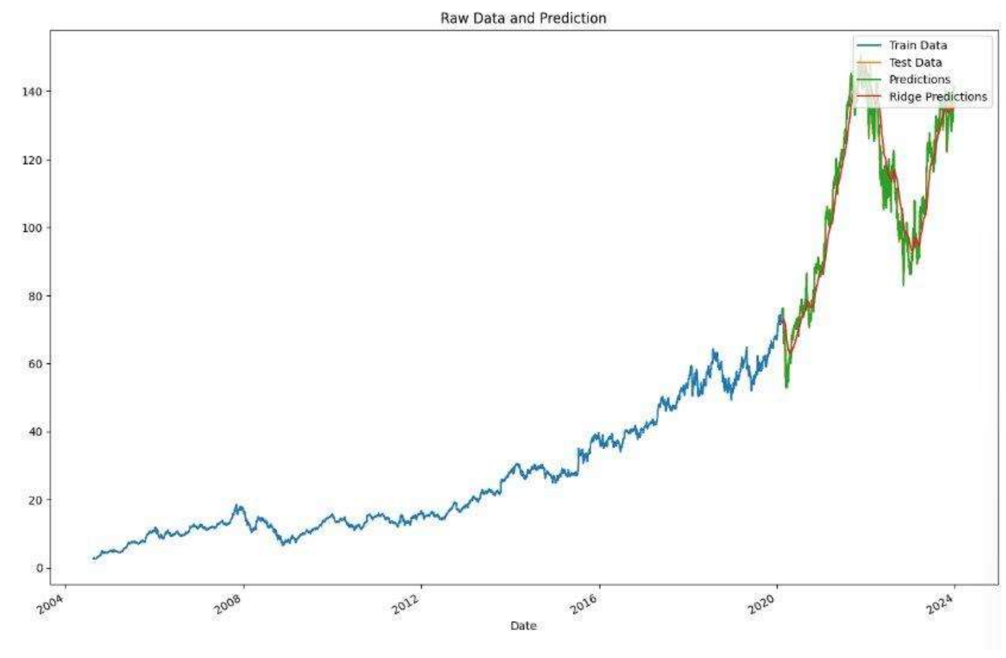
Vanilla linear regression is highly accurate but overfitting.



Sliding window linear regression is also highly accurate but overfitting.



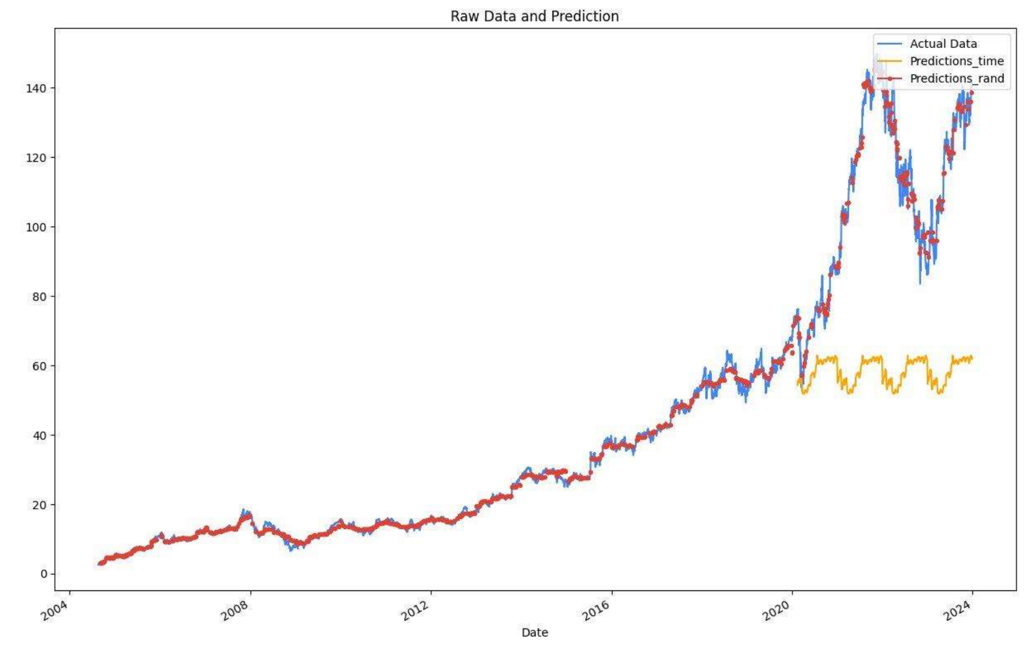
Ridge regression has little overfitting comparatively. It has reduced variance but increased bias because of the variance-bias tradeoff.



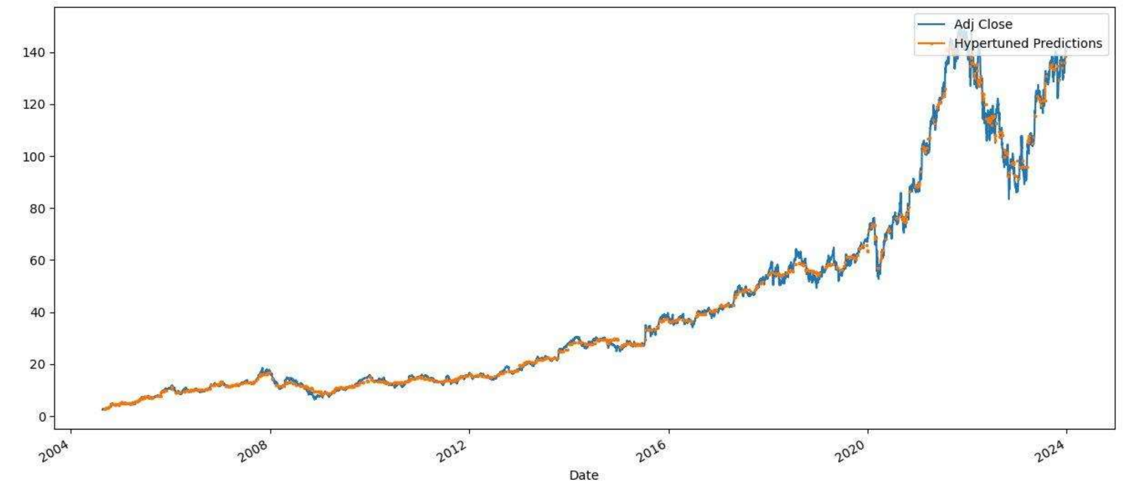
We choose ridge regression out of the three linear regression models to compare against the other machine learning models.

XGBoost

We see that time series split XG boost has a very inaccurate prediction, with a high RMSE of 56.3. Random split XG boost has a much more accurate prediction, with RMSE of 2.03.

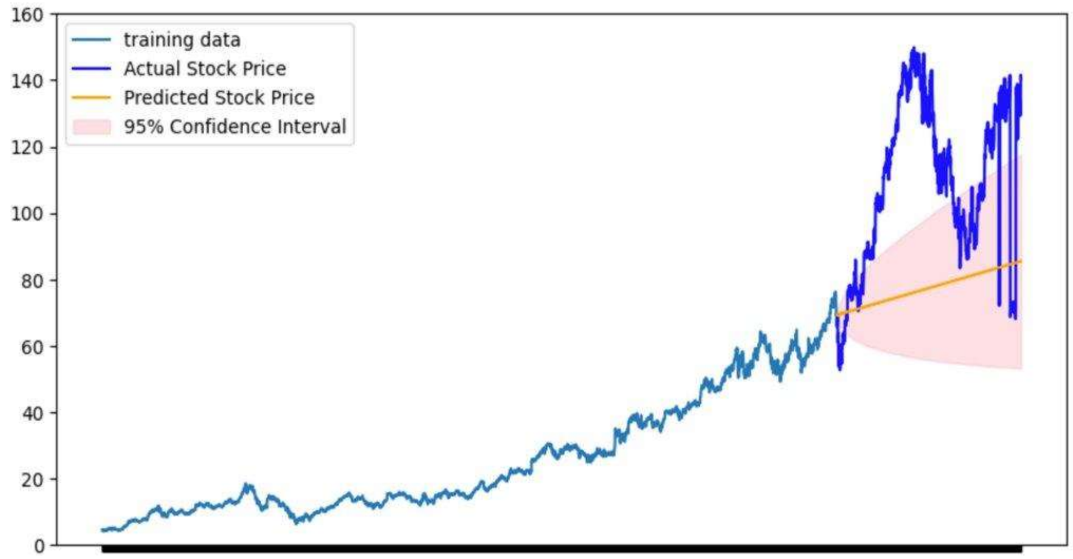


Random split XG boost with hypertuning is the most accurate and has the smallest RMSE of 1.59.



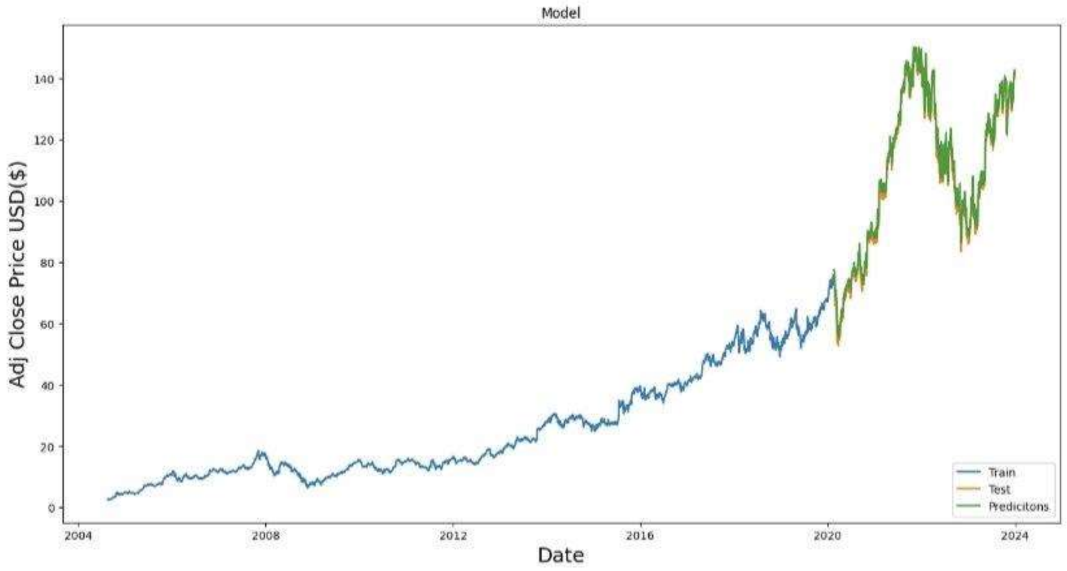
SARIMA

The SARIMA model underfits greatly, with prediction being a linear graph. The model correctly predicts the general rising trend of the price of the stock, but does not predict the spikes of the stock prices in the 95% confidence interval.

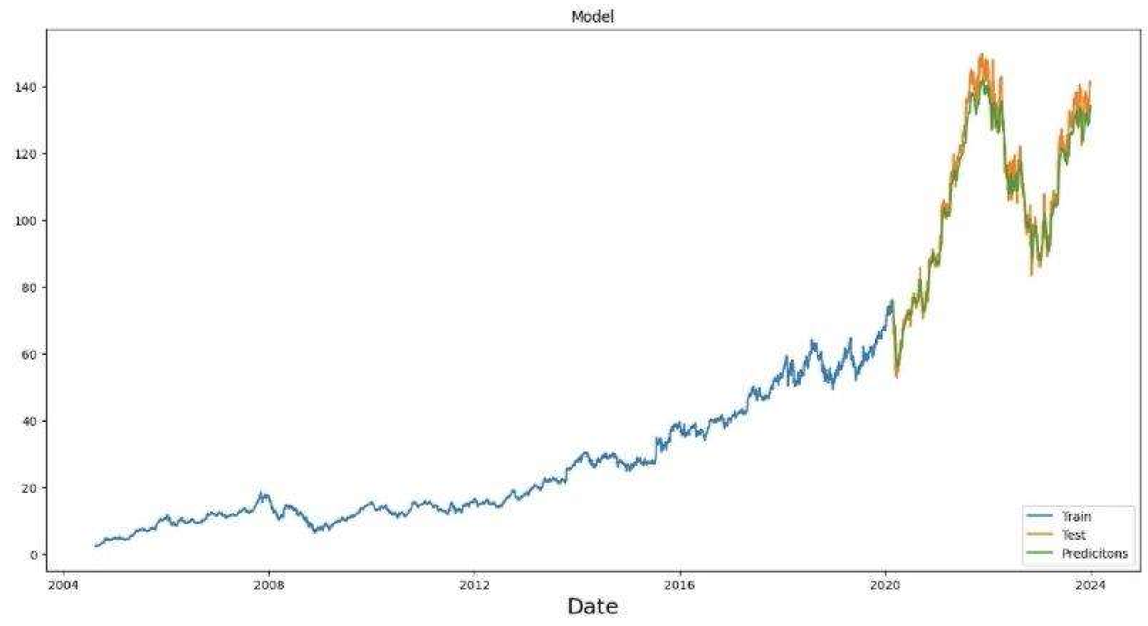


LSTM

The sliding window LSTM has a high accuracy, but overfits greatly.



The hypertuned sliding window LSTM has less overfitting but is still accurate.



Comparing all the models, the hypertuned LSTM seems to be the best model in predicting future stock price based on accuracy and fit, as it can predict anomalies like the general spikes in prices, and has the best fit among all the models.

4.2. Future Development

We aim to expand our project to test models to other stocks, firstly within the tech industry that Google is in, and then extending to stocks in other industries. With more stocks, we can compare the accuracy and fit of the LSTM model for different stocks and gather insights.

We also aim to automate periodic data scraping, so that users will be able to predict future stock prices in real time.