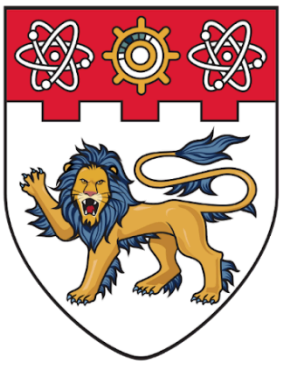


NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

NANYANG TECHNOLOGICAL UNIVERSITY

SCSE23-0954: ENHANCING STOCK PRICE PREDICTION USING MACHINE LEARNING TECHNIQUES: A COMPARATIVE ANALYSIS OF ARIMA, LSTM WITH SENTIMENT ANALYSIS, TRANSFORMERS, AND GPT-3

Agarwal Anusha

(U2023105H)

Project Supervisor: A/P Long Cheng

Examiner: A/P Qian Kemao

College of Computing and Data Science

Academic Year 2023/2024

NANYANG TECHNOLOGICAL UNIVERSITY

SCSE23-0954: ENHANCING STOCK PRICE PREDICTION USING MACHINE LEARNING TECHNIQUES: A COMPARATIVE ANALYSIS OF ARIMA, LSTM WITH SENTIMENT ANALYSIS, TRANSFORMERS, AND GPT-3

Submitted in Partial Fulfillment of Requirements

for the Degree of Bachelor of Engineering in Computer Science

of the Nanyang Technological University

by

Agarwal Anusha

(U2023105H)

College of Computing and Data Science

Academic Year 2023/2024

Abstract

This project aims to explore the use of various machine learning techniques for predicting stock prices, focusing on Apple Inc. (AAPL) stock data from 2015 to 2019. Traditional models like ARIMA are compared with more advanced architectures, including Long Short-Term Memory (LSTM) networks and Transformer models. The study also investigates the impact of incorporating sentiment analysis, using Twitter data within the LSTM model. This is used to determine whether sentiment features improve predictive performance. Additionally, GPT-3 is studied to assess its usage in predicting stock prices based on historical data.

The models are evaluated using key performance metrics such as Mean Squared Error (MSE), Mean Absolute Percentage Error (MAPE), and R^2. Results indicate that advanced models like LSTM and Transformer significantly outperform traditional approaches like ARIMA. However, the incorporation of sentiment analysis did not lead to any substantial improvement in the performance of the LSTM model. GPT-3 demonstrated strong performance in predicting stock prices, even when tested on recent stock data indicating its flexibility to adapt to financial analysis even though its primary motive is to perform natural language processing. This project concludes that Transformer-based models, particularly those incorporating multiple features, offer the best performance for stock price prediction.

Acknowledgement

I would like to express my deepest gratitude to Prof. Long Cheng for giving me the opportunity to work on this project. Under his invaluable guidance, I have strengthened my concepts of deep neural networks and gained a deeper appreciation for their application in the real-world stock price analysis. His continuous support, insights and encouragement were instrumental in the successful completion of this project.

I would also like to thank my peers for their feedback and support, which has contributed to the progress of my project. Lastly, I would like to express my gratitude to my friends and family who have supported me throughout the journey with constant encouragement during this challenging yet rewarding journey.

Table Of Content

Abstract.....i
Acknowledgement.....ii
Table Of Content.....iii
List Of Figures.....vi
1. Introduction.....1
2. Literature Review.....1
3. Research Gap.....2
4. Objective.....3
4.1. Evaluate Predictive Performance.....3
4.2. Incorporate Sentiment Analysis.....3
4.3. Comparative Analysis.....3
5. Exploratory Data Analysis.....3
5.1. Data Collection.....3
5.2. Data Processing.....4
5.2.1. Checking for missing values.....4
5.2.2. Linear Interpolation.....4
5.2.3. Visualizing the interpolated stock prices.....4
5.2.4. Bollinger Band Visualization.....5
6. ARIMA Model.....6
6.1. Data Split and Normalization.....6
6.2. Stationarity Testing.....7
6.3. First Order Differencing.....8
6.4. Autocorrelation & Partial Autocorrelation Plot Analysis.....9
6.5. Model Selection using Auto-Arima.....10
6.6. ARIMA Model fitting.....11
6.7. Forecasting on the Test Set.....12
6.8. Model Performance - Metrics and Visualization.....12
6.9. ARIMA Performance Visualization.....13
7. LSTM.....13
7.1. Data Split for and Data Normalization.....13
7.2. Sequence Creation for LSTM Model.....15
7.3. Hyperparameters Tuning - Grid Search.....15
7.4. LSTM Architecture.....17
7.4.1. Model Structure.....17
7.4.2. LSTM Gates and Equations:.....18
7.4.3. Early Stopping and Model Training.....19
7.4.4. Training Loss vs Validation Loss.....20

7.4.5. Performance Evaluation..... 20
7.4.6. Model Prediction Performance Visualization..... 21
8. LSTM With Sentiment Analysis..... 22
8.1. Sentiment Analysis on Apple Data (2015-2019)..... 22
8.1.1. Extraction of Apple Data from the Twitter Dataset..... 22
8.1.2. AAPL Twitter Data Cleaning and Preprocessing..... 23
8.1.3. Word Cloud Visualization..... 23
8.1.4. Sentiment Analysis..... 24
8.1.5. Aggregating Sentiment Scores by data..... 24
8.1.6. Daily Sentiment Trend Visualization..... 25
8.2. Merge Adjusted Close Price with Daily Sentiment Scores..... 25
8.3. Data Split and Data Normalization..... 25
8.4. Correlation Between Adjusted Close Price and Sentiment Score..... 26
8.5. Sequence Creation for LSTM Model..... 27
8.6. Hyperparameter Tuning - Grid Search..... 27
8.7. LSTM (with Sentiment Analysis) Architecture..... 29
8.7.1. Model Structure..... 29
8.7.2. Early Stopping and Model Training..... 30
8.7.3. Training Loss vs Validation Loss..... 31
8.7.4. Performance Metrics..... 31
8.7.5. Model Prediction Performance Visualization..... 32
8.8. Insight on hypothesis..... 33
9. Transformer..... 34
9.1. Feature Preparation..... 34
9.2. Feature Selection..... 36
9.3. Data Split and Data Normalization..... 37
9.4. Sequence Creation for Transformer Model..... 37
9.5. Hyperparameter Tuning (Grid Search and Bayesian Optimization)..... 38
9.6. Transformer Architecture..... 39
9.6.1. Optimum Parameters..... 39
9.6.2. Layer-by-Layer Explanation..... 39
9.6.3. Position Encoding..... 40
9.6.4. Transformer Encoder..... 41
9.6.5. Transformer Model Creation..... 41
9.6.6. Early Stopping and Model Training..... 42
9.6.7. Training Loss vs Validation Loss..... 42
9.6.8. Performance Metrics..... 42
9.6.9. Model Prediction Performance Visualization..... 43
9.7. Additional Analysis..... 44
9.7.1. Transformer Model on Adjusted Close Price..... 44
10. GPT-3..... 45

10.1. Data Retrieval and Preparation....46
10.2. GPT-3 Prompt Creation....46
10.3. Model Prediction Approach....47
10.3.1. Model Used....48
10.3.2. Chat Messages Setup....48
10.3.3. Response Limitation....48
10.3.4. Temperature Setting....48
10.4. Performance Metrics....48
10.5. Model Prediction Performance Visualization....48
10.6. Additional Analysis - Incorporating Moving Averages and RSI with GPT-3....49
10.7. Additional Analysis: Evaluating GPT-3 on Recent Data (2024-01-01 to 2024-10-01)....50
11. Comparative Model Performance....51
11.1. MSE....51
11.2. MAPE....51
11.3. R^2 Score....51
12. Conclusion....52
13. Future Work....53
14. References....54
15. Appendix....54
Project Structure....54

List Of Figures

Figure 5.1: Historical Stock Price data..... 3
Figure 5.2: Check for missing Values..... 4
Figure 5.3: Apply Linear Interpolation..... 4
Figure 5.4: Analyze the cleaned dataset..... 4
Figure 5.5: AAPL Stock Prices: Interpolated Candlestick Chart..... 5
Figure 5.6: AAPL Bollinger Band..... 5
Figure 6.1: AAPL Adjusted Close Price - Data Split..... 6
Figure 6.2: check_stationarity function implementation..... 7
Figure 6.3: Rolling Mean and Std Dev..... 7
Figure 6.4: ADF Statistics..... 8
Figure 6.5: First Order Differencing..... 8
Figure 6.6: Rolling Statistics after First Order Differencing..... 8
Figure 6.7: ADF Statistics after First Order Differencing..... 9
Figure 6.8: ACF and PACF Plot..... 9
Figure 6.9: Fit auto_arima to Differenced Data..... 10
Figure 6.10: Auto ARIMA model summary..... 10
Figure 6.11: Fit Arima Model on optimal parameters..... 11
Figure 6.12: ARIMA Model Diagnostic Plots..... 11
Figure 6.13: Forecasting on Test Set..... 12
Figure 6.14: ARIMA Performance Metrics..... 12
Figure 6.15: ARIMA Prediction vs Actual Data..... 13
Figure 7.1: Input Data Split (75:15:15)..... 14
Figure 7.2: Data Scaling..... 14
Figure 7.3: creat_sequence function for LSTM model..... 15
Figure 7.4: Split Data Shape Analysis..... 15
Figure 7.5: Training Loss curve across different num_units and batch Size..... 16
Figure 7.6: Hyperparameter Tuning - Best model architectures..... 16
Figure 7.7: create_lstm_model function..... 17
Figure 7.8: LSTM model summary..... 17
Figure 7.9: LSTM Model Architecture..... 17
Figure 7.10: Single LSTM cell..... 18
Figure 7.11: train_lstm_model function..... 19
Figure 7.12: LSTM Model: Training vs Validation Loss across epochs..... 20
Figure 7.13: LSTM model predictions..... 20
Figure 7.14: Rescale predictions to original scale..... 20
Figure 7.15: LSTM model performance metrics..... 21
Figure 7.16: Training: Actual vs Predicted 21

Figure 7.17: Validation: Actual vs Predicted..... 21
Figure 7.18: Testing: Actual vs Predicted..... 21
Figure 7.19: Actual vs Predicted Adj Close Price..... 22
Figure 8.1: Extract AAPL tweets..... 22
Figure 8.2: Twitter Data Cleaning..... 23
Figure 8.3: AAPL tweets preprocessing..... 23
Figure 8.4: AAPL tweets WordCloud..... 24
Figure 8.5: Sentiment Analysis on AAPL tweets..... 24
Figure 8.6: Aggregate Sentiment scores by date..... 24
Figure 8.7: Daily Average Sentiment Scores over Time..... 25
Figure 8.8: Input Data Split (75:15:15)..... 25
Figure 8.9: Data Scaling..... 26
Figure 8.10: Correlation between Adj Close Price and Sentiment Score..... 26
Figure 8.11: create_sequence for LSTM+Sentiment model..... 27
Figure 8.12: Analyze Input Data Shape..... 27
Figure 8.13: Training Loss curve across different num_units and batch Size..... 28
Figure 8.14: Hyperparameter Tuning: Best Model Architectures..... 28
Figure 8.15: create_lstm_model function for LSTM+Sentiment model..... 29
Figure 8.16: LSTM + Sentiment model summary..... 29
Figure 8.17: train_lstm_model function for LSTM+Sentiment model..... 30
Figure 8.18: Training vs Validation Loss curve across epochs..... 31
Figure 8.19: Predictions on LSTM+Sentiment Model..... 31
Figure 8.20: Rescale Predicted Values to Original Scale..... 32
Figure 8.21: LSTM+Sentiment Model Performance Metrics..... 32
Figure 8.22: Training: Actual vs Prediction 32
Figure 8.23: Validation: Actual vs Prediction..... 32
Figure 8.24: Testing: Actual vs Prediction..... 33
Figure 8.25: Actual vs Prediction Adj Close Price..... 33
Figure 9.1: AAPL Price Width..... 34
Figure 9.2: AAPL Percentage Price Change..... 35
Figure 9.3: AAPL Relative Strength Index (RSI)..... 35
Figure 9.4: AAPL Rate of Change..... 36
Figure 9.5: Feature Selection for Transformer..... 36
Figure 9.6: Input Data Split (75:15:15)..... 37
Figure 9.7: Input Data Scaling for Transformer..... 37
Figure 9.8: create_sequence function for Transformer..... 37
Figure 9.9: Analyze input shape for Transformer..... 37
Figure 9.10: Hyperparameter Tuning: Bayesian Optimization..... 38
Figure 9.11: Hyperparameter tuning: Best model architecture for Transformer..... 38
Figure 9.12: Transformer Architecture [3]..... 39
Figure 9.13: positional_encoding function..... 40

Figure 9.14: transformer_encoder function....41
Figure 9.15: build_transformer_model function....41
Figure 9.16: Compile and Train Transformer Model....42
Figure 9.17: Training vs Validation Loss across epochs....42
Figure 9.18: Predictions using transformer model....42
Figure 9.19: Transformer Model Performance Metrics....43
Figure 9.20: Training: Predictions vs Actual 43
Figure 9.21: Validation: Prediction vs Actual....43
Figure 9.22: Testing: Prediction vs Actual....43
Figure 9.23: Prediction vs Actual Adj Close Price....44
Figure 9.24: Top Architectures: Transformer Model (1 feature)....44
Figure 9.25: Performance Metrics: Transformer Model (1 feature)....45
Figure 9.26: Prediction vs Actual Adj Close Price: Transformer Model (1 feature)....45
Figure 10.1: Input Data Preparation for GPT-3....46
Figure 10.2: create_gpt_prompt function....46
Figure 10.3: GPT-3 Prompt Example....47
Figure 10.4: GPT-3 Prediction Setup....47
Figure 10.5: GPT-3 Performance Metrics....48
Figure 10.6: GPT-3 Performance Visualization: Actual vs Predicted Adj Close....49
Figure 10.7: Additional Features MA and RSI....49
Figure 10.8: GPT-3 with additional features - Performance Metrics....50
Figure 10.9: GPT-3 with 2024 data - Performance Metrics....50
Figure 11.1: Model Comparison:MSE 51
Figure 11.2: Model Comparison: MAPE....51
Figure 11.3: Model Comparison: R^2....52

1. Introduction

The stock market is a complex and highly volatile system and has a significant impact on both individuals and the overall economy. Accurate anticipation of market trends is therefore rather crucial for informed decision making and risk management. However, stock price prediction is challenging due to the dynamic nature of the market, influenced not only by economic indicators, but also geopolitical scenarios, and market sentiment.

Traditional statistical models like ARIMA have long been used for time series forecasting, but the need for capturing complex stock market patterns has led to the rise of advanced models such as LSTM networks and Transformer models. Moreover, GPT, while primarily build for natural language processing, may potentially be adopted to perform stock price forecasting.

This project explores and compares ML techniques like ARIMA, LSTM (with and without sentiment analysis from Twitter), Transformers, and GPT-3, to predict Apple (AAPL) stock prices from 2015 to 2019. It also assesses the impact of sentiment analysis on LSTM performance and identifies the best-performing model.

2. Literature Review

In recent years, with the advance in deep machine learning models, it has been extensively used to make data driven decisions within the financial sector, particularly in predicting stock prices. Traditional statistical methods such as **ARIMA** (AutoRegressive Integrated Moving Average) have been used for time series analysis for a very long time. Studies like Box and Jenkins (1970) [1] have laid the foundation for this model. ARIMA models rely heavily on stationarity and linear relationship between variables and are unable to completely capture the complexities underlying financial data.

More recently, Artificial Neural Networks are being explored to make such predictions and capture underlying complexities in the non-linear stock market time series data. Models such as **Long Short-Term Memory (LSTM)** networks, have gained traction because of their ability to model sequential data and capture long-term dependencies. Studies have shown that LSTM models outperform traditional stock prediction methods by effectively capturing the temporal

relationships within the data. Research by Fischer and Krauss (2018) [2] demonstrates the effectiveness of LSTM networks in predicting stock prices.

Additionally, **Transformers**, which are heavily used in natural language processing tasks, also have utility in forecasting time series data. Studies have shown transformers have superior performance to traditional RNNs due to their self attention mechanisms that allow transformers to weigh importance of different time steps' effectively, and capture dependencies over different time steps [3].

Beyond models based only on price and economic indicators, **sentiment analysis** has now emerged as a valuable feature in financial forecasting. Research has been conducted to explore relationships between public sentiment gathered from social media platforms such as Twitter (now X) and stock price movement. [4] While sentiment analysis has the potential to enhance stock price prediction models, its effectiveness may vary depending on the dataset and time period used, as seen in recent studies. Recently, research has also been conducted to explore the potential of large language models, such as **ChatGPT**, in forecasting stock price movements, demonstrating promising results [5].

3. Research Gap

While a significant amount of research has been done on using traditional models such as ARIMA and Machine Learning models such as LSTM, Transformers for forecasting stock price, there still remains a gap in **integrating sentiment analysis** within these models as an additional feature. Most studies are either focussed entirely on stock price prediction using historical data or on sentiment analysis separately. Therefore, there is a need for exploring an integrated approach that incorporates external sentiment factors as an additional feature to advanced ML techniques. In addition to bridging this gap, this project also aims to explore performance on **transformers on financial time series** and do a **comparative analysis** as to how well each of these models captures the complexities of financial data. Despite the advancements in LLM models like ChatGPT, limited research has been done on the model's true predictive capability for stock prices. In this project, we will also explore the **GPT-3 model’s performance** and prediction power given historical stock price data.

4. Objective

The primary objective of this project is to do a **comparative analysis** on the performance of various machine learning models - ARIMA, LSTM, LSTM + Sentiment Analysis, Transformer models, and GPT-3 models used to predict stock prices. We will specifically focus on AAPL stock data from 2015 to 2019. The goals of this project are -

4.1. Evaluate Predictive Performance

We will analyze the model performance using metrics such as Mean Square Error (MSE), Mean Absolute Percentage Error (MAPE) and R^2 score to determine accuracy and effectiveness of predictions.

4.2. Incorporate Sentiment Analysis

We will explore whether integrating sentiment analysis from Twitter can enhance the predictive power of the LSTM model.

4.3. Comparative Analysis

Investigate models predictive performance using aforementioned metrics and determine which model offers the best performance.

This report aims to assess model performance and potential application in real world scenarios such as trading strategies.

5. Exploratory Data Analysis

5.1. Data Collection

5.1.1. Download historical stock price data for Apple using '*yf.download*' for the past 9 years from 2015-01-01 to 2023-12-31.

```
# Download the historical stock price data for Apple
stock_data = yf.download('AAPL','2015-01-01','2023-12-31')
stock_data
```

```
[*********************100%***********************]  1 of 1 completed
```

Date	Open	High	Low	Close	Adj Close	Volume
2015-01-02	27.847500	27.860001	26.837500	27.332500	24.373959	212818400
2015-01-05	27.072500	27.162500	26.352501	26.562500	23.687305	257142000
2015-01-06	26.635000	26.857500	26.157499	26.565001	23.689533	263188400
2015-01-07	26.799999	27.049999	26.674999	26.937500	24.021715	160423600
2015-01-08	27.307501	28.037500	27.174999	27.972500	24.944685	237458000
...	...	...	...	...	...	...
2023-12-22	195.179993	195.410004	192.970001	193.600006	192.868149	37122800
2023-12-26	193.610001	193.889999	192.830002	193.050003	192.320221	28919300
2023-12-27	192.490005	193.500000	191.089996	193.149994	192.419830	48087700
2023-12-28	194.139999	194.660004	193.169998	193.580002	192.848206	34049900
2023-12-29	193.899994	194.399994	191.729996	192.529999	191.802185	42628800

2264 rows × 6 columns

Figure 5.1: Historical Stock Price data

5.2. Data Processing

5.2.1. Checking for missing values

There is no null value for the historical data of corresponding dates.

```
# Check for missing values
print(stock_data.isnull().sum())
```

```
Open         0
High         0
Low          0
Close        0
Adj Close    0
Volume       0
dtype: int64
```

Figure 5.2: Check for missing Values

5.2.2. Linear Interpolation

The **interpolate** method with the **linear** parameter was used to fill in missing dates, such as weekends, market holidays, or other instances where data was unavailable. This linear interpolation ensures that the gaps in the time series are smoothly filled, creating a continuous dataset.

```
# Apply linear interpolation to fill missing values (if any)

# Create a full date range for the given period (including weekends/holidays)
full_date_range = pd.date_range(start=start_date, end=end_date, freq='D')
apple_stock_data_full = apple_stock_data.reindex(full_date_range)

# Interpolate the missing data using linear method
apple_stock_data_interpolated = apple_stock_data_full.interpolate(method='linear')
apple_stock_data_cleaned = apple_stock_data_interpolated.dropna()
```

Figure 5.3: Apply Linear Interpolation

	Open	High	Low	Close	Adj Close	Volume
2015-01-02	27.847500	27.860001	26.837500	27.332500	24.373959	2.128184e+08
2015-01-03	27.589167	27.627501	26.675833	27.075834	24.145074	2.275929e+08
2015-01-04	27.330833	27.395000	26.514167	26.819167	23.916189	2.423675e+08
2015-01-05	27.072500	27.162500	26.352501	26.562500	23.687304	2.571420e+08
2015-01-06	26.635000	26.857500	26.157499	26.565001	23.689537	2.631884e+08

Figure 5.4: Analyze the cleaned dataset

After applying linear interpolation, all missing values were successfully filled, resulting in a complete and clean dataset. This step is crucial for maintaining the continuity of the time series, which is essential for accurate stock price prediction.

5.2.3. Visualizing the interpolated stock prices

Plot the interpolated stock prices along with the original prices.

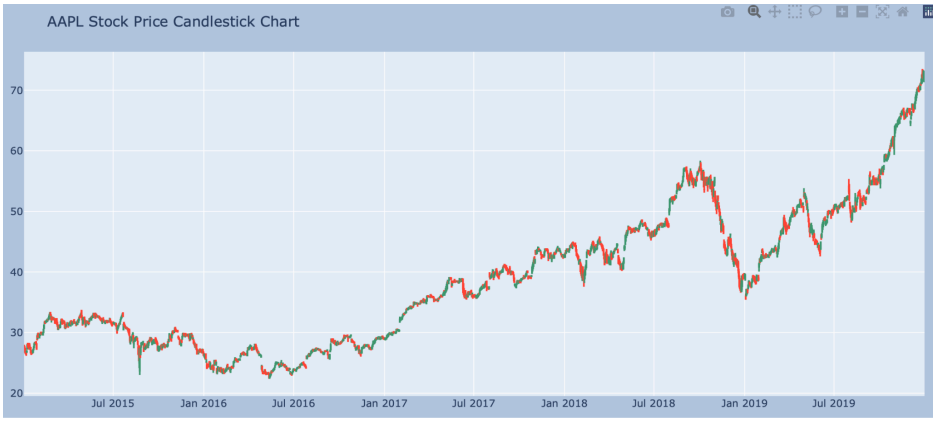


Figure 5.5: AAPL Stock Prices: Interpolated Candlestick Chart

5.2.4. Bollinger Band Visualization

Bollinger Bands are used to identify volatility of the stock prices. It consists of three lines:

- **Middle Band**: A simple moving average (SMA) of the asset's price, typically over 20 days.
- **Upper Band**: The middle band plus 2 standard deviations.
- **Lower Band**: The middle band minus 2 standard deviations.

Figure 5.6: AAPL Bollinger Band

The Bollinger Bands for AAPL show that the stock's closing price moves within the upper and lower bands most of the time. When the price approaches or breaches the upper band, it suggests potential overbought conditions, while movements near or below the lower band indicate possible oversold conditions. The widening of the bands during periods of volatility, particularly in late 2018

and 2019, indicates increased market volatility, while the narrowing bands suggest lower volatility.

6. ARIMA Model

ARIMA (AutoRegressive Integrated Moving Average) is a widely-used statistical model used for time series forecasting [1], based on the trends and patterns in historical data. As the name suggests, it is characterized by three components:

- **AutoRegression (AR)**: It models the relationship between the current value and its previous values, defined by the order of **p**.
- **Integrated (I)**: Accounts for differencing of the series to make it stationary, defined by the order **d**.
- **Moving Average (MA)**: Models the relationship between the current value and past forecast errors, defined by the order **q**.

6.1. Data Split and Normalization

Figure 6.1: AAPL Adjusted Close Price - Data Split

We first split the data into a **training set (80%)** and a **test set (20%)**, ensuring we have a portion of unseen data to evaluate the model's performance effectively.
Following this, the data is normalized using the **MinMaxScaler**, scaling the values between 0 and 1. This transformation is done to enhance the efficiency of ARIMA and standardize the inputs.

```
# Normalize the data
scaler = MinMaxScaler(feature_range=(0, 1))
train_data_scaled = scaler.fit_transform(train_data.values.reshape(-1, 1))
test_data_scaled = scaler.transform(test_data.values.reshape(-1, 1))
```

As shown in the figure, the **scaler** is fit to the training data and then applied to the test data to prevent information leakage from the training set into the test set. After the model is trained, the predictions are rescaled back to their original values for accurate interpretation.

6.2. Stationarity Testing

To apply the ARIMA model, the data needs to be **stationary**, meaning it should have a constant mean and variance over time. Stationary data allows the model to predict future values based on consistent patterns observed in the past, whereas non-stationary data with trends, seasonality or varying variance over time may lead to unreliable forecasts.

```python
# Function to perform ADF test and plot rolling statistics
def check_stationarity(data):
    # Calculate rolling statistics
    rolling_mean = data.rolling(window=30).mean()
    rolling_std = data.rolling(window=30).std()

    # Visualize rolling statistics
    plt.figure(figsize=(10,6))
    plt.plot(data, label='Original Data')
    plt.plot(rolling_mean, label='Rolling Mean')
    plt.plot(rolling_std, label='Rolling Std')
    plt.title('Rolling Mean & Standard Deviation')
    plt.legend()
    plt.show()

    # ADF Test
    adf_result = adfuller(data)
    print(f'ADF Statistic: {adf_result[0]}')
    print(f'p-value: {adf_result[1]}')
    print(f'Critical Values: {adf_result[4]}')

# Check stationarity of the scaled training data
check_stationarity(train_data_scaled)
```

Figure 6.2: check_stationarity function implementation

Two methods are used to test stationarity using check_stationarity method as shown -

6.2.1. **Rolling Statistics**: Rolling mean and Variance is plotted to analyze rolling mean and standard deviation.

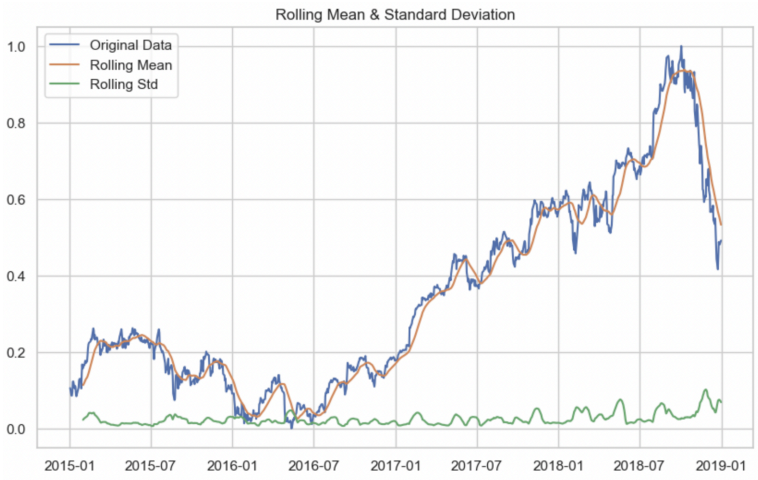


Figure 6.3: Rolling Mean and Std Dev

We see from the above graph that the rolling mean moves along with the adjusted close price, indicating **non-stationarity**, while the rolling standard deviation remains relatively stationary around 0.

6.2.2. **Augmented Dickey-Fuller (ADF) Test-** The null hypothesis is that the data is non-stationary. A p-value greater than 0.05 indicates non-stationarity.

```
ADF Statistic: -1.1583838403667157
p-value: 0.6912370099818075
Critical Values: {'1%': -3.4348678719530934, '5%': -2.863535337271721, '10%': -2.5678323015457787}
```

Figure 6.4: ADF Statistics

The **ADF test statistic** is **-1.158** with a **p-value of 0.691**, which is greater than 0.05. This suggests we fail to reject the null hypothesis and that the **series is non-stationary**, requiring differencing or transformation before proceeding with ARIMA modeling.

6.3. First Order Differencing

After we determine that the data is non-stationary, first order of differencing is performed using the equation :

$$[\ Y't = Y\_t - Y\{t\text{-}1\}\ ]$$

where (Y'_t) represents the differenced stock price at time (t).

```
# First-order differencing
train_data_diff = train_data_scaled.diff().dropna()
```

Figure 6.5: First Order Differencing

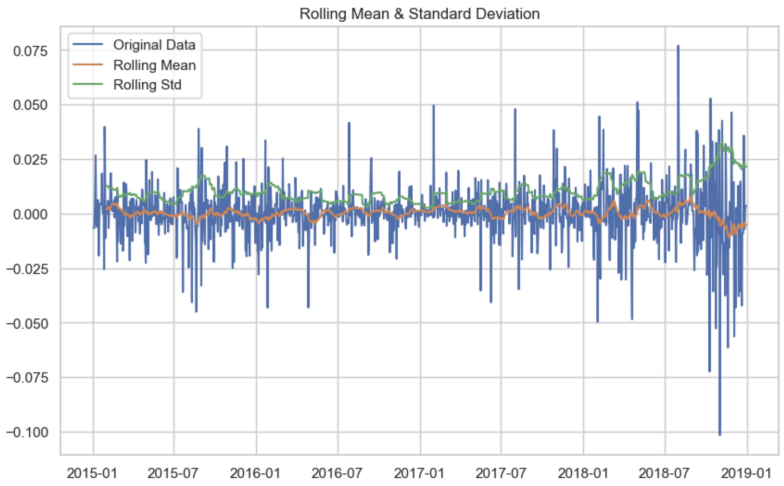


Figure 6.6: Rolling Statistics after First Order Differencing

After applying differencing, both the mean and standard deviation become stationary, indicating that the data is now likely stationary. To confirm this, we perform the **Augmented Dickey-Fuller (ADF) test** on the differenced data, ensuring the stationarity of the series.

```
ADF Statistic: -11.728135841265976
p-value: 1.3619416234954532e-21
Critical Values: {'1%': -3.4348678719530934, '5%': -2.863535337271721, '10%': -2.5678323015457787}
```

Figure 6.7: ADF Statistics after First Order Differencing

The ADF test statistic of -11.73 and a p-value of 1.36e-21 (well below 0.05) confirm that the null hypothesis can be rejected suggesting that the **differenced data is stationary**, as the test statistic is significantly lower than the critical values at all levels.

6.4. Autocorrelation & Partial Autocorrelation Plot Analysis

The **Autocorrelation Function (ACF)** and **Partial Autocorrelation Function (PACF)** plots are used to determine the appropriate values for the **moving average (MA)** term **q** and the **autoregressive (AR)** term **p**, respectively. Significant spikes at specific lag terms in the ACF indicate potential values for **q**, while spikes in the PACF suggest values for **p**.

As previously determined, first-order differencing is necessary to achieve stationarity in the data before applying the ARIMA model. Consequently, the **ACF** and **PACF** plots are analyzed based on the differenced data.

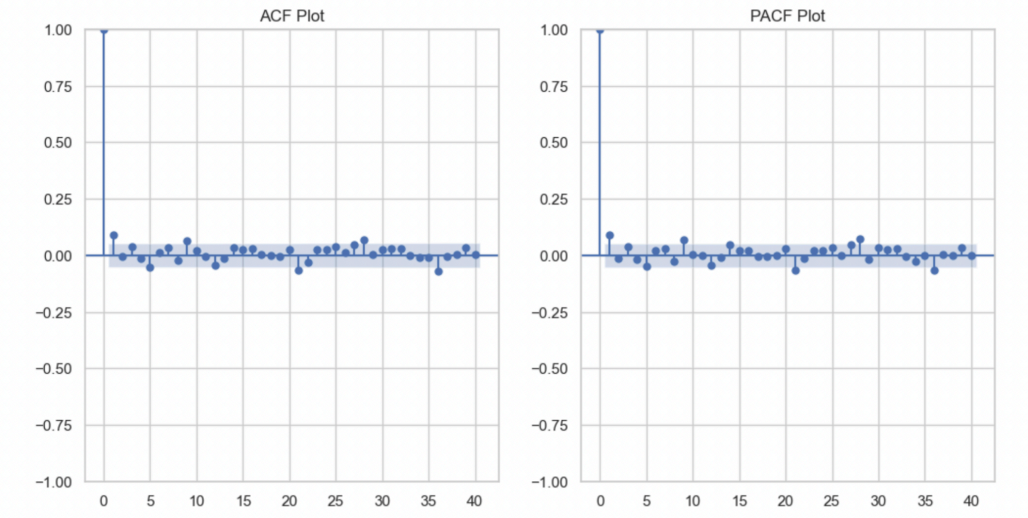


Figure 6.8: ACF and PACF Plot

The ACF plot shows a spike at 1, followed by values that quickly fall within the confidence interval. This suggests that a **moving average (MA)** term of **1** could be appropriate, as subsequent lags exhibit minimal autocorrelation.

Similarly, the **Partial Autocorrelation Function (PACF)** plot shows a spike at lag 1, with the remainder of the lags falling within the confidence interval. This indicates that an **autoregressive (AR)** term of **0** or **1 is** likely suitable.

Thus, based on the ACF and PACF plots, an **ARIMA(0, 0, 1)** or **ARIMA(1,0,1)** model appears to be the most appropriate choice for the differenced data.

6.5. Model Selection using Auto-Arima

Auto-ARIMA function from the **pmdarima** library is used to determine the best ARIMA model parameters, ensuring both accuracy in forecasts and model simplicity. This function was applied to the data that had already undergone first-order differencing to achieve stationarity.

This method automatically identifies the optimal values for the autoregressive (AR) term **p**, differencing term **d**, and moving average (MA) term **q** by assessing a variety of models based on the data. Auto-ARIMA selects the optimal model by minimizing the **Akaike Information Criterion (AIC)**, which balances model complexity and model fit.

```
# Fit auto_arima to the differenced data
model_auto = pm.auto_arima(train_data_diff, seasonal=False, trace=True, stepwise=True, suppress_warnings=True)
print(model_auto.summary())
model_auto.plot_diagnostics(figsize=(16,8))
plt.show()
```

Figure 6.9: Fit auto_arima to Differenced Data

```
Performing stepwise search to minimize aic
 ARIMA(2,0,2)(0,0,0)[0]             : AIC=-8841.115, Time=0.16 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=-8835.161, Time=0.02 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=-8845.651, Time=0.02 sec
 ARIMA(0,0,1)(0,0,0)[0]             : AIC=-8845.974, Time=0.02 sec
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=-8844.305, Time=0.02 sec
 ARIMA(0,0,2)(0,0,0)[0]             : AIC=-8844.126, Time=0.04 sec
 ARIMA(1,0,2)(0,0,0)[0]             : AIC=-8842.135, Time=0.06 sec
 ARIMA(0,0,1)(0,0,0)[0] intercept   : AIC=-8844.601, Time=0.13 sec

Best model:  ARIMA(0,0,1)(0,0,0)[0]
Total fit time: 0.474 seconds
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 1459
Model:               SARIMAX(0, 0, 1)   Log Likelihood                4424.987
Date:                Tue, 17 Sep 2024   AIC                          -8845.974
Time:                        20:30:41   BIC                          -8835.403
Sample:                    01-03-2015   HQIC                         -8842.031
                         - 12-31-2018
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ma.L1          0.0937      0.018      5.126      0.000       0.058       0.130
sigma2         0.0001   2.04e-06     66.707      0.000       0.000       0.000
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):              6908.22
Prob(Q):                              0.99   Prob(JB):                         0.00
Heteroskedasticity (H):               2.58   Skew:                            -0.58
Prob(H) (two-sided):                  0.00   Kurtosis:                        13.60
===================================================================================
```

Figure 6.10: Auto ARIMA model summary

The **Auto-ARIMA** function identified **ARIMA(0, 0, 1)** as the best model based on the lowest **AIC.** This is consistent with what was determined from the ACF and PACF plots.

6.6. ARIMA Model fitting

Based on the results from **Auto-ARIMA**, the **ARIMA(0, 0, 1)** model was selected as the optimal configuration. Using this model, we fit the differenced data to capture short-term dependencies as indicated by the significant moving average (MA) term at lag 1.

```
# Fit ARIMA model using the selected parameters (0, 0, 1) on differenced data
p, d, q = model_auto.order
model_arima = ARIMA(train_data_diff, order=(p, d, q))
arima_result = model_arima.fit()
```

Figure 6.11: Fit Arima Model on optimal parameters

After fitting the ARIMA model on the training data, the residuals were analyzed using diagnostic plots.

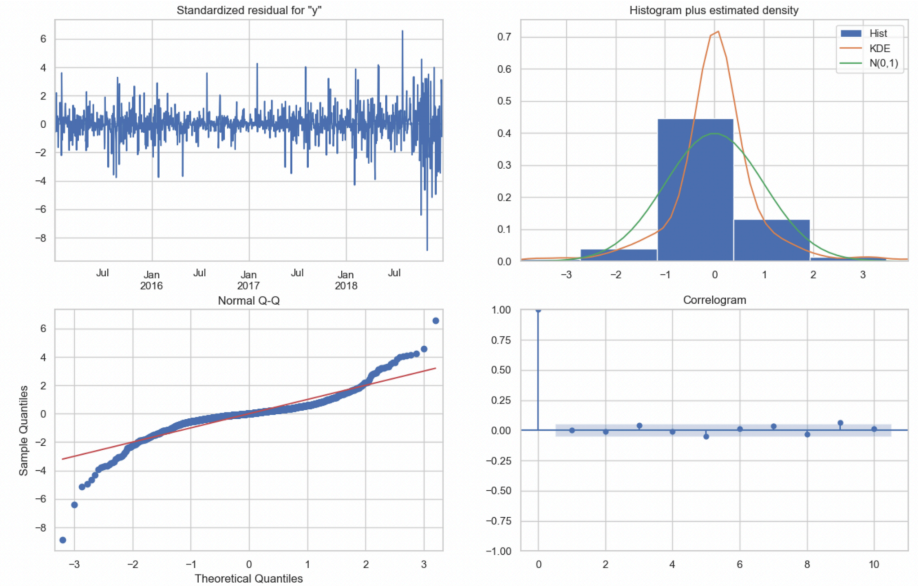


Figure 6.12: ARIMA Model Diagnostic Plots

The diagnostics plots help assess the residuals of the ARIMA model:

- **Standardized Residuals**: The residuals randomly fluctuate around zero which indicates that the model is successful in capturing the underlying trend of the data quite well.
- **Histogram plus Estimated Density**: The residuals appear roughly normally distributed, though there are slight deviations from the normal curve, suggesting minor non-normality.
- **Normal Q-Q Plot**: The Q-Q plot shows slight deviation from the red line only at the tails. This indicates that the residuals are not perfectly normally distributed.
- **Correlogram**: The autocorrelation of residuals falls within the confidence interval, which suggests that there is no significant autocorrelation left in the residuals. We can conclude that the model fits the data well.

Although there are minor deviations from normality in the residuals, the model seems to fit well on the differenced data.

6.7. Forecasting on the Test Set

```
# Forecasting for the test set length (differenced data)
test_data_scaled_diff = test_data_scaled.diff().dropna()
fc = arima_result.forecast(steps=len(test_data_scaled_diff), alpha=0.05)

# Convert the forecasted differenced data back to original scale
last_value = train_data_scaled.iloc[-1]  # Last value of scaled training data
fc_cumulative = last_value + np.cumsum(fc)  # Cumulative sum to revert differencing

# Convert forecasted cumulative sum back to original scale
fc_series_scaled = pd.Series(fc_cumulative.values, index=test_data_scaled.index[len(test_data_scaled) - len(fc_cumulative):])
fc_series = scaler.inverse_transform(fc_series_scaled.values.reshape(-1, 1))  # Inverse transform
fc_series = pd.Series(fc_series.flatten(), index=fc_series_scaled.index)
```

Figure 6.13: Forecasting on Test Set

The test set was first differenced to align with the preprocessing applied to the training data, ensuring consistency when applying the ARIMA model. After generating forecasts using the model, the differencing was reversed by applying a **cumulative sum method to restore the data to its original scale**. The forecasted values were then plotted to visually compare them with the actual test set values, providing a clear representation of the model's predictive accuracy.

6.8. Model Performance - Metrics and Visualization

The following table summarizes the performance of the ARIMA model on both the training and test datasets using key evaluation metrics such as R^2, Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), and Mean Squared Error (MSE).

	Metric	Training Data	Test Data
0	R^2	0.586365	-1.398789
1	MAE	4.475619	11.081220
2	MAPE	0.133206	0.202662
3	MSE	31.516749	178.536682

Figure 6.14: ARIMA Performance Metrics

The model shows **moderate performance on the training data**, with an R^2 value of 0.59 and reasonably low error metrics (MAE = 4.48, MAPE = 13%, MSE = 31.52), indicating that the model captures the overall trend reasonably well.

However, **performance on the test data is significantly worse**, as indicated by a negative R^2 (-1.40) and higher error metrics (MAE = 11.08, MAPE = 20%, MSE = 178.54). This suggests that the model struggles to generalize to unseen data and is likely **unable to capture the complexity of the time series dataset**.

6.9. ARIMA Performance Visualization

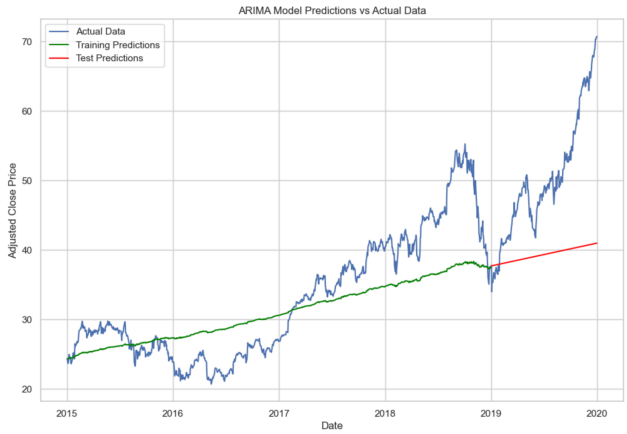


Figure 6.15: ARIMA Prediction vs Actual Data

Above graph gives visualization of ARIMA's predictive performance on both the training data and test data for adjusted close price of stocks.

- Training Prediction: The model **captures the general upward trend but lacks precision to accurately reflect the fluctuations** within the curve. It indicates that the model has a tendency to smoothen out the short-term fluctuations
- Test Predictions: There is a clear divergence from the actual stock prices even though it manages to capture the basic upward trend of the graph. It f**ails to capture the strong upward movement and volatility present within the actual data**. This causes the poor performance on the test set as seen by the negative **R^2** score.

Overall, the model fits the training data moderately well but **struggles to generalize to the unseen test data, potentially due to the limited complexity of the ARIMA configuration**.

7. LSTM

Long Short-Term Memory (LSTM) networks are a special type of recurrent neural network (RNN), that can be used in the domain of sequential data modeling such as stock price production as stock prices are inherently time-dependent. LSTM is able to capture long-term dependencies within the time series data while also mitigating the vanishing gradient problem of traditional RNN. In this project, LSTM is applied to predict stock prices of Apple based on historical adjusted close price data.

7.1. Data Split for and Data Normalization

The stock data for adjusted closing price (input feature) is split as follows:

- **Training Set: 70%** of total data, used to train the LSTM model
- **Validation Set: 15%** of the data used for tuning model parameters and prevent overfitting
- **Test Set: 15%** remaining data used to evaluate the model's performance on unseen data.

```
# Determine the split indices
total_length = len(apple_stock_data_cleaned)
train_end = round(0.7 * total_length)
val_end = train_end + round(0.15 * total_length)

# Split the data
train_data = apple_stock_data_cleaned[:train_end].iloc[:,4:5]
val_data = apple_stock_data_cleaned[train_end:val_end].iloc[:,4:5]
test_data = apple_stock_data_cleaned[val_end:].iloc[:,4:5]

# Display the sizes of each dataset
print(f"Training set size (70%): {len(train_data)}")
print(f"Validation set size (15%): {len(val_data)}")
print(f"Test set size (15%): {len(test_data)}")
```

```
Training set size (70%): 1278
Validation set size (15%): 274
Test set size (15%): 273
```

Figure 7.1: Input Data Split (75:15:15)

As seen in figure below, the **MinMaxScaler** is then used to normalize the data to a range between 0 and 1. Normalization of input features is crucial to ensure effective training of the LSTM model by stardizing the scale of data.

The scaler is fitted on the training set and transforms the data to fit within the specified range. The same scaler is then used to transform both the validation and test data so as to ensure consistency in data scaling across all datasets.

This **normalization inturn helps the model to converge at a faster rate and also improves the model's ability to learn the patterns within stock price data accurately**.

```
# Initialize the MinMaxScaler
scaler = MinMaxScaler(feature_range=(0,1))

# Fit the scaler on the training data and transform it
train_data_scaled = scaler.fit_transform(train_data)

# Transform the validation and test data using the same scaler
val_data_scaled = scaler.transform(val_data)
test_data_scaled = scaler.transform(test_data)
```

```
Scaled Training data:
 [[0.14426807]
 [0.13523101]
 [0.12619395]
 [0.11715689]
 [0.11724492]]
Scaled Validation data:
 [[0.93626669]
 [0.90571197]
 [0.91264776]
 [0.91958355]
 [0.94367097]]
Scaled Test data:
 [[0.99462286]
 [1.02097373]
 [1.03357999]
 [1.03680287]
 [1.04922026]]
```

Figure 7.2: Data Scaling

Reason to fit on only training set and then use the scalar to transform the Validation set and Test set:

The model should not have any prior knowledge about the validation and test data. Normalizing before splitting may unintentionally leak the information from validation or test sets into the training process which will lead to possibly overfitting the model or to produce biased results. Therefore, the scaler is evaluated using a training set and then used to transform the remaining dataset.

7.2. Sequence Creation for LSTM Model

```
def create_sequences(data, sequence_length):
    X, y = [], []
    for i in range(len(data) - sequence_length):
        X.append(data[i:i + sequence_length])
        y.append(data[i + sequence_length])
    return np.array(X), np.array(y)

# Define the sequence length
sequence_length = 30  # Number of time steps in each sequence

# Create sequences for training, validation, and test sets
X_train, y_train = create_sequences(train_data_scaled, sequence_length)
X_val, y_val = create_sequences(val_data_scaled, sequence_length)
X_test, y_test = create_sequences(test_data_scaled, sequence_length)
```

Figure 7.3: creat_sequence function for LSTM model

```
X_train shape: (1248, 30, 1)
y_train shape: (1248, 1)
X_val shape: (244, 30, 1)
y_val shape: (244, 1)
X_test shape: (243, 30, 1)
y_test shape: (243, 1)
```

Figure 7.4: Split Data Shape Analysis

Create_sequences function generates sequences of stock price data to be fed into the LSTM model. This function transforms the normalized data into s**equences with a fixed number of time steps (30 time steps)**. These sequences are used as input features to predict the next stock price.

- **Input**(X): Contains the **previous 30 time steps** of data.
- **Output**(y): The **corresponding ground truth label** (31st time step after every sequence of 30 time steps) which is the stock price of the next time step following the sequence.

7.3. Hyperparameters Tuning - Grid Search

Before finalizing the LSTM structure, an experiment was conducted using grid search with different hyperparameters specifications, specifically the **number of units** (neurons) and the **batch size**. The goal is to find the most optimal combination that minimizes the

validation loss as well as reduces the gap between the training and validation loss so as to avoid both underfitting and overfitting.
Following configurations were tested -

- Number of units: [50, 100, 200].
- Batch size: [16, 32, 64].

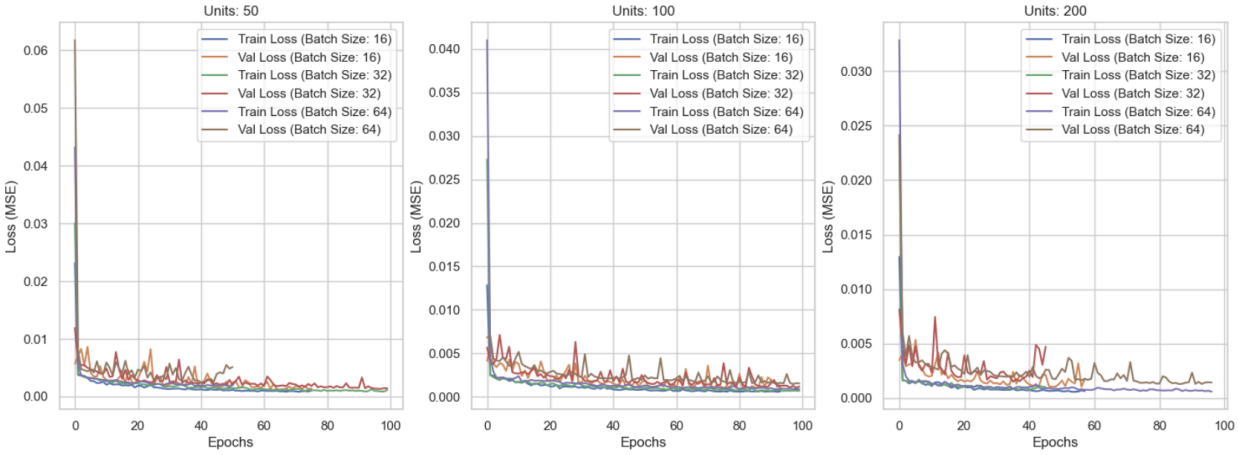


Figure 7.5: Training Loss curve across different num_units and batch Size

Best combinations based on Avg Validation Loss and Gap:

	Units	Batch Size	Avg Train Loss	Avg Val Loss	Gap
0	100	32	0.000696	0.001174	0.000478
1	50	32	0.000958	0.001383	0.000425
2	200	64	0.000667	0.001431	0.000764
3	50	16	0.000912	0.001492	0.000580
4	100	16	0.000609	0.001676	0.001066
5	100	64	0.000910	0.001758	0.000848
6	200	16	0.000631	0.001764	0.001133
7	200	32	0.001000	0.003825	0.002825
8	50	64	0.001918	0.004715	0.002797

Figure 7.6: Hyperparameter Tuning - Best model architectures

Figure above shows the Avg training loss, Avg Validation loss as well as the gap between the two across different combinations of number of units and batch size. From above table it is apparent that the most **optimal combination of hyperparameters** would be to set

- **Number of units =200**
- **Batch Size - 32**

This pair offers the **best balance between model performance and training stability**, with the smallest gap between training and validation loss, **ensuring effective generalization** to unseen data.

7.4. LSTM Architecture

```python
# Define the LSTM model
def create_lstm_model(input_shape, num_of_units, dropout_ratio, summary):
    lstm_model = Sequential()
    lstm_model.add(Input(shape=(X_train.shape[1], 1)))
    lstm_model.add(LSTM(num_of_units, return_sequences= True))
    lstm_model.add(Dropout(dropout_ratio))
    lstm_model.add(LSTM(num_of_units, return_sequences= False))
    lstm_model.add(Dropout(dropout_ratio))
    lstm_model.add(Dense(1))
    lstm_model.compile(optimizer = "adam", loss = "mean_squared_error")
    if summary==True:
        lstm_model.summary()
    return lstm_model

# Define input shape
input_shape = (X_train.shape[1], 1)  # (sequence_length, number of features)
```

```python
# Create the LSTM model
model = create_lstm_model(input_shape, num_of_units=100, dropout_ratio=0.2, summary=True)
```

Figure 7.7: create_lstm_model function

The create_lstm_model function is used to create the LSTM structure with following specifications (**2 hidden layers, each with 100 units and a dense layer to get the final output prediction**).

Model: "sequential_9"

Layer (type)	Output Shape	Param #
lstm_18 (LSTM)	(None, 30, 100)	40,800
dropout_18 (Dropout)	(None, 30, 100)	0
lstm_19 (LSTM)	(None, 100)	80,400
dropout_19 (Dropout)	(None, 100)	0
dense_9 (Dense)	(None, 1)	101

Total params: 121,301 (473.83 KB)
Trainable params: 121,301 (473.83 KB)
Non-trainable params: 0 (0.00 B)

Figure 7.8: LSTM model summary

7.4.1. Model Structure

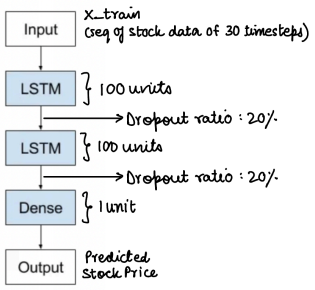


Figure 7.9: LSTM Model Architecture

- **Input Layer**: This layer takes in the input features as (X_train.shape[1], 1), where X_train.shape[1] is the sequence length and 1 represents the number of features.
- **First LSTM Layer**: This layer contains **100 units** and outputs the full sequence of hidden states, which is required for stacking LSTM layers.
- **Dropout Layer**: A dropout ratio of **0.2** is applied to the output of the first LSTM layer to prevent overfitting by randomly setting 20% of input units to 0 during training.
- **Second LSTM Layer** - Also contains **100 units** but does not return sequences. It only outputs the final hidden state which is then fed to the Dense Layer.
- **Dropout Layer**: Another dropout layer with a ratio of **0.2** is applied after the second LSTM layer.
- **Dense Layer**: The Dense layer with 1 unit produces the final output, which corresponds to the predicted stock price.
- **Compilation**: Model uses compiled with **Adam optimizer** and **MSE loss function**, that is suitable for regression tasks.

7.4.2. LSTM Gates and Equations:

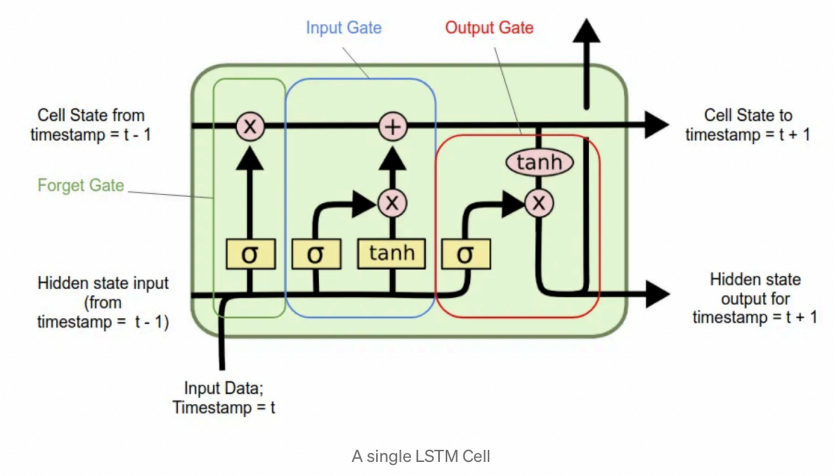


Figure 7.10: Single LSTM cell

LSTM incorporates memory units that allow the neural network to learn when to **forget the previous hidden states** and when to **update hidden states given new information.**

It uses gates to add and remove information from the cell state (long term memory) as required. It makes use of following gates -

- **Forget Gate**: Determines which information from the previous state to remember and which to forget, as needed.

$$\boldsymbol{f}(t) = \sigma\left(\boldsymbol{U}_f^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_f^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_f\right)$$

Sigmoid Layer outputs numbers in range [0,1] to determine how much of the information to let through.

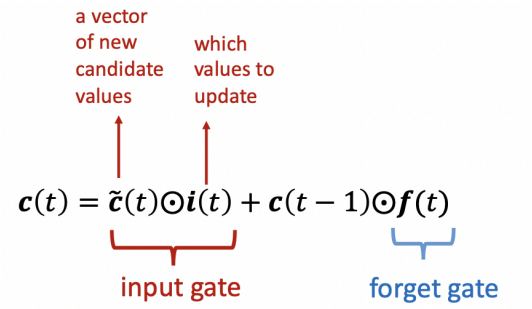
- **Input Gate**: Determines if the values should be updated in the cell state and if so, which ones should be updated. This is done using 2 parts -
    1. **Sigmoid Input Gate Layer**: Decide which values to update

$$\boldsymbol{i}(t) = \sigma\left(\boldsymbol{U}_{\boldsymbol{i}}^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_{\boldsymbol{i}}^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_{\boldsymbol{i}}\right)$$

    2. **Tanh Layer**: Create a layer of new candidate values to potentially be added to the state.

$$\tilde{\boldsymbol{c}}(t) = \phi\left(\boldsymbol{U}_{\boldsymbol{c}}^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_{\boldsymbol{c}}^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_{\boldsymbol{c}}\right)$$

- **Cell state**: Cell state gets updated by adding the old cell state with the new cell state



- **Output gate**: It allows the state of the memory cell to have an effect on the following neuron or prevent it.

$$\boldsymbol{o}(t) = \sigma\left(\boldsymbol{U}_{\boldsymbol{o}}^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_{\boldsymbol{o}}^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_{\boldsymbol{o}}\right)$$

$$\boldsymbol{h}(t) = \phi\left(\boldsymbol{c}(t)\right) \odot \boldsymbol{o}(t)$$

7.4.3. Early Stopping and Model Training

```python
# Train the LSTM model with early stopping
def train_lstm_model(model, batch_size, epochs, verbose):
    # Set up early stopping to monitor validation loss and restore best weights
    # patience (int) – Number of events to wait if no improvement and then stop the training.
    # verbose displays messages when the callback takes an action
    early_stop = EarlyStopping(monitor='val_loss', patience=10, verbose=0, restore_best_weights=True)
    history = model.fit(
        X_train,
        y_train,
        validation_data=(X_val, y_val),
        epochs=100,
        batch_size=batch_size,
        callbacks=[early_stop],
        verbose=verbose
    )
    return history
```

```python
history = train_lstm_model(model=model, batch_size=32, epochs=100,verbose=1)
```

Figure 7.11: train_lstm_model function

In the training process of learning optimal hyperparameters through weights and biases, **EarlyStopping** is used to **prevent overfitting**. It halts the training if no

improvement in the loss score is observed for 10 consecutive epochs. It ensures optimal performance by restoring the best weights to achieve the most accurate model possible.

The model is then trained for up to **100 epochs** with a **batch size of 32**, leveraging early stopping to optimize performance and prevent overfitting.

7.4.4. Training Loss vs Validation Loss

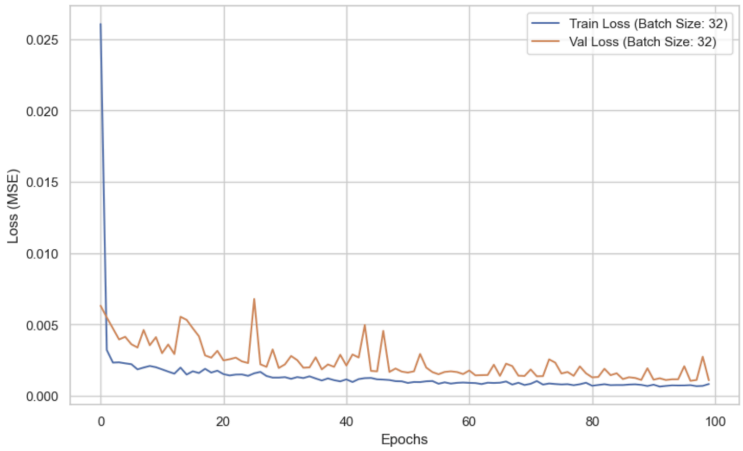


Figure 7.12: LSTM Model: Training vs Validation Loss across epochs

As seen in the graph, the **model converges quickly for both the training set as well as validation set**. The small gap between the convergence rates of the training set and the validation set suggests that there is no overfitting done on the data.

7.4.5. Performance Evaluation

```
# Make predictions using the trained LSTM model
predictions_train = model.predict(X_train)
predictions_valid = model.predict(X_val)
predictions_test = model.predict(X_test)
```

Figure 7.13: LSTM model predictions

After training, the model's performance is evaluated by making predictions on training set, validation set, and test dataset and comparing the prediction values against the ground truth label values.

```
# Rescale predictions and actual values back to their original scale
predictions_train_original = scaler.inverse_transform(predictions_train)
predictions_valid_original = scaler.inverse_transform(predictions_valid)
predictions_test_original = scaler.inverse_transform(predictions_test)

actual_train_original = scaler.inverse_transform(y_train)
actual_valid_original = scaler.inverse_transform(y_val)
actual_test_original = scaler.inverse_transform(y_test)
```

Figure 7.14: Rescale predictions to original scale

These predictions as well as the original values are then rescaled back to their original scale using the scaler to facilitate accurate performance evaluation.

Following this, Performance Metrics such as Mean Square Error (MSE), Mean Absolute Percentage Error (MAPE) and R^2 score are calculated to evaluate the model performance on train, validation and test datasets.

	Metric	Training	Validation	Test
0	MSE	0.170396	0.682905	1.456814
1	MAPE	0.009484	0.013258	0.016370
2	R^2	0.996362	0.981402	0.975440

Figure 7.15: LSTM model performance metrics

The model shows excellent performance on the training data with very low MSE and MAPE, which indicates a high accuracy. The validation and test metrics are slightly higher (as expected on unseen data vs seen data) but still reflects **strong predictive capability with R^2 values above 0.97**, **demonstrating good generalization**.

7.4.6. Model Prediction Performance Visualization

Model Predictions vs [Training, Validation, Test] Datasets are visualized as below -

Figure 7.16: Training: Actual vs Predicted

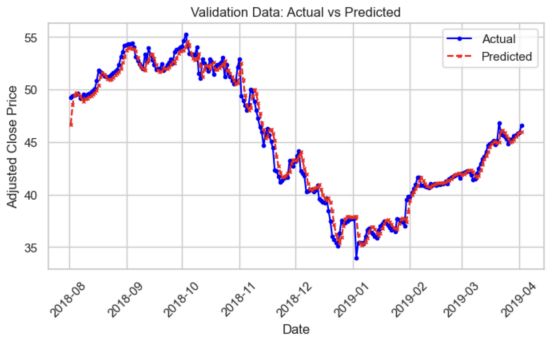


Figure 7.17: Validation: Actual vs Predicted

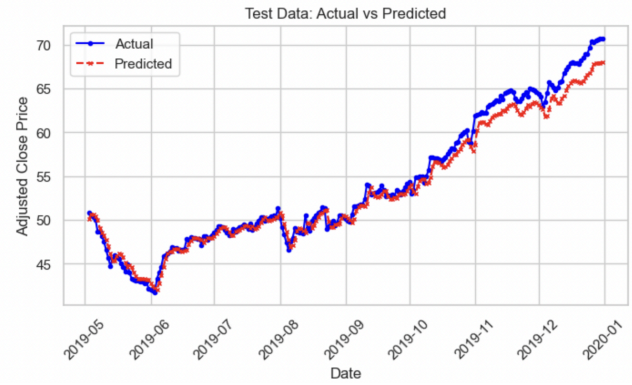


Figure 7.18: Testing: Actual vs Predicted

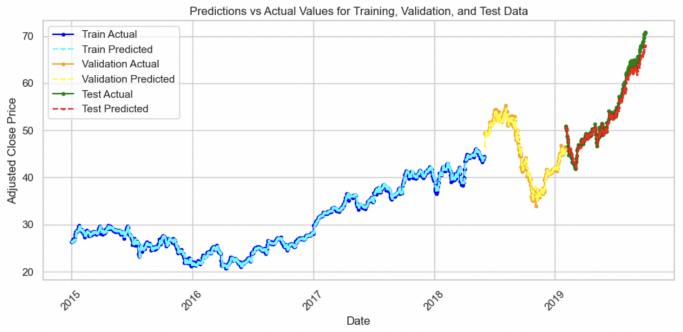


Figure 7.19: Actual vs Predicted Adj Close Price

These graphs further prove the excellent performance of the LSTM model to capture underlying relationships in historical data with respect to all training, validation and test sets. [6]

8. LSTM With Sentiment Analysis

Long Short-Term Memory (LSTM) networks are powerful tools for time series forecasting. When combined with sentiment analysis, this LSTM network will utilize not only the historical stock prices but also textual **data insights from social media such as X** so as to be able to make more informed predictions.

Hypothesis: Incorporating sentiment scores derived from tweets for the same time period of 2015 to 2019, can potentially enhance the model's ability to forecast stock price movements because market sentiment has a significant role in market trends.

8.1. Sentiment Analysis on Apple Data (2015-2019)

We begin by extracting data about 'AAPL' from the twitter dataset. The sentiment scores are then generated from these tweets to get a measure of market sentiment for each day. These sentiment scores are then incorporated into the input dataset for LSTM alongside Adjusted Close Price allowing the LSTM model to learn both from historical price data as well as the corresponding public sentiment.

8.1.1. Extraction of Apple Data from the Twitter Dataset

The original Twitter dataset contains 3,717,964 rows, with each row representing a tweet and columns like tweet_id, writer, post_date, body, comment_num, retweet_num, and like_num.

```
# Extract tweets about AAPL from twitter dataset
aapl_tweets = tweet_df[tweet_df['body'].str.contains('AAPL', case=False, na=False)]
aapl_tweets['post_date'] = pd.to_datetime(aapl_tweets['post_date'], unit='s')
aapl_tweets.head()
```

Figure 8.1: Extract AAPL tweets

After filtering the dataset to **extract tweets specifically mentioning "AAPL"**, we are left with 1,480,311 rows. These filtered tweets will be used for sentiment analysis related to Apple stock movements.

8.1.2. AAPL Twitter Data Cleaning and Preprocessing

1. Handling Missing Values:
   To ensure data quality and integrity, we check and drop any missing values within the body column (containing tweet text)

```
# Check for missing values
print(aapl_tweets.isnull().sum())  # Identify any columns with missing values
# Drop rows with missing values in the 'body' column
aapl_tweets.dropna(subset=['body'], inplace=True)
```

Figure 8.2: Twitter Data Cleaning

2. Preprocessing Text Data in tweet 'body'

```
# Preprocessing the text data
# Define a function to clean and preprocess the tweet text
def preprocess_text(text):
    # Convert text to lowercase
    text = text.lower()

    # Remove URLs
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)

    # Remove special characters, numbers, and punctuation
    text = re.sub(r'\@w+|\#', '', text)  # Remove mentions and hashtags
    text = re.sub(r'[^a-zA-Z\s]', '', text)

    # Tokenize the text
    tokens = word_tokenize(text)

    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words]

    # Lemmatize the tokens (you can switch to PorterStemmer if preferred)
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens]

    # Join tokens back into a string
    clean_text = ' '.join(tokens)
    return clean_text
```

Figure 8.3: AAPL tweets preprocessing

The tweet texts are then cleaned and preprocessed to remove irrelevant elements such as URLs, special characters, and punctuation. The text is tokenized, stop words are removed, and the words are lemmatized to reduce them to their base forms. This helps to improve the consistency and effectiveness of the sentiment analysis.

8.1.3. Word Cloud Visualization

The word cloud visualizes the most frequently mentioned words in the preprocessed tweets about Apple (AAPL), with larger words representing higher frequency.

Figure 8.4: AAPL tweets WordCloud

Here key terms highlighted are "aapl," "apple," "iphone," which refers to the company and its products, as well as other related tech stock symbols and keywords. This gives an immediate sense of the most common topics or sentiments expressed in the tweets.

8.1.4. Sentiment Analysis

We apply sentiment analysis using **NLTK's SentimentIntensityAnalyzer** to assign a sentiment score to each tweet.

```
# Sentiment Analysis using NLTK's SentimentIntensityAnalyzer
# Initialize the sentiment intensity analyzer
sia = SentimentIntensityAnalyzer()

# Define a function to calculate sentiment scores
def get_sentiment_scores(text):
    return sia.polarity_scores(text)['compound']

# Apply the function to the clean text column
aapl_tweets['sentiment_score'] = aapl_tweets['clean_body'].apply(get_sentiment_scores)
```

Figure 8.5: Sentiment Analysis on AAPL tweets

The compound score is extracted, which ranges from -1 (most negative) to +1 (most positive), providing a comprehensive measure of the overall sentiment in each tweet.

8.1.5. Aggregating Sentiment Scores by data

The sentiment scores are aggregated by date to calculate the average daily sentiment.

```
# Aggregating sentiment scores by date
# Ensure the 'post_date' column is in datetime format
aapl_tweets['post_date'] = pd.to_datetime(aapl_tweets['post_date'])

# Group by date and calculate the average sentiment score for each day
daily_sentiment = aapl_tweets.groupby(aapl_tweets['post_date'].dt.date)['sentiment_score'
```

Figure 8.6: Aggregate Sentiment scores by date

This allows us to observe the sentiment trend over time, helping to **identify how public opinion about Apple changes on a daily basis and correlating it with corresponding stock price movements.**

8.1.6. Daily Sentiment Trend Visualization

The trend of average daily sentiment scores over time is visualized, allowing us to track how public sentiment towards Apple has moved between 2015 and 2019.

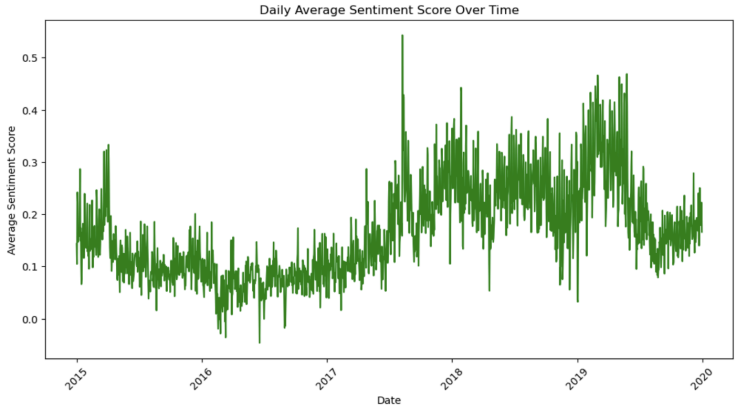


Figure 8.7: Daily Average Sentiment Scores over Time

The graph shows an increasing trend across time series which is consistent with the actual stock price movement during this time.

8.2. Merge Adjusted Close Price with Daily Sentiment Scores

To incorporate sentiment analysis into the stock prediction model, the daily sentiment scores are merged with the adjusted closing prices of Apple stock.
This **combined dataset with 2 features** provides both price data and public sentiment on each day, offering more insights to the LSTM model to model the relationship..

8.3. Data Split and Data Normalization

The merged data for adjusted closing price and sentiment score (input feature) is split as follows:

1. **Training Set: 70%** of total data, used to train the LSTM model
2. **Validation Set: 15%** of the data used to tuning model parameters and prevent overfitting
3. **Test Set: 15%** remaining data used to evaluate the model's performance on unseen data.

```
# Determine the split indices
total_length = len(merged_df)
train_end = round(0.7 * total_length)
val_end = train_end + round(0.15 * total_length)

# Split the data
train_data = merged_df[:train_end].iloc[:,1:3]
val_data = merged_df[train_end:val_end].iloc[:,1:3]
test_data = merged_df[val_end:].iloc[:,1:3]
```

Figure 8.8: Input Data Split (75:15:15)

Following the data split, model's input data - both the adjusted close price and sentiment scores are normalized using a **MinMaxScaler** to scale the features between 0 and 1. The

scaler is fit on the training set to avoid data leakage, followed by transforming and standardizing data across training, validation, and test sets.

```
# Initialize the MinMaxScaler
scaler = MinMaxScaler(feature_range=(0,1))

# Fit the scaler on the training data and transform it
train_data_scaled = scaler.fit_transform(train_data[['Adj Close', 'sentiment_score']])

# Transform the validation and test data using the same scaler
val_data_scaled = scaler.transform(val_data[['Adj Close', 'sentiment_score']])
test_data_scaled = scaler.transform(test_data[['Adj Close', 'sentiment_score']])
```

Figure 8.9: Data Scaling

8.4. Correlation Between Adjusted Close Price and Sentiment Score

Analysis is conducted on the training set to find correlation between the adjusted close price and the sentiment score for corresponding day.

Figure 8.10: Correlation between Adj Close Price and Sentiment Score

The correlation heatmap and scatter plot as shown above together demonstrate a strong positive relationship between the **Adjusted Close price** of Apple's stock and the **sentiment score** derived from tweets. The correlation coefficient of **0.76** in the heatmap indicates a **significant positive correlation**, while the scatter plot visually confirms this by showing that higher sentiment scores tend to align with higher stock prices. This suggests that as public sentiment towards Apple becomes more favorable, the stock price often increases and vice versa, highlighting the potential impact of sentiment on stock market movements.

8.5. Sequence Creation for LSTM Model

The function create_sequences is designed to generate sequences of stock data for the LSTM model, using 30 time steps.

```
def create_sequences(data, sequence_length):
    X, y = [], []
    for i in range(len(data) - sequence_length):
        # Store the sequence for all features
        X.append(data[i:i + sequence_length])
        # The target first feature (Adjusted Close) is the next time step
        y.append(data[i + sequence_length, 0])
    return np.array(X), np.array(y)

# Define the sequence length
sequence_length = 30  # Number of time steps in each sequence
```

Figure 8.11: create_sequence for LSTM+Sentiment model

```
X_train shape: (1248, 30, 2)
y_train shape: (1248,)
X_val shape: (244, 30, 2)
y_val shape: (244,)
X_test shape: (243, 30, 2)
y_test shape: (243,)
```

Figure 8.12: Analyze Input Data Shape

The dataset shapes indicate that each sequence in the training, validation, and test sets consists of **30 time steps** and **2 features** (Adjusted Close and sentiment score), while the **target is a single value** (the Adjusted Close at the 31st time step).

8.6. Hyperparameter Tuning - Grid Search

In this section, a **grid search approach** was used to identify the best combination of **LSTM units** and **batch sizes** for the model. The model was trained with different combinations of units (100, 200, 300) and batch sizes (16, 32, 64, 128) to observe the effect on training and validation loss over 200 epochs.

Each combination was evaluated based on three key metrics:

1. **Average Train Loss**: The mean loss over the last 5 epochs of training data.
2. **Average Validation Loss**: The mean loss over the last 5 epochs of validation data.
3. **Gap**: The difference between validation loss and training loss. A smaller gap indicates that the model is generalizing well to unseen data without overfitting.

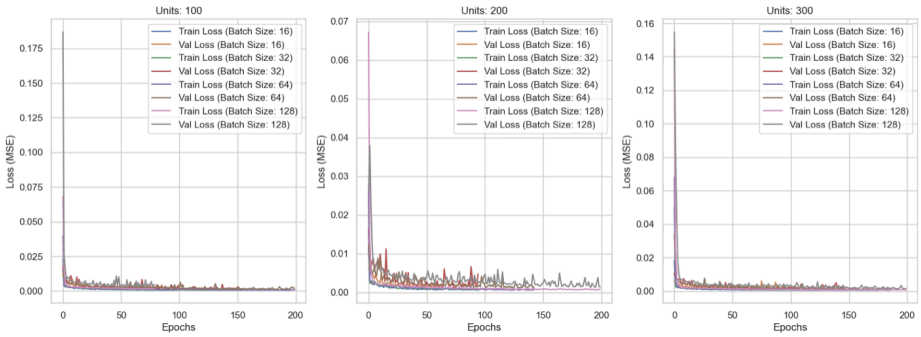


Figure 8.13: Training Loss curve across different num_units and batch Size

```
Best combinations based on Avg Validation Loss and Gap:
[44]:
```

	Units	Batch Size	Avg Train Loss	Avg Val Loss	Gap
0	300	64	0.000576	0.001311	0.000735
1	300	32	0.000555	0.001492	0.000937
2	200	64	0.000649	0.001614	0.000964
3	200	16	0.000752	0.001758	0.001006
4	300	128	0.000715	0.001838	0.001122
5	100	32	0.000679	0.001945	0.001267
6	300	16	0.000654	0.002012	0.001358
7	200	128	0.000737	0.002141	0.001403
8	100	64	0.000824	0.002187	0.001363
9	100	16	0.000703	0.002206	0.001503
10	200	32	0.000696	0.002728	0.002032
11	100	128	0.001601	0.004582	0.002981

Figure 8.14: Hyperparameter Tuning: Best Model Architectures

The above graphs visualize the **training and validation loss curves** for each combination, making it easier to observe the performance across different hyperparameters. From the table (figure 8.14), the optimal combinations were identified based on **minimal validation loss** and **gap** between training and validation.

1. Optimal Configuration: The configuration with **300 units** and **batch size 64** achieved the lowest **average validation loss**
2. General Trend: Models with **300 units** generally performed better across most batch sizes, though **smaller batch sizes** (e.g., 32 and 64) tend to yield more consistent results with lower validation loss.

8.7. LSTM (with Sentiment Analysis) Architecture

```python
# Define the LSTM model
def create_lstm_model(input_shape, num_of_units, dropout_ratio, summary):
    lstm_model = Sequential()
    lstm_model.add(Input(shape=(X_train.shape[1], 2)))
    lstm_model.add(LSTM(num_of_units, return_sequences= True, input_shape=input_shape))
    lstm_model.add(Dropout(dropout_ratio))
    lstm_model.add(LSTM(num_of_units, return_sequences= True))
    lstm_model.add(Dropout(dropout_ratio))
    lstm_model.add(LSTM(num_of_units, return_sequences= False))
    lstm_model.add(Dropout(dropout_ratio))
    lstm_model.add(Dense(1))
    lstm_model.compile(optimizer = "adam", loss = "mean_squared_error")
    if summary==True:
        lstm_model.summary()
    return lstm_model

# Define input shape
input_shape = (X_train.shape[1], 2)  # (sequence_length, number of features)
```

```python
# Create the LSTM model
model = create_lstm_model(input_shape, num_of_units=300, dropout_ratio=0.2, summary=True)
```

Figure 8.15: create_lstm_model function for LSTM+Sentiment model

The create_lstm_model function is used to create the LSTM structure with following specifications **(3 hidden layers, each with 300 units and a dense layer to get the final output prediction)**.

Model: "sequential_18"

Layer (type)	Output Shape	Param #
lstm_54 (LSTM)	(None, 30, 300)	363,600
dropout_54 (Dropout)	(None, 30, 300)	0
lstm_55 (LSTM)	(None, 30, 300)	721,200
dropout_55 (Dropout)	(None, 30, 300)	0
lstm_56 (LSTM)	(None, 300)	721,200
dropout_56 (Dropout)	(None, 300)	0
dense_18 (Dense)	(None, 1)	301

Total params: 1,806,301 (6.89 MB)
Trainable params: 1,806,301 (6.89 MB)
Non-trainable params: 0 (0.00 B)

Figure 8.16: LSTM + Sentiment model summary

8.7.1. Model Structure

The model consists of t**hree LSTM layers**, each containing 300 units, followed by **Dropout layers** to prevent overfitting. The final layer is a **Dense** layer that outputs the predicted stock price.

- **Input Layer**: Takes in **two features** (Adjusted Close and sentiment score) for each time step in the sequence. The shape of the input is

(sequence_length, 2) where 2 corresponds to the number of input features.

- **First LSTM Layer**: Contains **300 units** and returns the full sequence of hidden states. This allows for stacking multiple LSTM layers.
- **Dropout Layer**: A dropout ratio of **0.2** is applied to reduce overfitting by randomly setting 20% of the input units to zero during each update.
- **Second LSTM Layer**: Also contains **300 units** and returns the full sequence. This helps the model capture more complex temporal patterns.
- **Third LSTM Layer**: Contains **300 units** but returns only the last hidden state, which is passed to the dense layer for the final prediction.
- **Final Dense Layer**: A dense layer with **1 unit** outputs the predicted stock price for the next time step.
- **Compilation**: The model is compiled using the **Adam optimizer** and the **Mean Squared Error (MSE)** loss function, making it suitable for the regression task of predicting stock prices.

8.7.2. Early Stopping and Model Training

The train_lstm_model function trains the LSTM model using the **Adam optimizer** and **Mean Squared Error (MSE)**loss function. We include early stopping functionality to avoid overfitting and also to restore the best model weights when the validation loss stops improving for 20 consecutive epochs.

```
# Train the LSTM model with early stopping
def train_lstm_model(model, batch_size, epochs, verbose):
    # Set up early stopping to monitor validation loss and restore best weights
    # patience (int) – Number of events to wait if no improvement and then stop the training.
    # verbose displays messages when the callback takes an action
    early_stop = EarlyStopping(monitor='val_loss', patience=20, verbose=0, restore_best_weights=True)
    history = model.fit(
        X_train,
        y_train,
        validation_data=(X_val, y_val),
        epochs=200,
        batch_size=batch_size,
        callbacks=[early_stop],
        verbose=verbose
    )
    return history
```

```
history = train_lstm_model(model=model, batch_size=64, epochs=200,verbose=1)
```

Figure 8.17: train_lstm_model function for LSTM+Sentiment model

- **Early Stopping**: Monitors the validation loss and stops training if no improvement is observed for 20 consecutive epochs. This helps prevent overfitting while ensuring optimal model performance.
- **Model Training**: The model is trained with a **batch size of 64** and up to **200 epochs**, though early stopping ensures that training will stop if the model stops improving before 200 epochs.

8.7.3. Training Loss vs Validation Loss

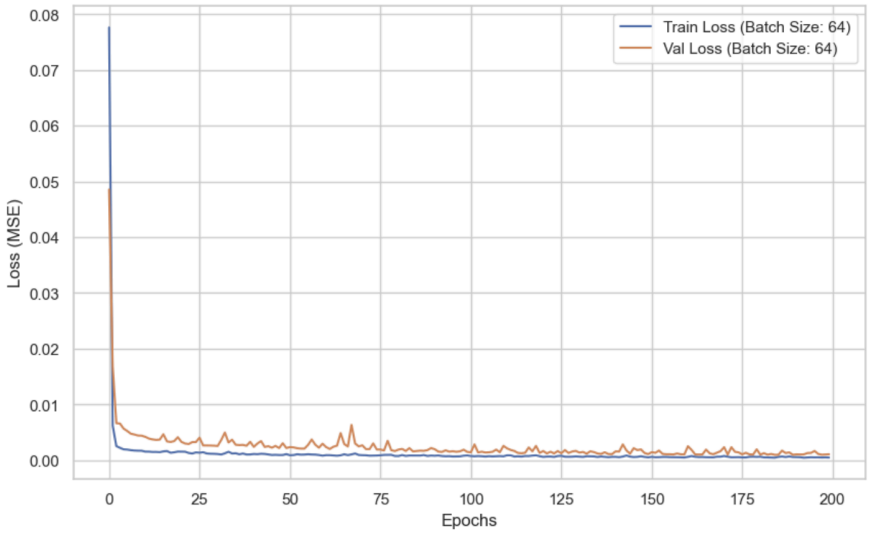


Figure 8.18: Training vs Validation Loss curve across epochs

The graph above shows the training loss decreases quickly and stabilizes at a low value, while the validation loss follows a similar trend with minor fluctuations. The small gap between the two indicates that the model **generalizes well to unseen data with no signs of overfitting**.

8.7.4. Performance Metrics

```
# Make predictions using the trained LSTM model
predictions_train = model.predict(X_train)
predictions_valid = model.predict(X_val)
predictions_test = model.predict(X_test)
```

Figure 8.19: Predictions on LSTM+Sentiment Model

After training the LSTM model, predictions are made on the training, validation, and test datasets using the model's .predict() function. These predictions correspond to the adjusted closing price at the next time step based on the historical stock data and sentiment scores.

Once predictions are obtained, they are **inverse transformed** to convert them back to their original scale. This is necessary because the data was initially scaled using the **MinMaxScaler** for better model performance during training.

```python
# Inverse transform predictions
def inverse_transform_single_feature(predictions, scaler, num_features=2):
    # Create an empty array with the same number of rows as predictions and columns matching num_features
    temp_array = np.zeros((predictions.shape[0], num_features))

    # Put the predicted values into the first column (for Adjusted Close)
    temp_array[:, 0] = predictions[:, 0]  # Assumes predictions is 2D (samples, 1)

    # Inverse transform using the scaler
    return scaler.inverse_transform(temp_array)[:, 0]  # Only return the first column (Adjusted Close)

# Rescale predictions for train, validation, and test sets
predictions_train_original = inverse_transform_single_feature(predictions_train, scaler)
predictions_valid_original = inverse_transform_single_feature(predictions_valid, scaler)
predictions_test_original = inverse_transform_single_feature(predictions_test, scaler)

# Rescale the actual values (y_train, y_val, y_test)
# Reshaping y_train, y_val, y_test before inverse transforming
y_train_reshaped = y_train.reshape(-1, 1)
y_val_reshaped = y_val.reshape(-1, 1)
y_test_reshaped = y_test.reshape(-1, 1)

actual_train_original = inverse_transform_single_feature(y_train_reshaped, scaler)
actual_valid_original = inverse_transform_single_feature(y_val_reshaped, scaler)
actual_test_original = inverse_transform_single_feature(y_test_reshaped, scaler)
```

Figure 8.20: Rescale Predicted Values to Original Scale

The inverse_transform_single_feature() function is used to rescale the predicted values as well as the actual stock prices. This ensures that the predicted and actual values are on the same scale, allowing for meaningful comparisons and further analysis.

	Metric	Training	Validation	Test
0	MSE	0.154271	0.622456	1.597258
1	MAPE	0.008848	0.012473	0.016400
2	R^2	0.996706	0.983048	0.973073

Figure 8.21: LSTM+Sentiment Model Performance Metrics

The model shows strong performance across all datasets. The **MSE** and **MAPE** values are low across the datasets showing minimal percentage error. The **R^2 scores** are very high, indicating that the model is successful in explaining 97% of the variance in the test data, indicating strong overall performance.

8.7.5. Model Prediction Performance Visualization

Below we analyze graphs of prediction vs each of [Training, Validation, Testing] datasets as well as an overall picture.

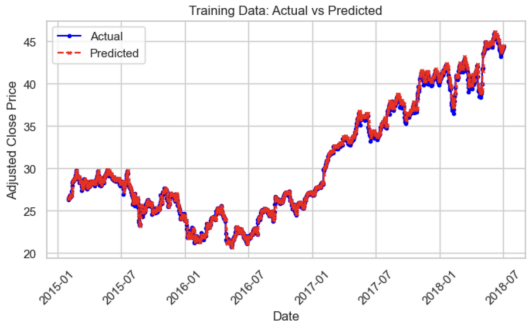


Figure 8.22: Training: Actual vs Prediction

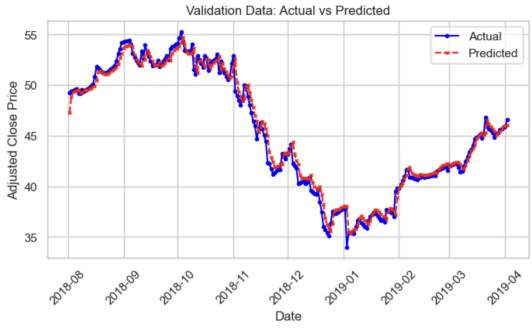


Figure 8.23: Validation: Actual vs Prediction

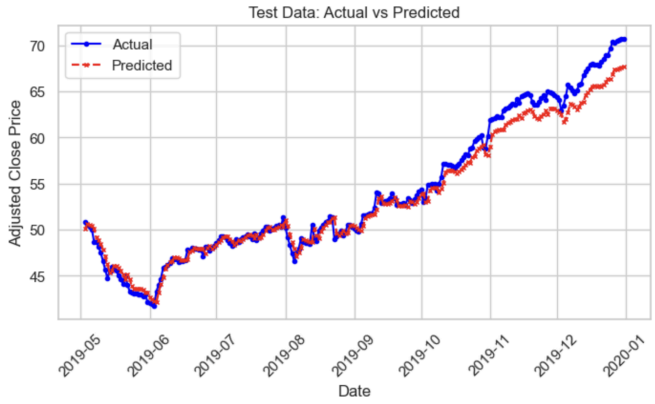


Figure 8.24: Testing: Actual vs Prediction

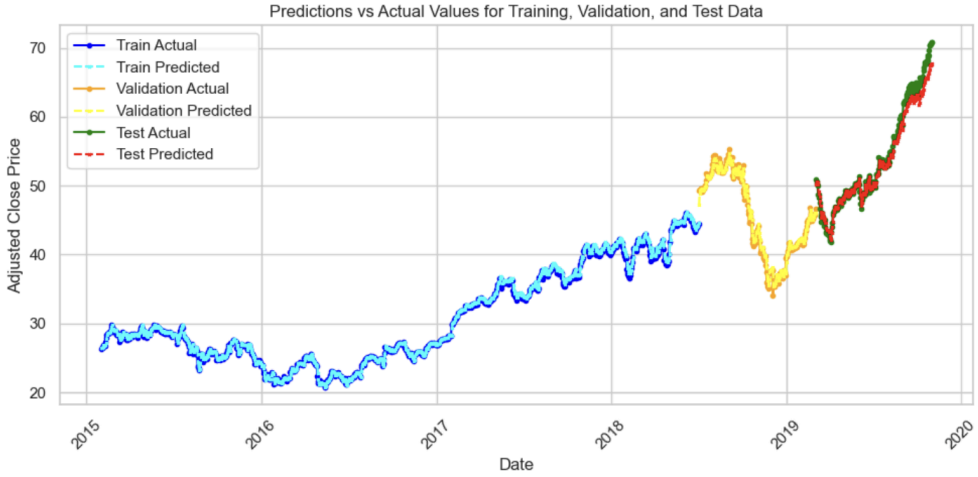


Figure 8.25: Actual vs Prediction Adj Close Price

Graphs above confirm a **considerably strong prediction performance by LSTM model** after incorporation of sentiment as a second feature in addition to adjusted close price. It generalized well to unseen data in general and gives high performance metrics.

8.8. Insight on hypothesis

Although the model performs well and is successful in capturing the complexities underlying the financial time series, incorporation of sentiment as an additional feature did not lead to any significant improvement in performance of original LSTM model doing prediction based on only the historic adjusted close price. This may be attributed to the analysis being conducted on older data; since the COVID-19 pandemic, social media activity has surged, potentially giving sentiment analysis more relevance in recent years. Additionally, the influence of real-time sentiment may play a larger role in predicting rapid market shifts in current trading environments.

9. Transformer

Transformer architecture is a powerful deep learning model that **leverages mechanisms such as self-attention** to help model complex relationships within sequential data. Unlike recurrent models, Transformers can handle long term dependencies more efficiently. Although its primary purpose is to analyze natural language, it can be tweaked to fit the complexities within financial time series. This makes transformers well-suited for time-series forecasting tasks such as stock price prediction.

Transformer proposed a self-attention mechanism with the core formula of as follows:

$$Attention(Q, K, V) = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

9.1. Feature Preparation

To improve the performance of the Transformer model, several technical indicators and features were calculated based on the stock price data. These features help to further capture essential patterns, trends, and volatility in the stock market, providing the model with valuable input to make accurate predictions.

1. **Daily Price Range (Width)**: The width is calculated as the difference between the daily high and low prices, providing a measure of the stock's daily volatility.

```
aapl_data['width'] = aapl_data['High'] - aapl_data['Low']  # Daily range (High - Low)
```

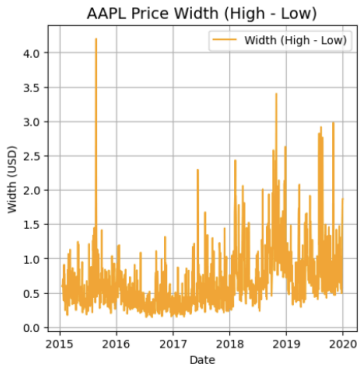


Figure 9.1: AAPL Price Width

The daily price width shows spikes in volatility, especially in late 2018, indicating periods of market uncertainty.

2. **Price Difference (Diff)**: The price difference represents the change in the closing price compared to the previous day. This feature highlights daily fluctuations in the stock price.

```
aapl_data['diff'] = aapl_data['Adj Close'].diff()  # Price difference
```

3. **Percentage Change (Pct Change)**: The percentage change indicates the rate of price movement, which normalizes the price difference by expressing it as a percentage of the previous day's price.

```
aapl_data['pct_change'] = aapl_data['Adj Close'].pct_change()  # Percentage p
```

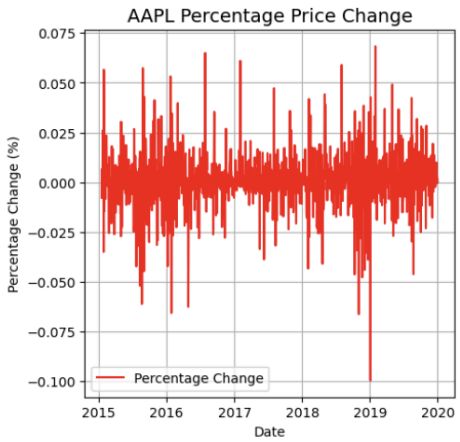


Figure 9.2: AAPL Percentage Price Change

This graph shows frequent daily volatility, with large swings especially during market downturns or recoveries, particularly in 2018 and 2019.

4. **Relative Strength Index (RSI)**: RSI is a momentum indicator that helps identify whether the stock is overbought or oversold. It is calculated using a 14-day window based on the ratio of recent price gains to losses.

```
# Relative Strength Index (RSI)
window_length = 14
delta = aapl_data['Adj Close'].diff()
gain = (delta.where(delta > 0, 0)).rolling(window=window_length).mean()
loss = (-delta.where(delta < 0, 0)).rolling(window=window_length).mean()
rs = gain / loss
aapl_data['RSI'] = 100 - (100 / (1 + rs))
```

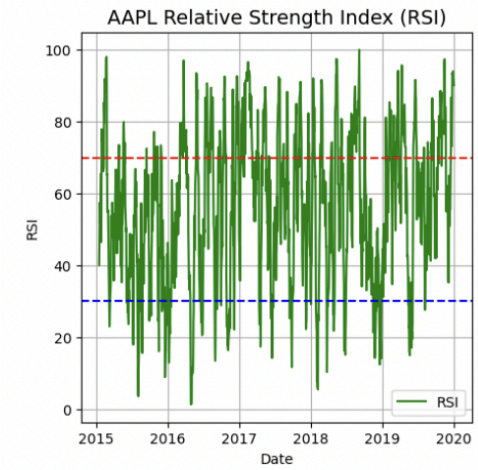


Figure 9.3: AAPL Relative Strength Index (RSI)

The RSI fluctuates mostly between 30 and 70, indicating that the stock was neither consistently overbought nor oversold but did approach these levels multiple times.

5. **Rate of Change (ROC)**: The ROC is another momentum indicator that measures the percentage change in price over a specific period (14 days). It helps detect shifts in stock price momentum.

```
# Rate of Change (ROC)
aapl_data['ROC'] = aapl_data['Adj Close'].pct_change(periods=window_length) * 100
```

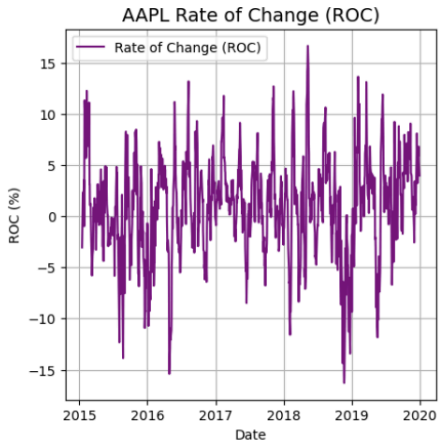


Figure 9.4: AAPL Rate of Change

The ROC indicates frequent sharp changes in price, reflecting rapid gains and losses, with noticeable spikes during market corrections.

Handling Missing Values: After generating these rolling features, any rows with missing values (generated due to the moving window) are removed to ensure a clean dataset for model training.

These features provide crucial insights into the stock's volatility, momentum, and price trends, allowing the Transformer model to make more informed predictions based on historical data.

9.2. Feature Selection

```
# Select features to be included in the model
features = ['Adj Close', 'width', 'RSI', 'ROC', 'Volume', 'diff', 'pct_change']
```

Figure 9.5: Feature Selection for Transformer

Above 7 features are used in prediction of stock prices movement.

9.3. Data Split and Data Normalization

```
# Define split percentages (e.g., 75% training, 15% validation, 15% test)
train_size = 0.7
val_size = 0.15

# Calculate indices for splitting
train_index = int(len(aapl_data) * train_size)
val_index = int(len(aapl_data) * (train_size + val_size))

# Split the data
train_data = aapl_data[features].iloc[:train_index]
val_data = aapl_data[features].iloc[train_index:val_index]
test_data = aapl_data[features].iloc[val_index:]
```

Figure 9.6: Input Data Split (75:15:15)

We divide the dataset into **training (70%), validation(15%), and test sets(15%)**. This ensures the model has sufficient data for learning while also preserving some unseen data for evaluation.

```
# Initialize the MinMaxScaler and fit only on the training data
scaler = MinMaxScaler(feature_range=(0, 1))

train_data_scaled = scaler.fit_transform(train_data)

# Transform the validation and test data using the same scaler
val_data_scaled = scaler.transform(val_data)
test_data_scaled = scaler.transform(test_data)
```

Figure 9.7: Input Data Scaling for Transformer

The **MinMaxScaler** is used to normalize the feature data to a range of 0 to 1. The scaler is fitted on the training data to prevent data leakage, and the same scaling is applied to the validation and test data, ensuring consistent scaling across all datasets.

9.4. Sequence Creation for Transformer Model

```
# Create sequences for Transformer input
def create_sequences_multifeature(data, sequence_length):
    X, y = [], []
    for i in range(len(data) - sequence_length):
        X.append(data[i:i + sequence_length])
        y.append(data[i + sequence_length, 0])  # Predicting the next 'Adj Close' price
    return np.array(X), np.array(y)

sequence_length = 30  # Using the last 30 days to predict the next day's price

# Create sequences for training, validation, and test sets
X_train, y_train = create_sequences_multifeature(train_data_scaled, sequence_length)
X_val, y_val = create_sequences_multifeature(val_data_scaled, sequence_length)
X_test, y_test = create_sequences_multifeature(test_data_scaled, sequence_length)
```

Figure 9.8: create_sequence function for Transformer

```
Training data shape: X_train: (1237, 30, 7), y_train: (1237,)
Validation data shape: X_val: (242, 30, 7), y_val: (242,)
Test data shape: X_test: (242, 30, 7), y_test: (242,)
```

Figure 9.9: Analyze input shape for Transformer

We prepare the input data for the transformer model using the function create_sequences_multifeature that generates sequences of input data for the Transformer model. It takes in a dataset and a sequence length (30 days) and constructs sequences for

prediction. The **input (X) consists of sequences of 30 days of data**, while the **target (y) is the adjusted closing price ('Adj Close') on the next day**. This process is applied to the training, validation, and test datasets to create the sequences needed to feed into the Transformer model.

9.5. Hyperparameter Tuning (Grid Search and Bayesian Optimization)

Within the transformer model, we have applied two hyperparameter tuning techniques: **Grid Search** and **Bayesian Optimization**. While both methods were useful, **Bayesian Optimization** proved to be more efficient in finding optimal hyperparameters with fewer evaluations, and also provided better results with less computational expense.

```
# Define the hyperparameter values to search over
d_model_values = [64, 128, 256]   # Model dimensions
num_heads_values = [2, 4, 8]      # Number of attention heads
dff_values = [128, 256, 512]      # Feed-forward network dimensions
batch_size_values = [32, 64]      # Batch sizes
num_features = len(features)

# Keep dropout fixed
dropout_rate = 0.2
# Dictionary to store results
results = []

# Loop over all combinations of hyperparameters
for d_model, num_heads, dff, batch_size in itertools.product(d_model_values, num_heads_values,
                                                             dff_values, batch_size_values):

    print(f"Training with d_model={d_model}, num_heads={num_heads}, dff={dff},
    batch_size={batch_size}")

    # Build the model with current hyperparameters
    transformer_model = build_transformer_model(sequence_length, num_features, d_model,
                                                num_heads, dff, dropout_rate)

    # Compile the model
    transformer_model.compile(optimizer='adam', loss='mean_squared_error')

    # Train the model
    history = transformer_model.fit(
        X_train, y_train,
        validation_data=(X_test, y_test),
        epochs=20,  # Train for fewer epochs to speed up the search
        batch_size=batch_size,
        verbose=0  # Suppress training output for readability
    )

    # Get the average training and validation losses from the last few epochs
    avg_train_loss = np.mean(history.history['loss'][-5:])
    avg_val_loss = np.mean(history.history['val_loss'][-5:])
    print(f"Validation loss : {avg_val_loss}")
    print("--------------------------------------------------------\n")

    # Store the results in the list
    results.append({
        'd_model': d_model,
        'num_heads': num_heads,
        'dff': dff,
        'batch_size': batch_size,
        'avg_train_loss': avg_train_loss,
        'avg_val_loss': avg_val_loss,
        'gap': abs(avg_train_loss - avg_val_loss)  # Difference between training and validation losses
    })
```

Figure 9.10: Hyperparameter Tuning: Bayesian Optimization

Through Bayesian optimization, we derived the **top 5 architectures** sorted by mean validation loss over the last 5 epochs.

	d_model	num_heads	dff	batch_size	loss
0	128	2	128	32	0.001993
1	128	8	512	32	0.003556
2	128	2	128	32	0.003588
3	128	8	128	32	0.004149
4	64	8	256	64	0.004667

Figure 9.11: Hyperparameter tuning: Best model architecture for Transformer

This helped decide the optimal architecture with **d_model** = **128**, **num_heads** = **2**, **dff** = **128** and a **batch size** = **32**.

9.6. Transformer Architecture

In this section, we describe the architecture of the Transformer model used for predicting stock prices, including the optimum parameters and explanations for each layer and component in the model. We have used the "vanilla transformer" with following structure -

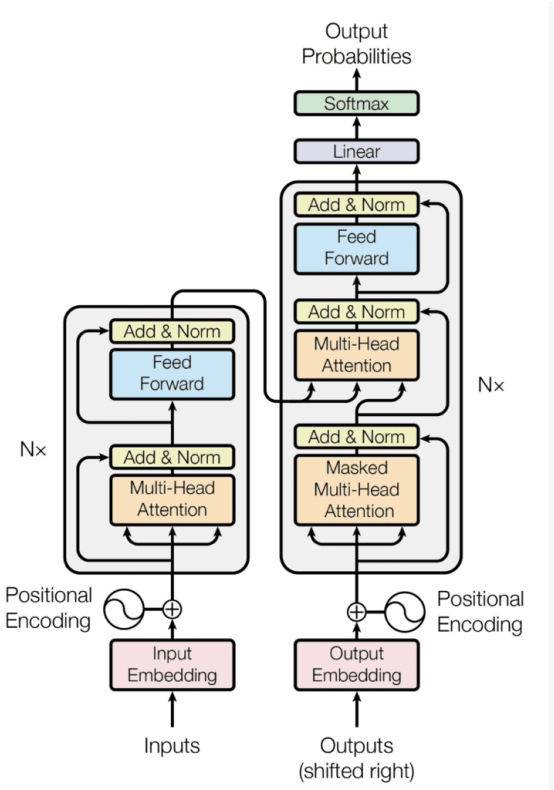


Figure 9.12: Transformer Architecture [3]

9.6.1. Optimum Parameters

- **Model Dimension (d_model):** 128
- **Number of Attention Heads (num_heads):** 2
- **Feed-forward Network Dimension (dff):** 128
- **Batch size**: 32
- **Dropout Rate:** 0.2
- **Number of Input Features:** 7 (Closing price, width, RSI, ROC, Volume, price difference and percentage change)

9.6.2. Layer-by-Layer Explanation

1. Input Layer:
   - **Shape:** (sequence_length=30, num_features=7)
   - Takes a sequence of 30 days of stock data, with 7 features, as input. These are projected into a d_model=128 dimensional space.
2. Positional Encoding
   - **Shape:** (sequence_length, d_model=128)

- Adds positional information to each time step in the sequence since the Transformer doesn't have sequential awareness on its own.

3. Multi-Head Attention:
   - **Heads:** num_heads=2
   - Allows the model to focus on different parts of the sequence simultaneously, learning the relationships across the 30-day period.
4. Add & Norm (Residual Connection):
   - Combines the output of the attention layer with the input and normalizes it, helping to stabilize training process using **LayerNormalization**.
5. Feed-Forward Network:
   - **Dimension:** dff=128
   - **Two dense layers** expand the data to 128 dimensions before reducing it back to d_model=128, helping the model learn complex representations.
6. Dense Output Layer:
   - Outputs a **single value prediction** (next day's Adjusted Close price) based on the last time step in the sequence.
7. Compilation:
   - **Optimizer:** Adam, **Loss Function:** Mean Squared Error (MSE)
   - The model is optimized using the Adam optimizer and MSE, commonly used for time series regression tasks.

This model leverages attention mechanisms to identify key patterns and relationships in stock data while the feed-forward layers capture complex nonlinear dynamics.

9.6.3. Position Encoding

```python
# Define function for positional encoding
def positional_encoding(position, d_model):
    angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                            np.arange(d_model)[np.newaxis, :],
                            d_model)
    # Apply sin to even indices and cos to odd indices
    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
    pos_encoding = angle_rads[np.newaxis, ...]
    return tf.cast(pos_encoding, dtype=tf.float32)

def get_angles(pos, i, d_model):
    angle_rates = 1 / np.power(10000, (2 * (i // 2)) / np.float32(d_model))
    return pos * angle_rates
```

Figure 9.13: positional_encoding function

Positional encoding is added to the input data to give the model a sense of order within the sequence. Since the transformer architecture doesn't understand the position of elements in the sequence on its own(like RNNs or LSTMs), this

encoding helps by introducing patterns (sine and cosine) that represent the positions.

9.6.4. Transformer Encoder

```python
#  Define the Transformer Model
def transformer_encoder(inputs, num_heads, dff, d_model, dropout_rate):
    attention = MultiHeadAttention(num_heads=num_heads, key_dim=d_model)
    attention_output = attention(inputs, inputs)

    # Add & Norm
    attention_output = LayerNormalization(epsilon=1e-6)(attention_output + inputs)

    # Feed-forward Network
    ff_output = Dense(dff, activation='relu')(attention_output)
    ff_output = Dense(d_model)(ff_output)

    # Add & Norm
    ff_output = LayerNormalization(epsilon=1e-6)(ff_output + attention_output)

    return ff_output
```

Figure 9.14: transformer_encoder function

We build one layer of the Transformer encoder. **Multi-head (2 heads) self-attention** is applied to learn relationships within the sequence, followed by a normalization step, which is then followed by a feed-forward network to further process the attention output. The model also uses skip connections (using layer normalization) to prevent information loss and stabilize training.

9.6.5. Transformer Model Creation

```python
# Transformer Model to include a Dense layer to match input features with d_model
def build_transformer_model(sequence_length, num_features, d_model, num_heads, dff, dropout_rate):
    inputs = Input(shape=(sequence_length, num_features))

    # Project the input features (num_features) into the
    # same dimension as the positional encoding (d_model)
    input_projection = Dense(d_model)(inputs)

    # Positional Encoding
    pos_encoding = positional_encoding(sequence_length, d_model)
    inputs_pos_encoded = input_projection + pos_encoding

    # Encoder layers
    encoder_output = transformer_encoder(inputs_pos_encoded, num_heads, dff, d_model, dropout_rate)

    # Output layer (regression task)
    outputs = Dense(1)(encoder_output[:, -1, :])  # Use the last time step for prediction

    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam', loss='mean_squared_error')

    return model
```

Figure 9.15: build_transformer_model function

This function builds the full transformer model. This is achieved by first projecting the input data into a higher dimensional space (matching d_model). Then we apply positional encoding, and pass the result through multiple layers of the transformer encoder. The final layer predicts the next day's stock price based on the learned sequence.

9.6.6. Early Stopping and Model Training

```
# Build and compile the model
transformer_model = build_transformer_model(sequence_length, num_features,
                                            d_model, num_heads, dff,
                                            dropout_rate)
```

```
# Train the model
early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
history = transformer_model.fit(X_train, y_train,
                                validation_data=(X_val, y_val),
                                epochs=100,
                                batch_size=32,
                                callbacks=[early_stop])
```

Figure 9.16: Compile and Train Transformer Model

After building the Transformer model, it is compiled with the Adam optimizer and Mean Squared Error (MSE) as the loss function. The model is then trained using the training dataset, with **early stopping** applied to halt training if no improvement is observed in the validation loss for 10 consecutive epochs. This prevents overfitting and ensures the best model is saved based on performance on validation dataset.

9.6.7. Training Loss vs Validation Loss

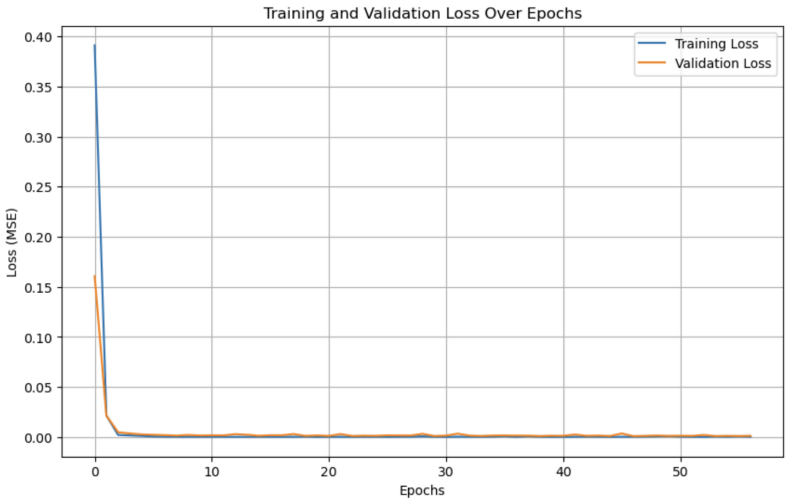


Figure 9.17: Training vs Validation Loss across epochs

The graph shows the training and validation loss (MSE) decreasing rapidly within the first few epochs, and stabilizing at a low value. This indicates that the **model quickly learns the relationships in the data and converges effectively**. The small gap between training and validation loss suggests minimal overfitting, indicating that the model generalizes well to unseen data.

9.6.8. Performance Metrics

```
# Make predictions on the test data
predictions = transformer_model.predict(X_test)

# Rescale the predictions and actual values back to original scale
predictions_rescaled = scaler.inverse_transform(np.concatenate([predictions, np.zeros((predictions.shape[0], len(features)-1))], axis=1))[:,0]
y_test_rescaled = scaler.inverse_transform(np.concatenate([y_test.reshape(-1, 1), np.zeros((y_test.shape[0], len(features)-1))], axis=1))[:,0]
```

Figure 9.18: Predictions using transformer model

The predictions on the test data are rescaled back to the original scale for comparison with the actual values. This is to ensure that both predicted and actual stock prices are in their original format, allowing for an accurate evaluation of the model's performance on the test set.

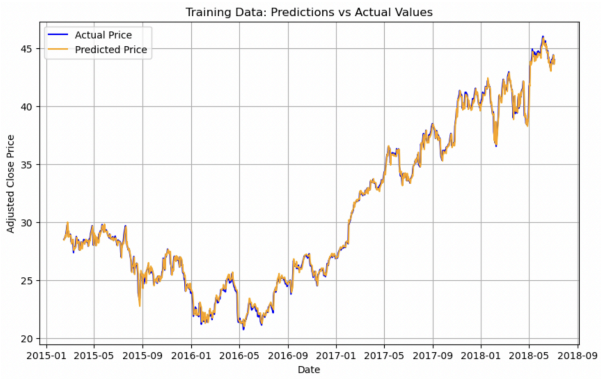
	Metric	Training Data	Validation Data	Test Data
0	MSE	0.116749	0.656560	0.508978
1	MAPE	0.007845	0.012908	0.010358
2	R^2	0.997542	0.982176	0.991449

Figure 9.19: Transformer Model Performance Metrics

The model demonstrates excellent performance on the training data with very low MSE and MAPE scores and an R^2 score of 0.997, indicating a nearly perfect fit. The validation and test data show slightly higher error values, as expected but the R^2 scores nearly 1 for both indicate s**trong generalization and predictive accuracy across unseen data, showcasing the model's robustness**.

9.6.9. Model Prediction Performance Visualization

Below we see the visualizations of forecast predictions over [train, validation, and test datasets].



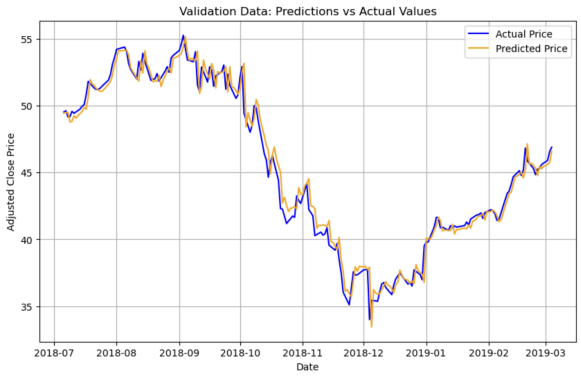


Figure 9.20: Training: Predictions vs Actual *Figure 9.21: Validation: Prediction vs Actual*

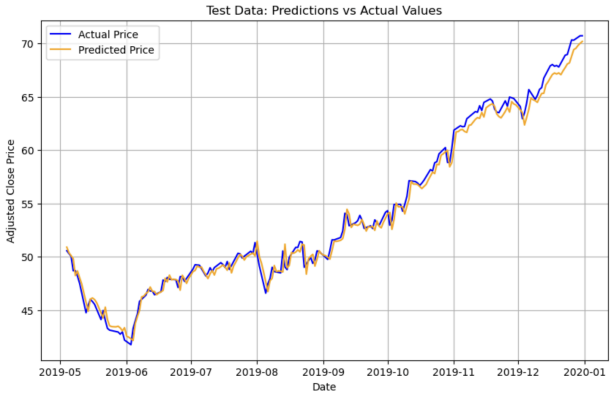


Figure 9.22: Testing: Prediction vs Actual

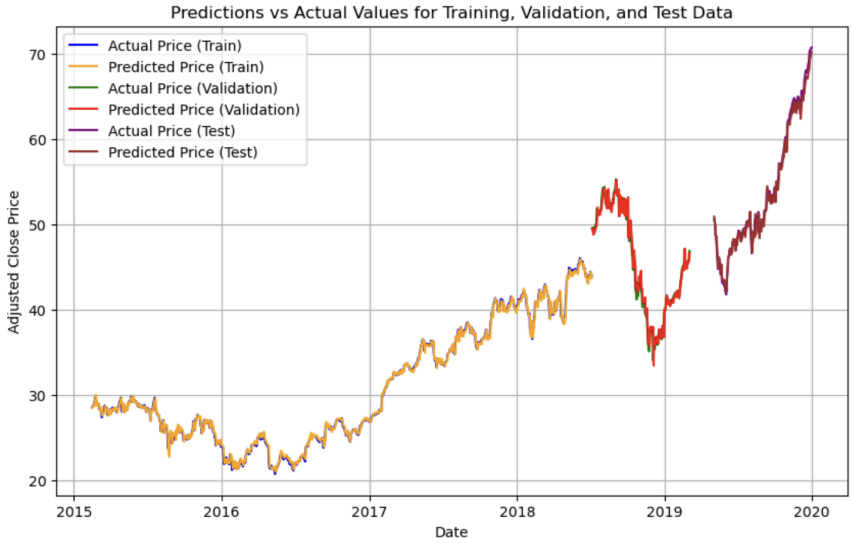


Figure 9.23: Prediction vs Actual Adj Close Price

9.7. Additional Analysis

9.7.1. Transformer Model on Adjusted Close Price

To maintain consistency in the evaluation and comparison of models like ARIMA and LSTM, we performed predictions using the **Transformer model with only the "Adjusted Close Price" feature**.

This analysis focused on utilizing a single feature—the stock's adjusted close price—allowing us to benchmark the model's performance against other models that we have also trained on this feature. The results from this single-feature analysis provide insights into the Transformer's ability to capture price patterns based solely on closing price trends.

1. Optimal Architectures using Bayesian Optimization

	d_model	num_heads	dff	batch_size	loss
0	64	8	128	64	0.003333
1	128	8	512	32	0.003890
2	128	4	512	32	0.005796
3	64	4	512	32	0.006988
4	128	4	512	64	0.008126

Figure 9.24: Top Architectures: Transformer Model (1 feature)

The table presents the top 5 architectures found through Bayesian Optimization, showing a range of different hyperparameter configurations.
Notably, the architecture with **64 units for d_model, 8 attention heads, and 128 units for dff** achieved the lowest validation loss, indicating strong performance with relatively moderate model complexity.

2. Performance Evaluation Metrics

	Metric	Training Data	Validation Data	Test Data
0	MSE	0.118643	0.565707	0.511887
1	MAPE	0.007439	0.011258	0.010247
2	R^2	0.997461	0.984602	0.991343

Figure 9.25: Performance Metrics: Transformer Model (1 feature)

The model shows strong performance across training, validation, and test sets with a high **R^2 score** of **0.991** on the test data, indicating an excellent fit.

Compared to the transformer model with 7 features, this performance suggests that both approaches are yielding highly accurate predictions, though additional features may introduce complexity that marginally improves performance depending on the dataset.

3. Performance Visualization

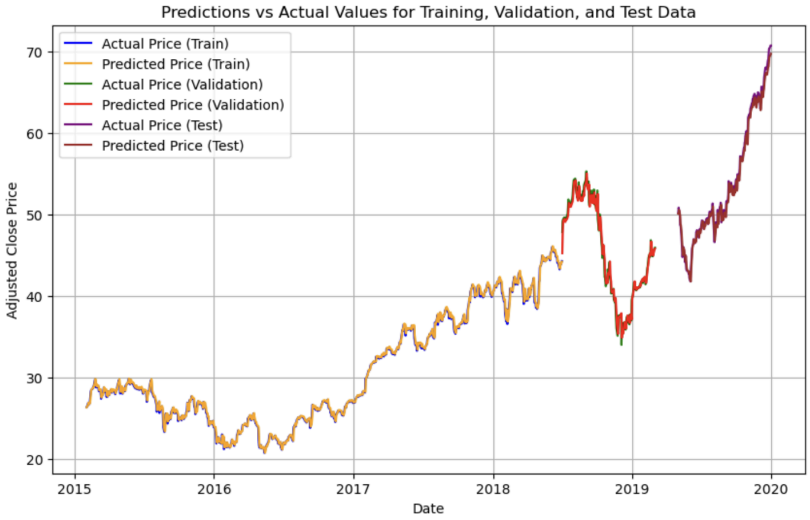


Figure 9.26: Prediction vs Actual Adj Close Price: Transformer Model (1 feature)

The above plot shows that the transformer model closely models the relationships of the actual stock prices in all sets, with minimal deviation. It is successful in effectively capturing overall trends and key inflection points.

10. GPT-3

GPT-3 (Generative Pretrained Transformer 3) is a large-scale language model trained by OpenAI. It is mainly known for its language processing abilities. However, its flexible architecture enables us to modify it and to use it for diverse tasks, including time series prediction, when prompted correctly.

10.1. Data Retrieval and Preparation

To maintain consistency, we will be using historical stock price data for AAPL using the yfinance. We will be using the test data between 2019-01-01 to 2019-12-31 as before since it is a pre-trained model.

```
# Initialize OpenAI API key
openai.api_key = key

# Retrieve historical data using yfinance
ticker = "AAPL"
start_date = "2019-01-01"
end_date = "2019-12-31"

# Fetch the historical data
aapl_data = yf.download(ticker, start=start_date, end=end_date)

# Drop any NaN values (just to be sure)
aapl_data.dropna(inplace=True)
```

Figure 10.1: Input Data Preparation for GPT-3

The OpenAI API key is initialized, which will be used later for making predictions via GPT-3. Any missing or NaN values in the dataset are removed to ensure data integrity before feeding it into the model.

10.2. GPT-3 Prompt Creation

To enable the GPT-3 model to make predictions based on historical adjusted close price of stocks, we create a function to convert the data into a textual format the model can interpret.

```
# Function to create GPT prompt
def create_gpt_prompt(data, days=30):
    """Creates a textual representation of stock data for GPT input."""
    prompt = "Below is the daily adjusted closing price of AAPL for the past 30 days. \
    Based on this data, predict the closing price for the next day as a number only, no explanation:\n\n"
    for i in range(days):
        date = data.index[i].strftime('%Y-%m-%d')
        adj_close = data['Adj Close'].iloc[i]
        prompt += f"Day {i+1} ({date}): Adj Close: {adj_close:.2f}\n"
    prompt += "\nPredict the adjusted close price for Day 31 as a numerical value."
    return prompt
```

Figure 10.2: create_gpt_prompt function

The code above generates a prompt from the most recent 30 days of adjusted close price of AAPL stock. The prompt attempts to generate a human-readable conversation that GPT-3 responds to with a prediction. The figure below illustrates an example input prompt.

```
Example GPT Prompt:

Below is the daily adjusted closing price of AAPL for the past 30 days. Based on this data, predict th
e closing price for the next day as a number only, no explanation:

Day 1 (2019-11-15): Adj Close: 64.48
Day 2 (2019-11-18): Adj Close: 64.81
Day 3 (2019-11-19): Adj Close: 64.61
Day 4 (2019-11-20): Adj Close: 63.86
Day 5 (2019-11-21): Adj Close: 63.57
Day 6 (2019-11-22): Adj Close: 63.52
Day 7 (2019-11-25): Adj Close: 64.63
Day 8 (2019-11-26): Adj Close: 64.13
Day 9 (2019-11-27): Adj Close: 64.99
Day 10 (2019-11-29): Adj Close: 64.84
Day 11 (2019-12-02): Adj Close: 64.10
Day 12 (2019-12-03): Adj Close: 62.95
Day 13 (2019-12-04): Adj Close: 63.51
Day 14 (2019-12-05): Adj Close: 64.44
Day 15 (2019-12-06): Adj Close: 65.68
Day 16 (2019-12-09): Adj Close: 64.76
Day 17 (2019-12-10): Adj Close: 65.14
Day 18 (2019-12-11): Adj Close: 65.70
Day 19 (2019-12-12): Adj Close: 65.87
Day 20 (2019-12-13): Adj Close: 66.76
Day 21 (2019-12-16): Adj Close: 67.90
Day 22 (2019-12-17): Adj Close: 68.04
Day 23 (2019-12-18): Adj Close: 67.88
Day 24 (2019-12-19): Adj Close: 67.94
Day 25 (2019-12-20): Adj Close: 67.80
Day 26 (2019-12-23): Adj Close: 68.91
Day 27 (2019-12-24): Adj Close: 68.97
Day 28 (2019-12-26): Adj Close: 70.34
Day 29 (2019-12-27): Adj Close: 70.32
Day 30 (2019-12-30): Adj Close: 70.73

Predict the adjusted close price for Day 31 as a numerical value.
```

Figure 10.3: GPT-3 Prompt Example

10.3. Model Prediction Approach

We leverage GPT-3 model to predict stock prices for AAPL based on 30 day historical data and evaluate the model's performance using backtesting. This is done by iteratively generating predictions using GPT-3 and then comparing these predictions against actual stock prices.

```python
# Using 30 days of data for each prediction
lookback_days = 30

for i in range(lookback_days, len(aapl_data)):
    historical_data = aapl_data.iloc[i - lookback_days:i]
    actual_value = aapl_data['Adj Close'].iloc[i]
    actual_date = aapl_data.index[i]
    actual_values.append(actual_value)
    dates.append(actual_date)

    # Create GPT prompt
    gpt_prompt = create_gpt_prompt(historical_data)

    # Make prediction using GPT-3.5 (chat-based model)
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[
            {"role": "system", "content": "You are a financial stock market expert."},
            {"role": "user", "content": gpt_prompt}
        ],
        max_tokens=10,
        temperature=0.2
    )

    # Extract GPT prediction and handle non-numeric cases
    try:
        predicted_value = float(response['choices'][0]['message']['content'].strip())
    except ValueError:
        print(f"Invalid GPT response: {response['choices'][0]['message']['content']}")
        predicted_value = np.nan  # Handle invalid response by setting NaN
    predictions.append(predicted_value)

    print(f"Date: {actual_date.strftime('%Y-%m-%d')}, Actual: {actual_value:.2f},
    GPT Prediction: {predicted_value:.2f}")
```

Figure 10.4: GPT-3 Prediction Setup

10.3.1. Model Used

gpt-3.5-turbo is being used for the prediction task, which is optimized for chat-based tasks.

10.3.2. Chat Messages Setup

1. Define the context of conversation using the messages argument.
2. The **system** message sets the expertise of the model. We prompt it to act as the **financial stock market expert** to help the model to respond in the context of stock market analysis.
3. The **user** message contains the **gpt_prompt** which is the prompt we input telling GPT-3 to make a prediction for next day's adjusted close price based on 30 day historic data.

10.3.3. Response Limitation

We set max_tokens=10 ensuring only the numerical predicted value is returned without any additional text or explanation

10.3.4. Temperature Setting

We define the temperature=0.2 to be able to control the randomness of GPT's predictions. This is useful for tasks like stock price prediction, where precision is important.

10.4. Performance Metrics

GPT-3's performance is evaluated by comparing the predicted price vs the actual price using performance metrics such as (R^2, MSE, and MAPE).

	Metric	Value
0	MSE	0.708104
1	MAPE	0.012459
2	R^2	0.987729

Figure 10.5: GPT-3 Performance Metrics

As seen above, GPT-3 performs commendably, with s**cores at par with other advanced models such as LSTM and Transformers**. The MSE and R^2 scores suggest that the model successfully captures the trends in stock price and models the relationships in historical price data with **low error rates**.

10.5. Model Prediction Performance Visualization

We will now visualize the Predictions vs Actual Adjusted price on the test data.

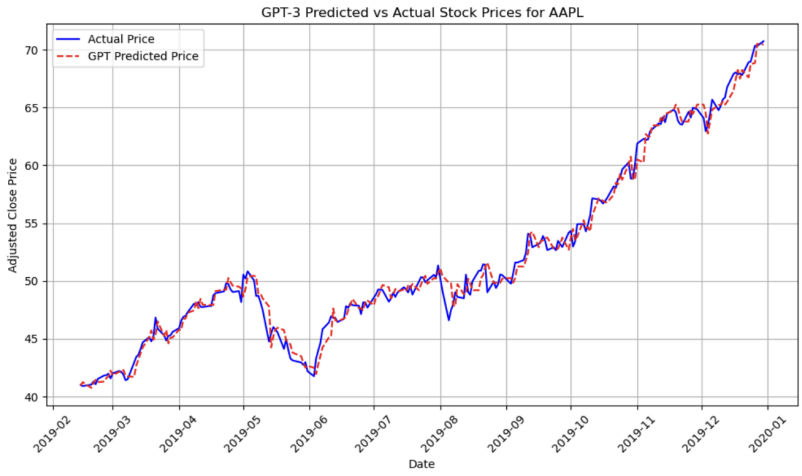


Figure 10.6: GPT-3 Performance Visualization: Actual vs Predicted Adj Close

As seen in the figure above, GPT-3 model adapts well, and successfully captures the historical relationships between adjusted close price.

10.6. Additional Analysis - Incorporating Moving Averages and RSI with GPT-3

In this section, we have conducted an additional analysis to test whether incorporating technical indicators, such as **Moving Averages (MA)** and the **Relative Strength Index (RSI)**, would improve GPT-3's predictive performance for stock prices. The features used in this extended analysis are:

- **MA5**: A 5-day moving average to smoothen short-term fluctuations highlighting recent price trends.
- **MA30**: A 30-day moving average to provide a longer-term view of the stock's price trend.
- **RSI**: It is used to indicate overbought or oversold conditions in a stock by measuring the movements.

```
# Create additional features: Moving averages and RSI
aapl_data['MA5'] = aapl_data['Adj Close'].rolling(window=5).mean()
aapl_data['MA30'] = aapl_data['Adj Close'].rolling(window=30).mean()

# RSI Calculation
window_length = 14
delta = aapl_data['Adj Close'].diff()
gain = (delta.where(delta > 0, 0)).rolling(window=window_length).mean()
loss = (-delta.where(delta < 0, 0)).rolling(window=window_length).mean()
rs = gain / loss
aapl_data['RSI'] = 100 - (100 / (1 + rs))
```

Figure 10.7: Additional Features MA and RSI

We used these additional features and got predictions using a similar process of prompting the GPT-3 model as used above. Following this we got the following performance metrics scores

	Metric	Value
0	MSE	0.771404
1	MAPE	0.012764
2	R^2	0.985325

Figure 10.8: GPT-3 with additional features - Performance Metrics

Insight - As seen above, despite incorporating additional features into the analysis, the performance metrics remained consistent with earlier analysis, indicating no significant improvement in the model's accuracy. This showcases that adding additional technical indicators such as these has no substantial effect on GPT-3's predictive capability for this particular stock price prediction.

10.7. Additional Analysis: Evaluating GPT-3 on Recent Data (2024-01-01 to 2024-10-01)

To further evaluate the robustness of the GPT-3 model, we conducted an **additional analysis using more recent stock data**.
The purpose of this analysis was to test whether the model's high performance on historical data from 2019 was due to some form of overfitting or memorization. By using unseen and recent stock data, we aim to verify if the model could maintain similar levels of performance.

	Metric	Value
0	MSE	12.452401
1	MAPE	0.012679
2	R^2	0.974840

Figure 10.9: GPT-3 with 2024 data - Performance Metrics

On conducting the prediction, we found the performance metrics as above. We can therefore conclude the model upholds its performance on recent data as well. It manages to achieve **R^2 score of 0.974**, which can generalize well to unseen data, and model the underlying relationship within the time series data. The low MAPE and MSE scores show low error margins.

Insights - The overall performance on recent data suggests that GPT-3's predictive power is not only good on older data but maintains its **high performance for unseen and up-to-date stock price data**. This makes GPT-3 a viable tool for stock price forecasting in the daily trading sector indicating that its performance is not limited to some specific period and its predictive capabilities are not due to potentially overfitting on historical trends.

11. Comparative Model Performance

To evaluate the effectiveness of the models, we compared the performance of ARIMA, LSTM, LSTM + Sentiment, Transformer models (using Adjusted Close price and multi-feature inputs) and GPT-3 model using three key metrics:

11.1. MSE

The ARIMA model displayed significantly higher MSE values, indicating poor predictive performance. In contrast, all LSTM and Transformer models exhibited much lower MSE values, with Transformer models (especially with 7 input features) achieving the lowest MSE on the test data. The GPT model performed comparably to the Transformer model.

11.2. MAPE

ARIMA performed poorly with a high MAPE value, showing it struggled to accurately predict stock price changes in percentage terms. On the other hand, LSTM and Transformer models excelled with near-zero MAPE, reflecting their ability to minimize prediction errors effectively. GPT also showed similar performance to Transformer models, further validating its use in stock price prediction.

11.3. R^2 Score

ARIMA's negative R^2 score suggests it failed to capture any meaningful variance in the data. LSTM, Transformer and GPT models, however, demonstrated near-perfect performance, with R^2 values close to 1.0 on both training and test data, highlighting their robustness and consistency.

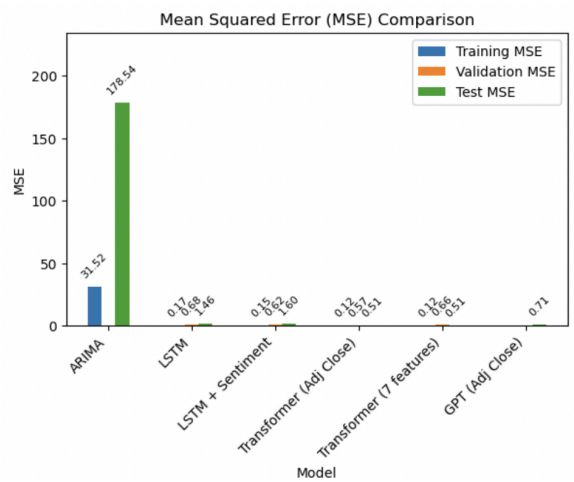


Figure 11.1: Model Comparison:MSE

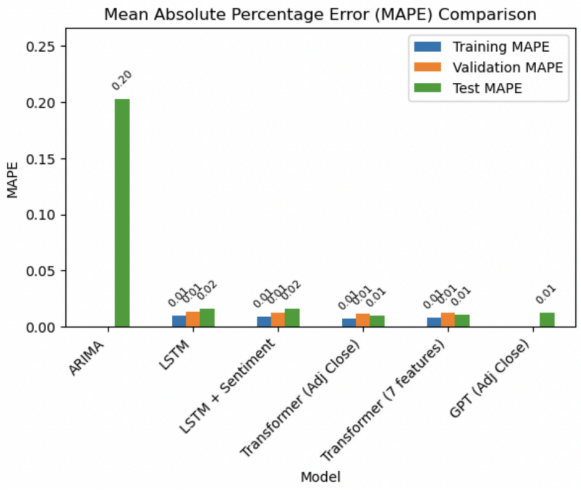


Figure 11.2: Model Comparison: MAPE

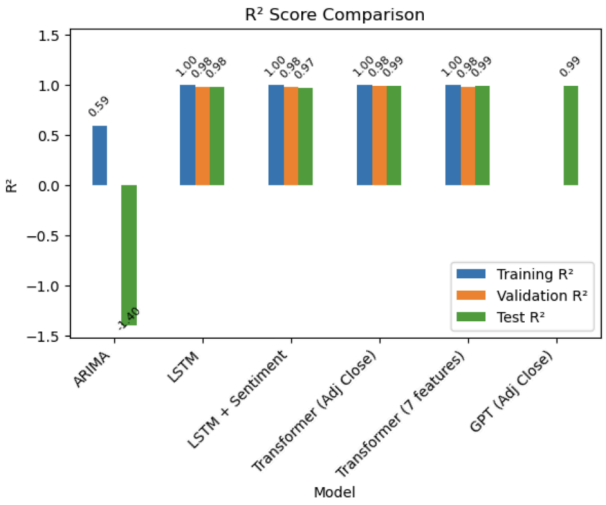


Figure 11.3: Model Comparison: R²

The above graphs clearly illustrates that the **Transformer models**, especially with multi-feature inputs, deliver the best overall performance across all metrics. These models significantly outperform ARIMA and show marginal improvements over LSTM models, making them the preferred choice for stock price prediction. While GPT also performed well, it was slightly behind the multi-feature Transformer in terms of overall performance.

Insight: The addition of sentiment analysis to the LSTM did not yield a substantial improvement. This may however be due to analyzing older data. Incorporating real-time sentiment from recent tweets could potentially have a greater influence on stock price prediction. Additionally, GPT-3s exceptional performance to adapt to financial data highlights its flexibility even though its main purpose is natural language processing.

12. Conclusion

In this project, we have explored the effectiveness of various machine learning models in predicting stock prices. This was done specifically for Apple Inc. (AAPL) over a period from 2015 to 2019. Models such as ARIMA, LSTM, LSTM integrated with sentiment analysis, and Transformer models were evaluated based on three key performance metrics: Mean Squared Error (MSE), Mean Absolute Percentage Error (MAPE), and R² score. Through comprehensive analysis, we observed that traditional models like ARIMA underperformed significantly, exhibiting high error rates and poor variance explanation on both training and test data. This is because it is not able to model the underlying complexity of the stock prices.

While the LSTM + Sentiment model slightly improved over basic LSTM, the addition of sentiment analysis did not yield a significant difference, highlighting that sentiment alone may not have a profound impact on stock price predictions. However, LSTM models, both with and without sentiment features, demonstrated substantial improvements in predictive accuracy, with near-zero MAPE values and R^2 scores approaching 1.

The Transformer model, particularly when fed with multiple features (Adjusted Close price, RSI, ROC, etc.), emerged as the most robust approach, achieving the lowest MSE and the highest R^2 scores on the test dataset. This highlights the potential of deep learning and attention-based mechanisms like Transformers in handling complex, multi-feature time-series forecasting tasks.

We also saw the exceptional performance of GPT-3 to adapt to financial data even though it was primarily built to process natural language. It shows its flexibility and versatility to be able to compete with other advanced models such as LSTM and Transformers.

Such analysis has potential practical applications for improving the accuracy of stock price predictions, which can be highly beneficial in daily trading strategies and portfolio management. By leveraging advanced models like Transformers, investors and traders can make more informed decisions, particularly in short-term trading where capturing daily price fluctuations is crucial.

13. Future Work

While the results we obtained using the transformer model are promising, there are still several opportunities that could be explored to refine the prediction analysis.

13.1. Incorporating Informer Model
Informer model was specifically designed for forecasting long sequence time series. It uses ProbSparse Attention mechanism for better handling of long-range dependencies. It is also more computationally efficient. [7]

13.2. Extended Forecast Horizon
Instead of predicting the prices for only the next day, future models could focus on prediction over longer horizons such as a week, or a month. This would be useful for portfolio management.

13.3. Advanced Feature Integration
It can be useful to include macroeconomic factors like market volatility indices or even real time sentiment analysis. This can help models to better capture underlying complexities behind the time series and enhance predictive accuracy.

14. References

[1]G. Edward and Gwilym Meirion Jenkins, *Time series analysis : forecasting and control*. San Francisco, Fl: Holden-Day, 1970.

[2]T. Fischer and C. Krauss, "Deep learning with long short-term memory networks for financial market predictions," *European Journal of Operational Research*, vol. 270, no. 2, pp. 654–669, Oct. 2018, doi: https://doi.org/10.1016/j.ejor.2017.11.054.

[3] A. Vaswani *et al.*, "Attention Is All You Need," Jun. 2017. Available: https://arxiv.org/pdf/1706.03762

[4] T. H. Nguyen, K. Shirai, and J. Velcin, "Sentiment analysis on social media for stock movement prediction," *Expert Systems with Applications*, vol. 42, no. 24, pp. 9603–9611, Dec. 2015, doi: https://doi.org/10.1016/j.eswa.2015.07.052.

[5] A. Lopez-Lira and Y. Tang, "Can ChatGPT Forecast Stock Price Movements? Return Predictability and Large Language Models," *SSRN Electronic Journal*, 2023, doi: https://doi.org/10.2139/ssrn.4412788.

[6] R. T. J. J, "LSTMs Explained: A Complete, Technically Accurate, Conceptual Guide with Keras," *Analytics Vidhya*, Sep. 10, 2021. https://medium.com/analytics-vidhya/lstms-explained-a-complete-technically-accurate-conceptual-guide-with-keras-2a650327e8f2

[7]H. Zhou *et al.*, "Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting," *arXiv (Cornell University)*, Dec. 2020, doi: https://doi.org/10.48550/arxiv.2012.07436.

15. Appendix

Project Structure

The project directory contains the following key files and folders:

- **Model Notebooks**:

1. AAPL Sentiment Analysis.ipynb: Sentiment analysis on Apple-related tweets.
2. ARIMA.ipynb: ARIMA model for stock price prediction.
3. Comparative Analysis.ipynb: Comparison of model performances using metrics.
4. LSTM.ipynb: LSTM-based stock prediction model.
5. LSTM + Sentiment.ipynb: LSTM with sentiment analysis for stock prediction.

6. Final Transformers (Adj Close).ipynb: Final Transformer model using Adj Close Price feature for prediction.
7. Final Transformers F7.ipynb: Final Transformer model using 7 features for prediction.
8. GPT-3 Stock Prediction (Adj Close).ipynb: GPT-3 model using only Adjusted Close Price for prediction
9. GPT-3 Stock Prediction F4.ipynb: GPT-3 model using Adjusted Close Price, MA(5), MA(30) and RSI for prediction
10. GPT-3 Stock Prediction- 2024.ipynb: GPT-3 model Adjusted Close Price prediction for 2024 test data.

- **Datasets**:

1. AAPL_Tweets_with_Sentiment.csv: Apple tweets with sentiment scores.
2. AAPL_Tweets.csv: Raw Apple-related tweets data.
3. Tweet.csv: General Twitter dataset related to financial tweets.

- **Model Weights and Versions**:

1. The folder Final Transformer Model - 7 features contains Transformer model weights and architecture for further analysis.
2. The folder Transformer Model - 1 feature contains Transformer mode trained using only 1 feature's weights and architecture for further analysis.
3. The folder GPT-3 Predictions contains the actual and predicted scores using the GPT-3 model.