

# 

**Final Project Report**

# 0. Contents

0. Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.1 Project Definition

1.2 Breakdown of Goals

2. Overall Summary of Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1 Executive Summary

3. Data Preprocessing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

3.0 Summary

3.0.1 Data Attributes and Source

3.0.2 Techniques Used

3.1 Data Acquisition

3.1.1 Yahoo! Finance Stock Dataset

3.1.2 Stock and Time Frame Selection

3.1.3 Data Visualization and Distribution

3.2 Addressing Missing Values

3.2.1 Drop N/A Values

3.2.2 Filling in N/A Values with Statistical Data

3.3 Unskewing Data

3.3.1 Standard Scaler

3.3.2 Min-Max Scaler

3.4 Feature Engineering

3.4.1 Moving Averages

3.4.2 Lagging Prices

3.4.3 Other Technical Indicators

3.5 Train Test Split

Test Set Size and Methodology

4. Pipeline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

4.1 Feature Selection

4.1.1 Closing Prices

4.1.2 Correlation Heat Map

4.2 Loss Metrics

4.2.1 Mean Absolute Error (MAE)

4.2.2 Mean Squared Error (MSE)

4.2.3 Root Mean Squared Error (RMSE)

5. Model Specifics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

5.0 Summary

5.0.1 Model Training

5.0.2 Model Predictions

5.1 Base Model Exploration

5.1.1 Linear Regression

5.1.2 Random Forests

5.1.3 XGBoost Regressor

5.2 Neural Networks

5.2.1 Simple RNN

5.2.2 LSTM 1

5.2.3 LSTM 2

5.2.4 LSTM 3

5.2.5 MLP

5.2.6 Transformers

6. Findings and Predictions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

6.0 Summary

6.0.1 Best Model and Predictions

6.0.2 Overview of Same Stock Future Price Predictions

6.0.2 Overview of Same Industry Predictions

6.0.3 Overview of Different Industry Predictions

6.1 Individual Results

6.1.1 Simple RNN

6.1.2 LSTM 1

6.1.3 LSTM 3

6.1.4 Transformer

6.1.5 Baseline Models

6.1.6 LSTM 2

6.1.7 MLP

7. Conclusion and Future Considerations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

7.1 Conclusion

7.2 Future Considerations

# 

# 1. Introduction

## 

## 1.1 PROJECT DEFINITION

Our project focuses on historical stock data and we wanted to know which model can best predict the future closing price of a stock. We also explored how well these models can perform on other stocks within the same industry or outside in other industries as the original stock that the model was trained on.

The motivation behind this project is that the stock market is a huge source of information and data that gets expanded every day and has implications in people’s everyday lives from consumers, traders, to businesses. Being able to predict the future prices of stocks would aid in trading and financial well being.

This topic was inspired by a notebook reviewed in class relating to stock data as well as other Kaggle notebooks, datasets, and external references (See ***Section 9*** for references).

## 1.2 BREAKDOWN OF GOALS

In completing this project, we aimed to do the following:

1. Picked industries and groups of stocks to focus on
2. Train a few models on the stock to see which model performs best
3. Using the models trained above, predict the future closing prices of one of several stocks in the same industry.
4. Using the same models, predict the closing price of one of several stocks in a different industry
5. Compare the prediction results and determine which model was the best at predicting the future closing prices of stocks and see how the models performed on stocks inside and outside of their respective industries.

# 

# 2. Overall Summary of Approach

## 2.1 EXECUTIVE SUMMARY

In general, looking at the table of contents in ***Section 0*** will give a good overview of what was done in this project.

This project focuses on predicting the future closing prices of stocks within different industries using various models. We chose to complete this project because we were all interested in the stock market behaviour, as predicting stock market data may help with financial wellbeing and is something that is a part of our everyday life. In addition, the stock market holds large amounts of data, which is perfect for a machine learning and data science project. See ***Section 1*** for a deeper description of this project.

Our data came from the Yahoo! Finance Dataset, which was readily integrated with the finance python library. We focused on several select stocks within several select industries we deemed stable yet offering enough variability in prices to train our models with. As for data preprocessing, N/A values were dropped and our data was unskewed using Standard Scaler and Min-Max Scaler. Additional features such as moving averages were created through feature engineering. We had no categorical data, hence no encoding was needed. See ***Section 3*** for more details. For the exact code used, see ***Section 8***.

In order to find out which models would perform the best on predicting the future closing prices of stock data, we tried several different models including Linear Regression, Random Forest, XGBoost Regressor, RNN, LSTM, MLP, and Transformers. See ***Section 5*** for more details about the models themselves and how they were trained. However, prior to training our models, we performed feature selection and also explored different loss metrics including MSE and MAE to evaluate the effectiveness of our models. See ***Section 4*** for more details.

In conclusion, our results (See ***Section 6*** for a detailed description of our findings) revealed that models performed the best when predicting the future closing price of a the same stock that it is trained on, suggesting that the best results can be achieved by training a model individually for each stock and having that model predict the future closing prices for that one stock rather than attempting to train a model that can be generalized to other stocks. Linear Regression and the MLP model performed the best in predicting future closing prices of a stock, but all models suffered when predicting other stocks that it had never seen before, regardless if the new stock was in a similar industry or not. Nevertheless, future considerations include trying different methodology and incorporating additional feature engineering. See ***Section 7*** for more details.

Any cited sources or references used can be found in ***Section 9***.

# 3. Data Preprocessing

## 

## 3.0 SUMMARY

### 3.0.1 Data Attributes and Source

We used the yfinance python library to pull historical stock data from the Yahoo! Finance website (See ***Section 3.1*** for more details). For each stock, this dataset includes the open price, high price, low price, close price, volume, dividends, and stock splits. There are about 250 entries of daily stock data per year since the stock market is not open all 7 days in a week.

We focused on the time period between 2000 to just before 2020 to avoid the effects of the covid pandemic. Daily stock data was used instead of intraday stock data since it was more stable and less noisy (See ***Section 3.1.2*** for more details). We chose to look at the following industries and stocks:

**Table 3.0.1-A:** Stocks Chosen for Study in This Project

|  |  |  |  |
| --- | --- | --- | --- |
| Technology | Healthcare | Construction | Food |
| NVDA  GOOG  AAPL | LLY  ABBV  MRK | TT  CRH  URI | ADM  KHC  GIS |

### 3.0.2 Techniques Used

We checked for and dropped any missing or empty data entries and also scaled our data using the standard scaler and the min-max scaler. As for feature engineering, we calculated moving averages including the 7 day moving average and 21 day moving averages. We also created features with previous day prices, ranging from yesterday’s price to prices from 5 days ago. (See ***Section 3.4*** for more details on feature engineering).

## 3.1 DATA ACQUISITION

### 

### 3.1.1 Yahoo! Finance Stock Dataset

We used the yfinance python library to download historical stock data directly from the Yahoo! Finance website. The Yahoo! Finance website contains all historical stock data as recent as the current last open market day to as far back before some of the stocks existed. Most stocks were not available before 1970. The yfinance python library scrapes data directly from the Yahoo! Finance site, and therefore the data is available to use for free.

Implementation:

**import yfinance as yf**

**# Create Dataset**

**ticker\_set = 'healthcare' # set group of tickers**

**start\_date = '2000-01-01' # set start date**

**end\_date = '2020-01-01' # set end date**

**symbol = tickers[ticker\_set][0]**

**stock = yf.Ticker(symbol)**

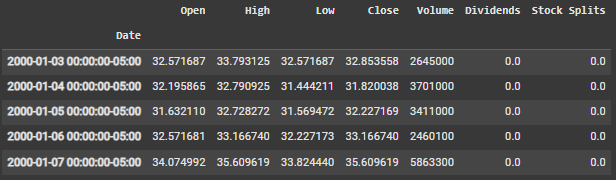
**data = stock.history(start=start\_date, end=end\_date)**

**description = 'Stock ' + symbol + ' From ' + start\_date + ' to ' + end\_date**

**print(description)**

The dataset contains a stock’s open, close, high, low, volume, adjusted close, dividends, and stock splits.

Example Dataset:



These values are the daily prices of the stocks, and do not include the intraday prices. We chose to focus on daily prices since they are typically more stable than intraday prices and have less noise. There are about 250 entries per year, since the stock market is not open all 7 days a week. We decided to use daily stock data, as intraday stock data would be too volatile and noisy as well as too short term to be useful.

### 3.1.2 Stock and Time Frame Selection

We chose stocks from a variety of industries such as technology, healthcare, construction and food. We picked these industries because we thought that they were more stable industries, with technology being a bit more volatile to introduce variability. We also decided to focus on the time period between 2000 to before 2020 to avoid the effects of the covid pandemic on the stock market.

Implementation:

**# Define Tickers of Interest**

**tickers = {'technology': ['NVDA', 'GOOG', 'AAPL'],**

**'healthcare': ['LLY', 'ABBV', 'MRK'],**

**'construction': ['TT', 'CRH', 'URI'],**

**'food': ['ADM', 'KHC', 'GIS']}**

**start\_date = '2000-01-01' # set start date**

**end\_date = '2020-01-01' # set end date**

We chose the following stocks:

Technology:

* NVDA (Nvidia Corp) NASDAQ
* GOOG (Alphabet Inc Class C) NASDAQ
* AAPL (Apple Inc) NASDAQ

Healthcare:

* LLY (Eli Lilly and Co) NYSE
* ABBV (AbbVie Inc) NYSE
* MRK (Merck & Co Inc) NYSE

Construction:

* TT (Trane Technologies PLC) NYSE
* CRH (CRH PLC) NYSE
* URI (United Rentals Inc) NYSE

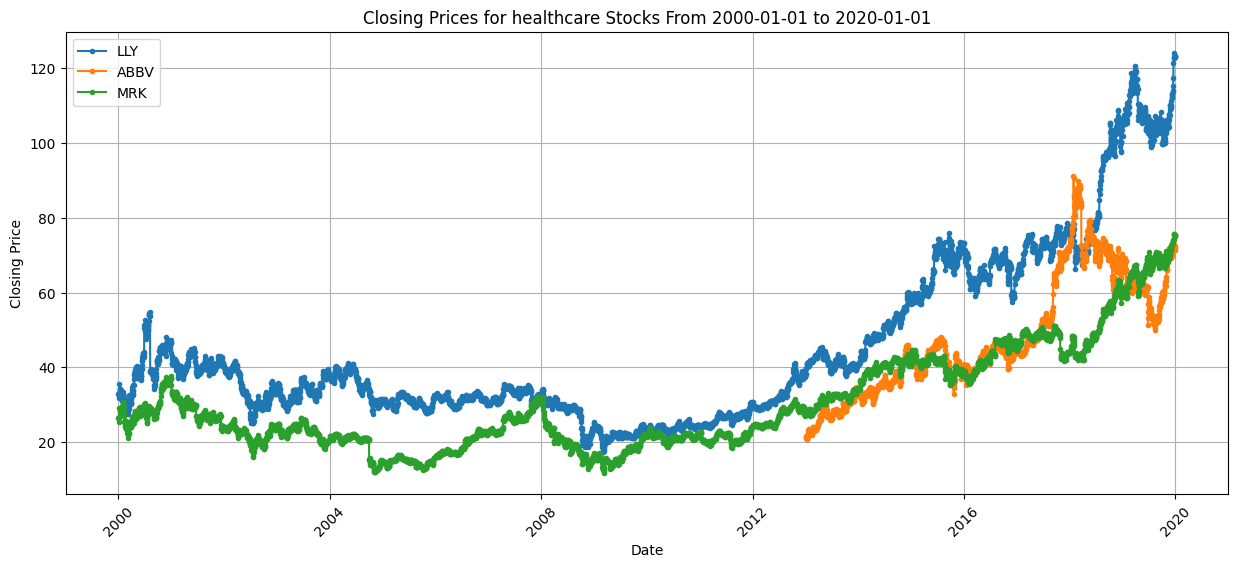
Food:

* ADM (Archer-Daniels-Midland Co) NYSE
* KHC (Kraft Heinz Co) NASDAQ
* GIS (General Mills Inc) NYSE

Note: Some stocks such as ABBV did not exist throughout the entirety of 2000 to 2020. In the case of ABBV, the stock was created starting 2013.

### 3.1.3 Data Visualization and Distribution

The stock’s closing prices can be plotted on a chart with time on the x-axis and prices on the y-axis. Stocks within a particular industry turned out to have prices in relatively similar price ranges. For example, the healthcare stocks are shown here:



Note that the ABBV stock does not contain any data before approximately the year 2013. This is because the ABBV company itself was not founded until April of 2012 and its stock was not created until 2013.

## 3.2 ADDRESSING MISSING VALUES

### 3.2.1 Drop N/A Values

Almost all machine learning models do not work or struggle with missing values, so the simplest solution would be to remove all entries with missing values.

Implementation:

**# Remove missing values, if any**

**X\_stock = data[tickers[ticker\_set][single\_index] + '-Close']**

**X\_stock = list(X\_stock.dropna().to\_numpy())**

However, simply removing values can have unintended consequences with time-series data by creating gaps in the timeline. It could also potentially significantly reduce the amount of data entries in a dataset. Fortunately, the Yahoo! Finance Dataset contains extremely large amounts of historical data that this usually will not be a problem. Therefore, we will use this approach to addressing missing values.

### 3.2.2 Filling in N/A Values with Statistical Data

Typically, another common strategy to address missing values is to replace the missing values with the mean, median, mode, or some other common statistical measurement. However, we felt that this was inappropriate for our dataset since the only missing values were those before a certain time period before the existence of a certain stock (See ***Section 3.1.3*** for an example where the ABBV stock has missing values). Since the missing values were aggregated together rather than sporadically mixed in with the data, it was more in our favor to simply drop the missing values as described above in ***Section 2.1.0***.

## 3.3 UNSKEWING DATA

### 3.3.1 Standard Scaler

Since the stocks have several different ranges of data, we used the Standard Scaler from Sci-kit Learn to unskew our data. Specifically, it shifts the mean and scales the variance to standard values.

Implementation:

**# standardize features of our dataset**

**scaler = StandardScaler()**

**X\_train = scaler.fit\_transform(X\_train)**

**X\_test = scaler.transform(X\_test)**

### 3.3.2 Min-Max Scaler

Another approach to standardizing our data is to use the Min-Max Scaler from Sci-kit Learn. This rescales all values linearly so that the minimum value is 0 and the maximum value is 1.

Implementation:

**# Normalize data**

**scaler = MinMaxScaler(feature\_range=(0, 1))**

**data['Close'] = scaler.fit\_transform(data[['Close']])**

## 3.4 FEATURE ENGINEERING

For feature engineering, we calculated moving averages including the 7 day moving average and 21 day moving averages. We also created features with previous day prices, ranging from yesterday’s price to prices from 5 days ago.

### 3.4.1 Moving Averages

A popular indicator in stock trading is the moving average, which takes the past number of days and averages them. These averages can be used to signal movement of the stock in one direction or the other and can show signs of a trend up or down. Due to this, we will see if using these moving averages can aid in our predictions. In the stock trading world, these are known as simple moving averages (SMA).

Implementation:

**# using 7-day and 21-day Moving Averages**

**data['7\_MA'] = data['Close'].rolling(window=7).mean()**

**data['21\_MA'] = data['Close'].rolling(window=21).mean()**

Here we only chose the 7-day and 21-day moving averages, one for shorter term patterns and another for more long term patterns. However, other different moving averages could be explored.

### 3.4.2 Lagging Prices

Another feature that could be useful is the previous day’s prices. Taking into account the previous day’s prices, we can see if the price moved up or down over the short term and help predict the next day’s price.

Implementation:

**# feature engineering: lagging values of closing price**

**data['Lag\_1'] = data['Close'].shift(1)**

**data['Lag\_2'] = data['Close'].shift(2)**

While only the previous 2 days’ prices are shown above in the example, some models incorporated prices from up to 5 days into the past.

Note: Taking the difference between the current day’s price and the previous day’s price will give a discrete derivative of the stock prices, which could also offer more insight.

### 3.4.3 Other Technical Indicators

Other feature engineering includes more advanced technical indicators such as exponential moving averages (EMA) and the relative strength index (RSI) could be calculated to aid in price predictions as these indicators are used by professional stock traders to evaluate a stock’s performance.

## 3.5 TRAIN TEST SPLIT

### 3.5.1 Test Set Size and Split Methodology

When splitting our data into a training and testing set, we chose a 80% - 20% split between train and test respectively. This is a common ratio used in a wide variety of machine learning applications. Importantly, we did not shuffle our data since our data is time-dependent and it is essential that our data is kept in chronological order. For some quick testing purposes on smaller subsets of the dataset, the last 100 entries of stock closing prices were reserved for predictions.

Implementation:

**# shuffle=False because we want to keep time series data in order**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, shuffle=False)**

# 

# 4. Pipeline

## 4.1. FEATURE SELECTION

Due to the low number of features, including the original ones from the dataset and feature engineering, our feature selection was limited. At a basic level, our feature selection included just looking at the closing price. However, we also did additional feature selection considering the stock’s volume, previous day (lagging) prices, and moving averages.

### 4.1.1 Closing Prices

For some of our models, we simply trained on the stocks’ closing prices, since those were the values that aimed to predict in future time periods. Closing prices give a good representation of a stock’s performance over time since it is the final trading price for a certain day. Additionally, closing price is a standard measurement/benchmark for stock traders and investors to evaluate a stock.

### 4.1.2 Correlation Heat Map

To help with feature selection, a heat map showing the pairwise correlations from a calculated correlation matrix can be displayed.

Implementation:

**import seaborn as sns**

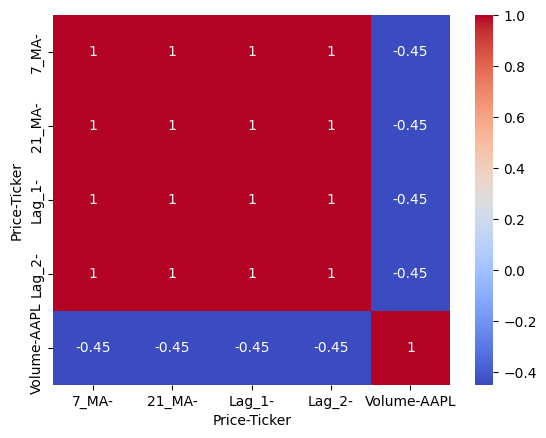
**import matplotlib.pyplot as plt**

**X = data[['7\_MA', '21\_MA', 'Lag\_1', 'Lag\_2', 'Volume']] # features**

**correlation\_matrix = X.corr()**

**sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm')**

**plt.show()**



Positive values (shown in red) depict direct correlations while negative values (shown in blue) depict inverse correlations. The further away from 0, the stronger the correlation. We want to pick features that are not linearly dependent, or in other words, features that don’t have too much correlation. Features that have high correlation give similar information compared to features that aren’t too correlated. By choosing the features with low correlation, we can make sure that the features that we choose give new information compared to each other.

**# features**

**importances = rf\_model.feature\_importances\_**

**feature\_importance = pd.DataFrame({'Feature': X.columns, 'Importance': importances})**

**feature\_importance = feature\_importance.sort\_values(by='Importance', ascending=False)**

**print(feature\_importance)**

## 4.2 LOSS METRICS

### 

When Training our models, the types of loss functions that we use can have great impacts on the outcome of the models. The two different loss metrics that we explored were mean absolute error and mean squared error. Both measure the distance from the target, but differ in the administration of penalties.

**# model evalulation here**

**y\_pred = mlp\_model.predict(X\_test\_scaled).flatten()**

**# check errors again**

**mse = mean\_squared\_error(y\_test, y\_pred)**

**mae = mean\_absolute\_error(y\_test, y\_pred)**

**print(f"Mean Squared Error (MSE): {mse}")**

**print(f"Mean Absolute Error (MAE): {mae}")**

### 4.2.1 Mean Absolute Error (MAE Loss)

Mean absolute error is a direct measurement of the difference between the predicted and actual prices. It tells us exactly how far off we were from the target, scoring each value linearly. It is a fair measurement, unlike MSE loss, where outliers have more impact. We chose this loss metric due to its intuitive interpretability in regards to having the same units to measure differences in stock prices and its more equal averaging.

### 4.2.2 Mean Squared Error (MSE Loss)

Mean squared error is another commonly used loss metric similar to MAE, but places more emphasis on outliers. We also chose this metric as it gives a better sense of variance and has a higher penalty for outliers and worse predictions. Since our dataset from Yahoo! Finance is pretty reliable and higher quality without outliers, MSE loss is suitable for detecting outliers in our predictions and helping our models avoid making outlying predictions. However, due to the squaring aspect of this metric, the exact units of MSE become unclear and harder to interpret.

### 4.2.3 Root Mean Squared Error (RMSE Loss)

Root mean squared error is a compromise between MAE and MSE loss, as it takes the square root of the error. It has units that match the original, similar to MAE, which make this error metric easier to interpret while also placing heavier penalties to outlying values and capturing the variance, similar to MSE. While we ended up sticking to MAE and MSE, RMSE might be something that we explore in the future.

# 

# 5. Model Specifics

## 5.0 SUMMARY

### 5.0.1 Model Training

We decided to try a variety of different models to see which one would perform best in predicting future closing prices. We used various models from pytorch, tensorflow, and sci-kit learn.

Some of the models that we chose are:

* RNN
* LSTM
* Random forest
* XGB Regressor
* MLP
* Transformer

Before training these models, we gathered and preprocessed our data (See ***Section 3***). In training these models, we used feature selection (See ***Section 4.1***) and explored different loss functions including MSE and MAE (See ***Section 4.2***). trained and evaluated these models using the MSE and MAE loss functions. To fine tune the models, hyperparameter tuning was performed.

### 5.0.2 Model Predictions

We used our models to predict the future closing price of the stock that the model was trained on, but also other stocks within the same industry and stocks outside of the same industry. We then plotted our results to see how well the models performed.

It turns out that — was the best model that gave us an error of — when predicting —. See ***Section 6*** for more details on our findings.

## 5.1 BASE MODEL EXPLORATION

### 

### 5.1.1 Linear Regression

The statistical model, linear regression, is an effective model to map the relationship between two variables. There is an independent variable, usually x, and a dependent variable, y. The dependent variable is the variable we are trying to predict. Linear regression assumes a linear relationship between the variables y = β0​ + β1​x1 ​+ β2​x2​ + ⋯ + βn​xn ​+ ϵ where y is the variable we are trying to predict, x are the features or predictors (independent variables), β are the coefficients which influence each feature when predicting y. Lastly, ϵ term is the bias or error term in which the model is unable to predict. Linear regression is often used for its simplicity because the model predictions are easier to interpret and understand within the context of the dataset. For the purposes of stock price predictions, linear regression can often show a trend of the stock data, either upwards or downwards, indicating the growth or loss of stock value. In the event where the closing stock price is influenced by predictors such as trading volume, moving averages, or opening prices, linear regression is able to capture this behavior and identify the trends between these factors.

In the following code segment below, a simple linear regression model is initialized and fit to the Apple stock closing price training data. Once trained, we use the model to predict on a test set, a portion of data that the model has never seen before, and observe its performance.

**model = LinearRegression()**

**model.fit(X\_train, y\_train)**

**# Make predictions**

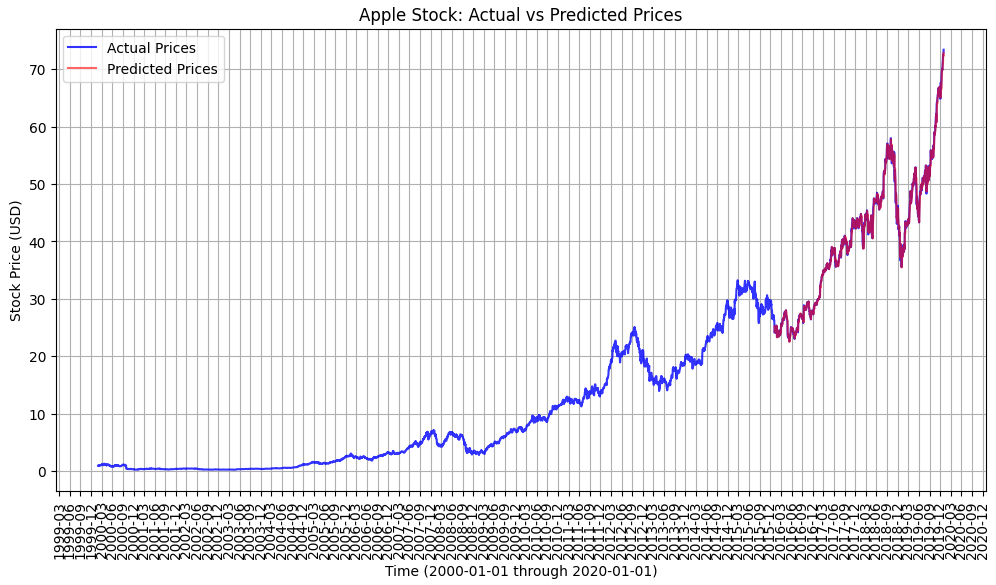
**y\_pred = model.predict(X\_test)**

Mean Squared Error: 0.3855769089002136

Mean Absolute Error: 0.42952491585062325

As seen from the linear regression model error, we observe both the mean squared error and mean absolute error. Mean squared error is analyzed to observe outliers in the data as they will get a heavier penalty from the error metric. If our model is unable to predict values closer to ground truth, then the resulting MSE will be larger, letting us know the model is not performing as well. We observe mean absolute error because it is easier to analyze and interpret how our model is performing. On average, we can see how far off our model is able to predict the closing price of a stock compared to the actual closing price.

In the figure below, we plot the actual closing price of Apple stock and the model’s predicted price. Notice that the last 20% of the data is predicted as we have done an 80 to 20 train test split. Visually, it may seem that this model performs quite well at the high level but we observe the MSE and MAE and there is room for improvement. We will test out other models to see if we can achieve better results.



Linear regression may struggle with stock price predictions because stock prices are usually not deterministic by linear factors. A wide range of nonlinear factors such as market sentiment, global events, and current news can sway the stock’s closing price. Within the stock price, its behavior usually does not assume a linear relationship due to its volatile nature. The linear model may exhibit difficulties in dealing with highly correlated features and instances of time dependent situations. The stock may also contain outliers due to price movements and this behaviour may not be captured appropriately by the model.

### 5.1.2 Random Forests

The random forest model approach is an ensembling method which builds multiple decision trees in the training stage and combines the trees’ outputs, usually by averaging in regression or voting in classification, to improve the model performance and this process inherently reduces overfitting. Bootstrap aggregating (bagging) takes place where the multiple smaller models are trained on a random different subset of training data. This will often lead to better accuracy and reduced model variance. In class, we have discussed the benefits of using random forest models because they are robust and reduce the chance of overfitting since they average the predictions of multiple trees in our case of predicting stock closing prices. Random forest can deal with nonlinear relationships in the stock market’s closing prices and is able to handle a higher number of dataset dimensionality compared to the linear regression model. Since we are attempting to predict the stock market’s closing price which usually depends on nonlinear factors, the random forest model approach seems feasible for our purposes.

In the following code segment, we create our RandomForestRegressor with a baseline of 100 estimators and a random state of 42 so our results are reproducible. X\_train is a subset of the Apple stock we train on. We have done an 80 to 20 train test split between 2000-01-01 through 2020-01-01. The first 80 percent of this data is used to train the random forest model. The remaining 20 percent of the data is used to test, this is data that the model has never seen before.

**# train and make predictions using random forest**

**rf\_model = RandomForestRegressor(n\_estimators=100, random\_state=42)**

**rf\_model.fit(X\_train, y\_train)**

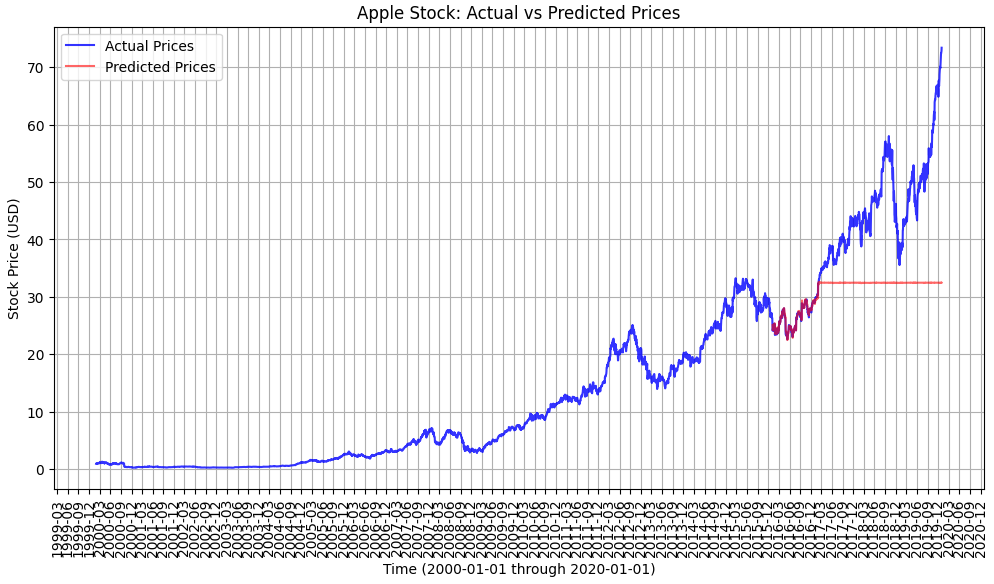
**# make our prediction**

**y\_pred = rf\_model.predict(X\_test)**

Mean Squared Error: 186.5795595537429

Mean Absolute Error: 10.054858566314604

The two error metrics above show the baseline performance of the random forest approach. We notice there is room for improvement but we set this as a baseline of comparison for other models. We are interested in utilizing the time series data with other models further down. In the figure below, we observe the model’s prediction on the last 20 percent of the data. Note that time corresponds to the last 20 percent of the time interval as listed above. We see that the random forest model is able to predict the first few years of the test set and then determines the rest of the test samples fall under a certain category as seen by the flatline. This indicates that the model can be adjusted and possibly have more subtrees or a greater depth to handle the intricacy of the data.



There are drawbacks to using the random forest model as it does not inherently handle time series data which is oftentimes, crucial in successful predictions. More data preprocessing and feature selection may be needed for random forest to train better as it may fit to noise with larger dataset. We utilize random forest as a baseline model to observe its behaviour, especially not handling the times series data.

### 5.1.3 XGBoost Regressor

The XGBoost Regressor model is an extreme gradient boosting approach of gradient boosting usually to solve regression and classification problems. The model iterative constructs trees, usually weaker learners, and then corrects errors from the previous iterations so future iterations are improved, thus improving overall model performance. Due to the structure of this model, it is able to outperform a traditional regression model, like the first baseline model we discussed. The XGBoost Regressor can capture nonlinear patterns within the features. Our dataset had a moderate number of features but this type of model is capable of handling datasets with higher feature dimensions. Since we have discussed how stock closing prices are influenced by many different factors, many being non linear, this model is ideal to analyze these features.

In the following code segment below, we create an XGBRegressor from the xgboost library and set a baseline model to compare against. We use 500 estimators, a learning rate of 0.2, max depth of 8, and a random state of 42 for model reproducibility. We use an 80 to 20 train test split of the Apple stock on its closing price. This means 80 percent of the data is used to train the model and the remaining unseen 20 percent of the data is used to test the model and observe its performance.

**# train model**

**xgb\_model = XGBRegressor(n\_estimators=500, learning\_rate=0.2, max\_depth=8, random\_state=42)**

**xgb\_model.fit(X\_train, y\_train)**

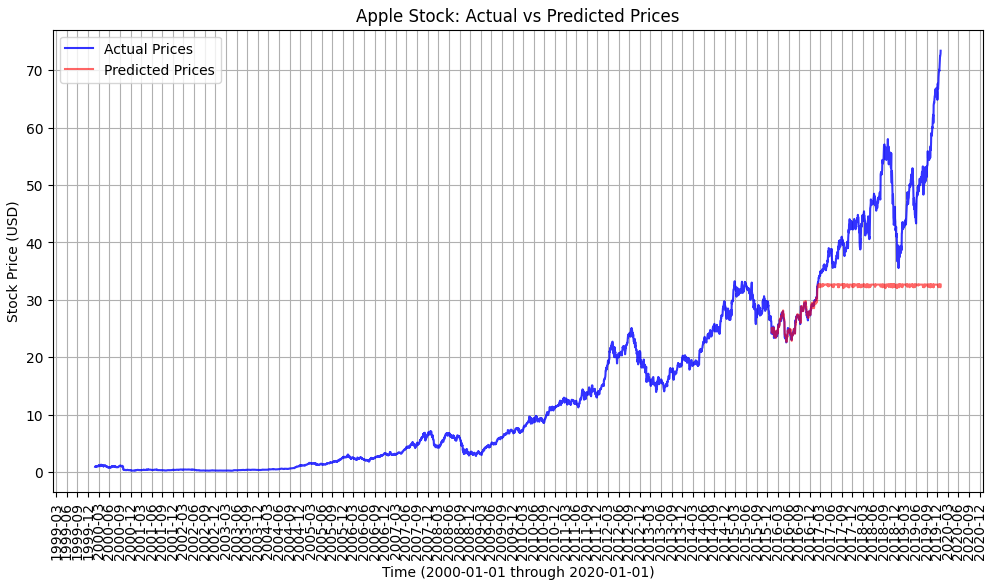
**# have model make predictions**

**y\_pred = xgb\_model.predict(X\_test)**

Mean Squared Error (MSE): 184.37444927664396

Mean Absolute Error (MAE): 9.991655056878315

As a baseline model, we expect the MSE to be larger as the Apple stock can have fluctuations in its closing prices, thus leading to what seemed as outliers for the model. As a baseline, the XGBRegressor seems to have performed similarly to the baseline random forest model above which makes sense because they both utilize decision trees in model predictions, similar internal structure.



In the figure above, once again, we see that the decision tree may be able to predict the first couple of entries of Apple’s test set but then flat line as we may need to finetune this model.

The XGBRegressor model runs into the problem of predicting the time-series data because it is unable to capture its temporal pattern. As a result, model performance may suffer because many factors, including its previous time-series data, usually have a significant effect on successful closing price predictions. We also observe that with the fluctuating closing price of the Apple stock, the model may think these are outliers and fit to random noise and result in poor generalization to new data points.

## 5.2 NEURAL NETWORKS

### 5.2.1 Simple RNN

A Recurrent Neural Network (RNN) is a specific type of deep neural network designed to work with sequential data and make sequential predictions. Since stock data is time-dependent and in chronological order, it would make sense to use an RNN to predict the future prices of a stock. In this project, a pytorch RNN was defined with a hidden layer with size 256 and a fully connected linear layer.

**class SimpleRNN(nn.Module):**

**def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):**

**super(SimpleRNN, self).\_\_init\_\_()**

**self.hidden\_size = hidden\_size**

**self.rnn = nn.RNN(input\_size, hidden\_size, batch\_first=True)**

**self.fc = nn.Linear(hidden\_size, output\_size)**

**def forward(self, x):**

**h0 = torch.zeros(1, x.size(0), self.hidden\_size).to(x.device)**

**out, \_ = self.rnn(x, h0)**

**out = self.fc(out)**

**return out**

The input size was simply 1 as each closing price was a single value. The output size was 1 to predict the next day’s price. The model was trained only on closing price data over 5000 epochs which eventually resulted in a training loss of 0.81. The model was optimized based on the MSE loss metric with the ADAM optimizer and a learning rate of 0.001. The high number of epochs was chosen over increasing the learning rate since the loss metric continued to decrease with more training, and increasing the learning rate resulted in spikes in the training loss. In addition, in order to predict future prices, the features and target vector were offset by 1 day. When predicting other healthcare stocks, there was an MSE of 4.8. While not the best, this was a good starting point for improvement.

### 5.2.2 LSTM 1

The LSTM is a special type of RNN that can keep track of long term values, which can reduce the problem of vanishing gradients of traditional RNNs.

**model\_LLY = Sequential([layers.Input((5, 1)),**

**layers.LSTM(64),**

**layers.Dense(32, activation='relu'),**

**layers.Dense(32, activation='relu'),**

**layers.Dense(1)])**

A series of simple sequential neural networks (model\_LLY, model\_ABBV, model\_MRK) were designed to predict the closing price of several stocks. The input layer expects a shape of (5,1) representing the 5 previous days’ closing prices. It begins with a simple LSTM layer containing 64 units and was chosen to capture time dependencies and patterns in the stock data. Following this layer, two Dense layers with 32 units and ReLU activations functions were introduced to allow the model to learn complex relationships. The final layer outputs a scalar value which will serve as the prediction of the closing price. As for some of our hyperparameters, we choose an MSE loss function and the ADAM optimizer with a learning rate of 0.001 making the model efficient for adaptive learning capabilities such as stock trends. The model was trained over 100 epochs. Across the 3 healthcare stocks, we observed the best-performing model (model\_MRK) had an MSE of 2.27. This model served as a basic entry point for evaluating the prediction performance of LSTMs with zero feature selection or hyperparameter tuning. We will consider this as a basis for future model architectures relying on LSTMs.

### 5.2.3 LSTM 2

The long short-term memory (LSTM) model is a type of recurrent neural network. This type of model can capture sequential data, especially time-series data. In traditional recurrent neural networks, they rely on backpropagation through time, computing the gradients of the loss function. However, this process can lead to the vanishing gradient problem or exploding gradients. As the RNN backpropagates through time, the gradients can become exponentially small and essentially shrink to zero valued. This means long term series data are essentially forgotten. In the exploding gradient scenario, the calculated gradients through time may end up infinitely large. This leads to model instability with infinitely large loss values, thus the model is unable to train and previous history is lost. The LSTM can address these problems by using a memory cell to determine the flow of information so that the two issues from the RNN do not arise.

Benefits of using an LSTM can include its ability to capture and predict time-series data, deal with flexible dimensional datasets, resolve nonlinear relationships in the predictors, and the ability to train without feature engineering. In the following code segment below, we have already done the data preprocessing and splitting beforehand (see above). We create a Sequential model from the tensorflow set of keras models. First, we have an LSTM with 50 neurons to capture the differences in the sequence data. Additional layers are added as an activation layer to add nonlinearity to the model and allow it to capture complex patterns. The final layer is a single output so that we can calculate the regression of the stock closing price. The Adam optimizer is good for gradient-based optimization, hence it is used in our model. MSE is the models loss metric when training.

**# building the LSTM Model**

**model = Sequential()**

**model.add(LSTM(50, return\_sequences=True, input\_shape=(X\_train.shape[1], 1)))**

**model.add(LSTM(50, return\_sequences=False))**

**model.add(Dense(25, activation='relu'))**

**model.add(Dense(1)) # single output for regression here**

**model.compile(optimizer='adam', loss='mean\_squared\_error')**

**# train**

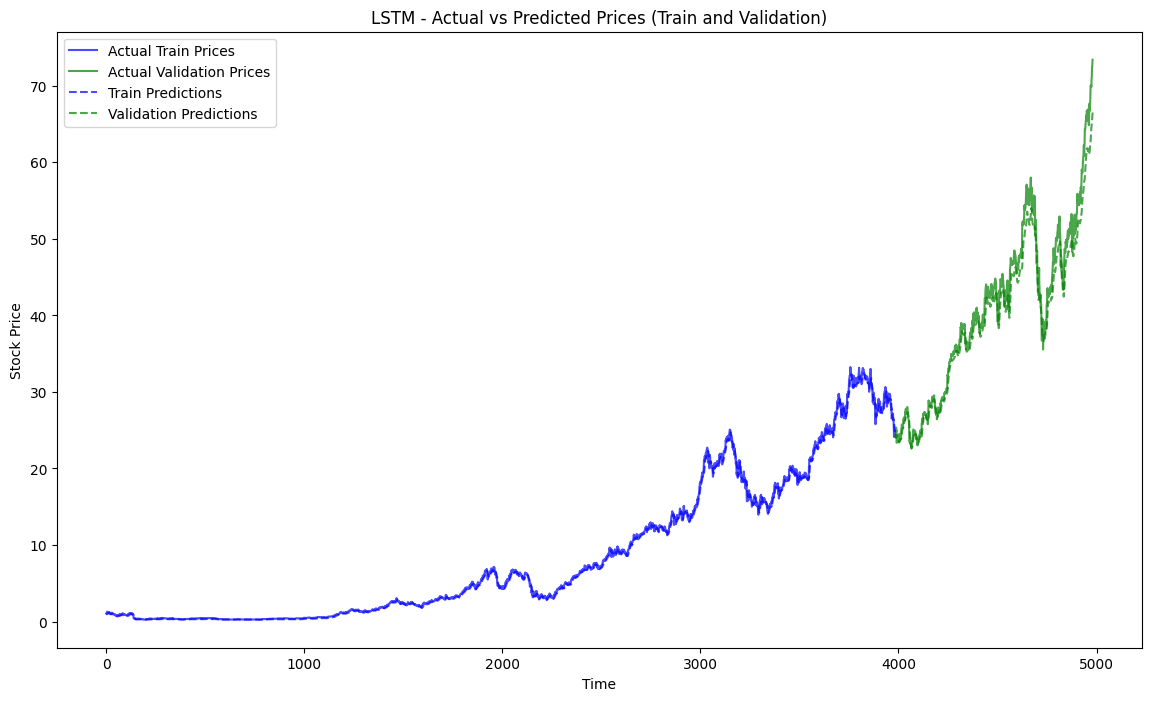
**model.fit(X\_train, y\_train, batch\_size=32, epochs=20)**

# make predictions

y\_pred = model.predict(X\_test)

Train MSE: 0.14920573391301184

Validation MSE: 4.623605735044073



In the figure above, we are able to visualize the LSTM model’s performance on the test set. Though it appears the model seems to follow the behaviour of the ground truth, there is room for improvement as seen in the validation mean squared error. This indicates there is room for hyperparameter tuning and adjustments. The LSTM model usually requires a larger set of training data, increased chances of overfitting if model is too complex, sensitivity to hyperparameters, and difficulty in including factors (current events, news, trends, etc.) as it focuses on previous existing data.

### 5.2.4 LSTM 3

The third iteration of the LSTM model introduced significant enhancements to address overfitting and improve the generalization of predictions. The architecture begins with an LSTM layer containing 50 units, configured with return\_sequences=True, which allows the network to propagate temporal features through subsequent layers. To counter overfitting, a dropout layer with a rate of 0.2 was added immediately after the first LSTM layer. This layer randomly deactivates 20% of the neurons during training, effectively regularizing the model. A second LSTM layer, also with 50 units, further processes the temporal data, followed by an additional dropout layer with the same rate. The network concludes with two Dense layers: the first with 25 units to introduce additional complexity for learning nonlinear relationships, and the final output layer with a single neuron to predict the stock closing price.

**# Build LSTM Model**

**model = Sequential([**

**LSTM(50, return\_sequences=True, input\_shape=(X\_train.shape[1], 1)),**

**Dropout(0.2),**

**LSTM(50, return\_sequences=False),**

**Dropout(0.2),**

**Dense(25),**

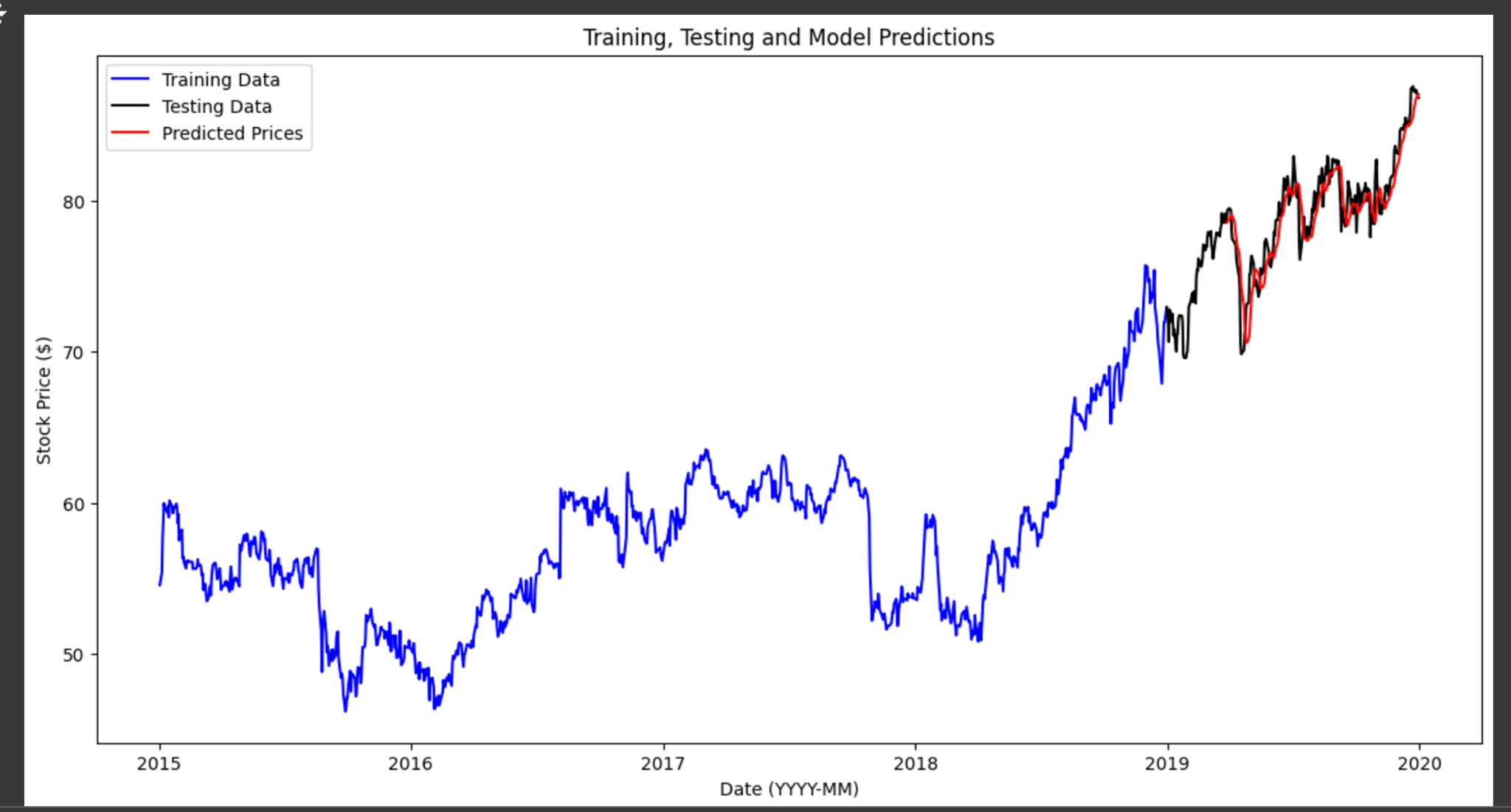
**Dense(1) # Output layer for predicted price**

**])**

**# Compile the model**

**model.compile(optimizer='adam', loss='mean\_squared\_error**

For optimization, the Adam optimizer was chosen due to its adaptive learning capabilities, and mean squared error (MSE) was selected as the loss function to evaluate model performance. Training was conducted using a batch size of 32, and the EarlyStopping and ModelCheckpoint callbacks were employed to streamline the training process. EarlyStopping halted training if the validation loss did not improve for 5 consecutive epochs, while ModelCheckpoint ensured the model with the lowest validation loss was saved for evaluation.



The results demonstrated significant improvements over previous iterations. The training loss steadily decreased, stabilizing at approximately 0.0032, while the validation loss converged to 0.0008, as shown in the loss curve visualization. These results suggest that the model effectively avoided overfitting while learning meaningful patterns in the data. The test set predictions aligned closely with the actual stock prices, as depicted in the prediction visualization, capturing the upward and downward trends in the stock data with reasonable accuracy. However, small deviations between the predicted and actual prices highlight potential areas for further optimization, such as hyperparameter tuning or incorporating additional features.

The model achieved a test MSE of 1.047, which marked a notable improvement compared to LSTM 2. This enhancement can be attributed to the inclusion of dropout layers, which effectively regularized the network, and the use of callbacks to prevent unnecessary overfitting during training. However, challenges remain in fully capturing the complexities of stock price movements. LSTMs are inherently limited by their reliance on historical data, and factors such as current events, market news, and investor sentiment are not explicitly represented in this approach. Additionally, the model remains sensitive to hyperparameter selection and requires careful tuning to achieve optimal performance.



Despite these limitations, this iteration of the LSTM model provides a strong foundation for future developments. By incorporating techniques such as feature engineering or attention mechanisms, the model could potentially improve its ability to predict stock prices more accurately. The results obtained here underline the potential of LSTMs in time-series forecasting and highlight the importance of balancing model complexity with generalization.

### 5.2.5 Multilayer Perceptron (MLP)

The multilayer layer perceptron can be used for regression and classification problems. This type of neural network has interconnected layers to process inputs and make model predictions. The input features are mapped to an input neuron in the first layer. The hidden layers have a weighted sum of the inputs and apply a bias along with additional non linear activation functions to allow the model to predict complex patterns in the data. Since we are predicting the closing price of a stock, we will have the final output layer be a single neuron with model prediction.

In the segment of code below, we use a Sequential model to create our layers. Notice that the final output layer is 1 neuron which is our model making a prediction the stock’s closing price. We perform hyperparameter training on this MLP to determine which values give the best results.

**# create the MLP model**

**mlp\_model = Sequential([**

**Dense(128, activation='relu', input\_dim=X\_train\_scaled.shape[1]), # input**

**Dense(64, activation='relu'),**

**Dense(32, activation='relu'),**

**Dense(1) # output**

**])**

**# make model**

**mlp\_model.compile(optimizer=Adam(learning\_rate=0.0001), loss='mse', metrics=['mae'])**

**# traiing our model**

**history = mlp\_model.fit(**

**X\_train\_scaled, y\_train,**

**validation\_data=(X\_test\_scaled, y\_test),**

**epochs=100,**

**batch\_size=32,**

**verbose=1**

**)**

**# evaluate the model and see its performance**

**y\_pred = mlp\_model.predict(X\_test\_scaled).flatten()**

**Table 5.2.5-A:** MLP (Using all features + engineered ones too)

|  |  |  |  |
| --- | --- | --- | --- |
| # of epochs | Learning Rate | MSE | MAE |
| 50 | 0.001 | 0.8838916657260595 | 0.7527006896637967 |
| 100 | 0.001 | 1.0876943397779977 | 0.8030306225640704 |
| 100 | 0.0001 | 0.6852525015661706 | 0.5920064190211348 |

**Table 5.2.5-B:** MLP (With feature selection: only using 'Lag\_1', '7\_MA')

|  |  |  |  |
| --- | --- | --- | --- |
| # of epochs | Learning Rate | MSE | MAE |
| 100 | 0.0001 | 2.574264275539379 | 1.179527199042521 |
| \*100 | \*0.001 | \*0.4465073164240513 | \*0.46208631528814437 |
| 100 | 0.01 | 2.187106382032504 | 1.30975598138446 |

*\*Best Result with Lowest Error*

Test MSE: 0.4465073164240513

Test MAE: 0.46208631528814437

The tables above demonstrate a few stages of hyperparameter tuning with most features compared to fewer selected features. We also completed a random grid search to find the best parameters and models as shown below.

Best hyperparameters from RandomizedSearch: {'batch\_size': 89, 'epochs': 50, 'model\_\_layer\_size': (64, 32, 16), 'model\_\_learning\_rate': 0.030352075650423926, 'optimizer': 'sgd'}

Best best\_model\_random : KerasRegressor(

model=<function create\_mlp\_model at 0x7ad41fc8cd30>

build\_fn=None

warm\_start=False

random\_state=None

optimizer=sgd

loss=None

metrics=None

batch\_size=89

validation\_batch\_size=None

verbose=0

callbacks=None

validation\_split=0.0

shuffle=True

run\_eagerly=False

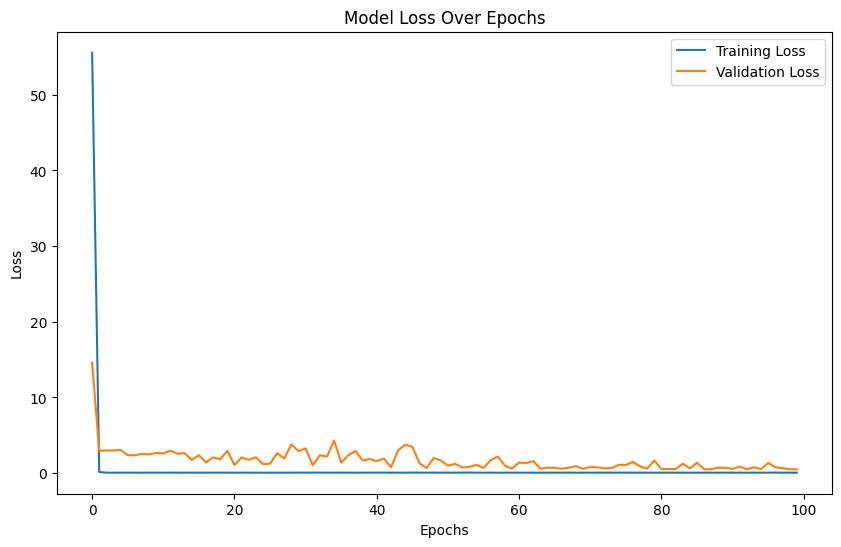
epochs=50

model\_\_layer\_size=(64, 32, 16)

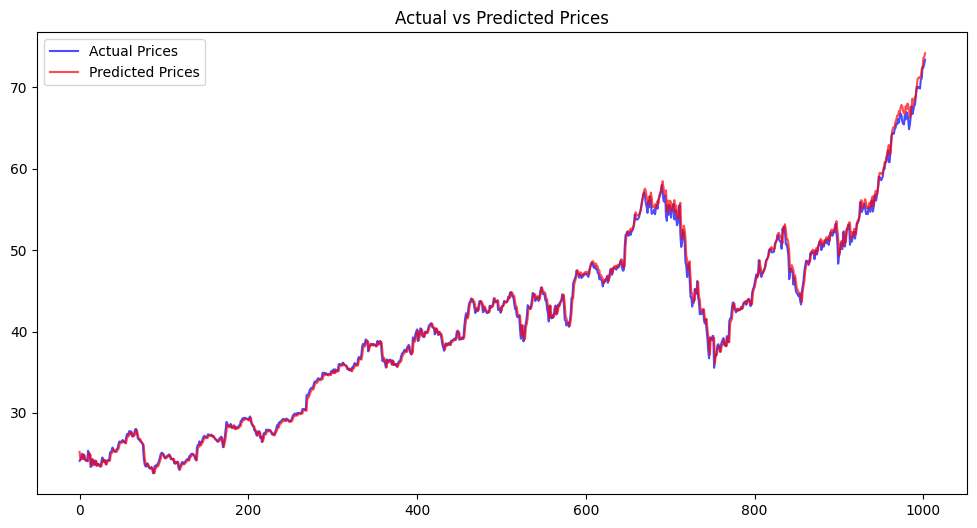
model\_\_learning\_rate=0.030352075650423926

)

As the model trains, we can observe its loss over the epochs. This gives us a sense of how the model is performing and prompts us if we should adjust hyperparameters for future improvements.



In the graph below, we observe the actual price of Apple stock closing price against the model’s prediction on the last 20 percent of data within the data set time frame we listed above. So far, it seems to be producing relatively close patterns to the ground truth data. We can attempt to improve the model by including additional layers, more hyperparameter tuning, and testing different features to see how the model is affected.



Multilayer perceptron models will need feature augmentation that account for previous history, such as moving averages of the past number of days. Regularization helps the MLP model from learning noise in the data. Other factors such as current events, live news, and market sentiment may not be captured as effectively in this model, leading to missing data to consider in predicting stock closing prices.

### 5.2.6 Transformer

The transformer is a deep learning model architecture that relies on the self-attention mechanism as introduced in the seminal paper “Attention is All You Need.” This model is best at tasks that involve sequential or structured data like Natural Language Processing, Image Processing, and time series data like the closing price of a stock. One of the transformer’s main strengths is that it processes sequences in parallel using its attention matrix and does not have the problem of a vanishing gradient like RNNs. This means that the transformer is theoretically faster to train on and can be more accurate than even an LSTM RNN. Another advantage of the transformer is that its able to capture long-term dependencies and assign dynamic weighting to inputs. This ability to capture these dependencies will help predict the stock prices better as stock prices are influenced by patterns and trends over time and the dynamic weighting will enable the model to focus on the most relevant parts of the data without much feature selection.

A PyTorch Transformer model with two layers was created with input dimension and output dimension of size 1 as this model was only predicting the closing price of a stock.

**class TransformerModel(nn.Module):**

**def \_\_init\_\_(self, input\_dim, embed\_dim, num\_heads, ff\_dim, num\_layers, output\_dim, dropout=0.1):**

**super(TransformerModel, self).\_\_init\_\_()**

**self.embedding = nn.Linear(input\_dim, embed\_dim)**

**self.positional\_encoding = nn.Parameter(torch.zeros(1, SEQ\_LENGTH, embed\_dim))**

**self.encoder\_layer = nn.TransformerEncoderLayer(**

**d\_model=embed\_dim,**

**nhead=num\_heads,**

**dim\_feedforward=ff\_dim,**

**dropout=dropout**

**)**

**self.transformer = nn.TransformerEncoder(self.encoder\_layer, num\_layers=num\_layers)**

**self.fc\_out = nn.Linear(embed\_dim, output\_dim)**

**def forward(self, x):**

**# Add embeddings and positional encoding**

**x = self.embedding(x) + self.positional\_encoding**

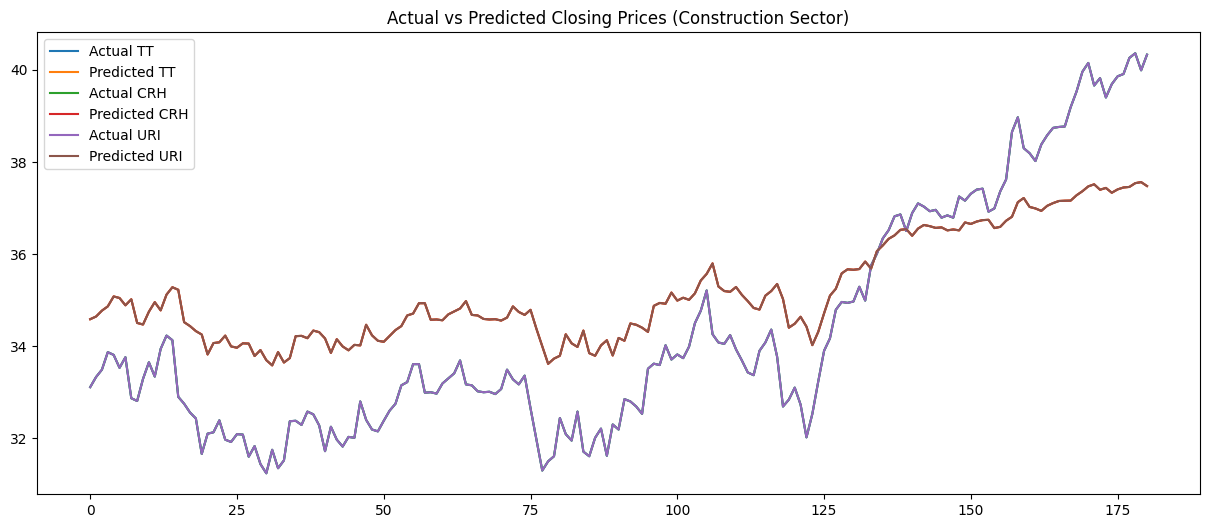
**x = self.transformer(x)**

**# Take the output of the last time step**

**out = self.fc\_out(x[:, -1, :])**

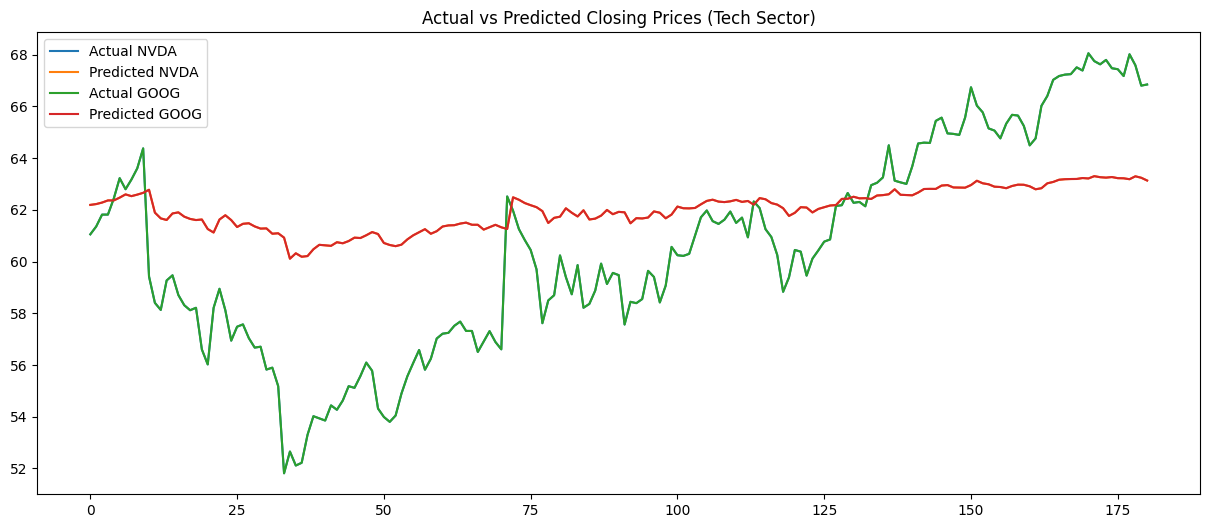
**return out**

The input size was simply 1 as each closing price was a single value. The output size was 1 to predict the next day’s price. The model was trained on 20 epochs as it was quickly observed that the loss was leveling off really quickly, so a smaller epoch count was chosen to speed up iterating on the model. The test loss was 0.008322, which was an okay result. The MSE was a rather high score of 2.6365912 and the test MAE was 1.3823779. The following is a graph that shows the actual vs predicted closing price for the construction sector stocks.



The results show that the overall trend lines and small fluctuations in the day to day stock price were correctly predicted but the magnitude of the stock price was very off.

The model trained on construction was run on a tech sector stock and produced the following graph



As shown in the graph, the model does not fare well when predicting the stock price from a different industry. This could be due to more volatility in the technology sector, which limits the model’s ability to transfer what it learned. The Test Loss was 0.0069509, the Mean Absolute Error (MAE) was 3.066314220428467, and the Mean Squared Error (MSE) was 13.151952743530273, demonstrating a considerable regression in performance.

Overall, this model was a good starting point on using the transformer for predicting stock prices and could be combined with a CNN to preprocess time-series data (e.g., extracting short-term trends) before feeding it to a transformer for modeling long-range dependencies.

# 6. Findings and Predictions

## 6.0 SUMMARY

Our findings show that it is possible to predict the future closing prices of stocks when a model is specifically trained on past historical data for that stock (See ***Section 6.0.2***). However, it is significantly more difficult to predict the closing prices of other stocks that the model was not trained on, even stocks that were in the same industry as the training stock (See ***Section 6.0.3***). When attempting to predict future prices on stocks in a different industry compared to the training stock (See ***Section 6.0.4***), the performance was even worse. This implies that the combination of these models and stock data do not generalize well to other stocks.

### 6.0.1 Best Model and Predictions

The best model at predicting the future price of a stock turns out to be the Linear Regression Model, as it has the lowest MAE score of 0.429 as well as the lowest MSE score of 0.385 out of all the models we used.

Out of the models we tested, the best model a predicting the prices of other stocks within the same industry turns out to the MLP model. However, the error scores are extremely high, suggesting that the best model isn’t adequate enough to predict the prices of other stocks even within the same industry.

For predicting the stocks in other different industries, the LSTM 1 model performed the best with a relatively low MSE score. However, despite being the best, this MSE score is also too high to be useful.

### 6.0.2 Overview of Same Stock Future Price Predictions

Here are the results of models trained on a certain industry of stocks and then predicting the future prices of that same stock. The mean absolute error (MAE) and mean squared error (MSE) are recorded.

**Table 6.0.2-A:** Prediction Error Scores for Various Models Predicting Future Prices of Stocks

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | MAE | MSE | Stock | Industry |
| Linear Regression | 0.429524915850 | 0.385576908900 | AAPL | Technology |
| Random Forest | 10.05485856631 | 186.5795595537 | AAPL | Technology |
| XGBoost | 9.991655056878 | 184.3744492766 | AAPL | Technology |
| Simple RNN | 1.474582443237 | 4.814963037291 | LLY | Healthcare |
| LSTM 1 | - | 2.2735 | MRK | Healthcare |
| LSTM 2 | - | 4.623605735044 | AAPL | Technology |
| LSTM 3 | 1.095504 | 2.28531 | MRK | Healthcare |
| MLP | 0.462086315288 | 0.446507316424 | AAPL | Technology |
| Transformer | 1.3823779 | 2.6365912 | TT | Construction |

From the Table 6.0.2-A shown above, it is clear that for the technology industry, the linear regression model performed the best with MSE and MAE errors well below 1. However, the MLP model is a close 2nd place as it also has error scores just barely above those for linear regression. For the healthcare industry, the best model was LSTM 1. Overall, the model that performed the best out of all models was the linear regression model. The model that performed the worst was the Random Forest model.

### 6.0.3 Overview of Same Industry Predictions

We also explored how well models could predict different stocks from the same industry.

**Table 6.03-A:** Prediction Error Scores for Various Models Predicting Same Industry Stocks

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Model | MAE | MSE | Train Stock | Test stock | Industry |
| MLP | 17.2422698127 | 481.23895901007 | AAPL | NVDA | Technology |
| MLP | 222.939357525 | 71813.730312219 | AAPL | GOOG | Technology |
| LSTM 1 | 45.129402 | 2224.8428 | MRK | ABBV | Healthcare |
| LSTM 1 | 57.978508 | 3543.2827 | LLY | ABBV | Healthcare |
| LSTM 1 | 121.67775 | 15564.102 | ABBV | LLY | Healthcare |
| LSTM 1 | 161.36241 | 26796.455 | MRK | LLY | Healthcare |
| LSTM 1 | 38.03222 | 1457.0818 | ABBV | MRK | Healthcare |
| LSTM 1 | 101.650406 | 10344.734 | LLY | MRK | Healthcare |
| Simple RNN | 11.2549534606 | 180.69957864880 | LLY | MRK | Healthcare |

It seems that the Simple RNN and MLP models perform better than the LSTM model in predicting the prices of other stocks from the same industry that it is trained on. LSTM model has higher error, but it is worth noting that the LSTM model performs better when the prices ranges are similar between the train and test stocks. This would suggest an error in scaling, which could be addressed in future endeavors. Furthermore, another factor is differences in the industry trained since the MLP model was trained on the technology industry while the LSTM and RNN model was trained on the healthcare industry. Regardless, it turns out that even stocks from the same industry are difficult to predict and result in much higher error compared to predicting future prices of the same stock.

### 6.0.4 Overview of Different Industry Predictions

**Table 6.0.4-A:** Prediction Error Scores for Various Models Predicting Different Industry Stocks

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | Train Industry | MAE | MSE | Test Industry |
| LSTM 1 | Healthcare | - | 323.94366 | Construction |
| \*LSTM 1 | Healthcare | \*6.874618 | \*99.03091 | Construction |
| MLP[1] | Technology | 275.9162184117 | 87714.49943855 | Food |
| MLP[2] | Technology | 330.9713993685 | 124301.7595811 | Food |
| Simple RNN | Healthcare | 8.905188446044 | 87.11786111679 | Construction |
| Transformer | Construction | 3.066314220428 | 13.15195274353 | Technology |

*\*The LSTM 1 Model was retrained and obtained a better testing score*

*[1]: MLP trained on AAPL, predicting ADM*

*[2]: MLP trained on AAPL, predicting GIS*

From Table 6.0.4-A shown above, the Transformer showed the best performance when predicting the prices of stocks outside of the industry that the model was trained on. The MLP model fared the worst. While MAE and MSE scores were still higher than those of models predicting future prices of the same stock, these loss values are considerably lower than those when predicting the prices of stocks within the same industry.

## 6.1 INDIVIDUAL RESULTS

### 6.1.1 Simple RNN

Simple RNN (Trained on Healthcare, predicting Same Stock Healthcare):

MAE Loss: 1.4745824432373047

MSE Loss: 4.814963037291309

****

Simple RNN (Trained on Healthcare, predicting Healthcare):

MSE Loss: 180.69957864880126

MAE Loss: 11.25495346069336



Simple RNN (Trained on Healthcare, predicting Construction):

MSE Loss: 87.11786111679277

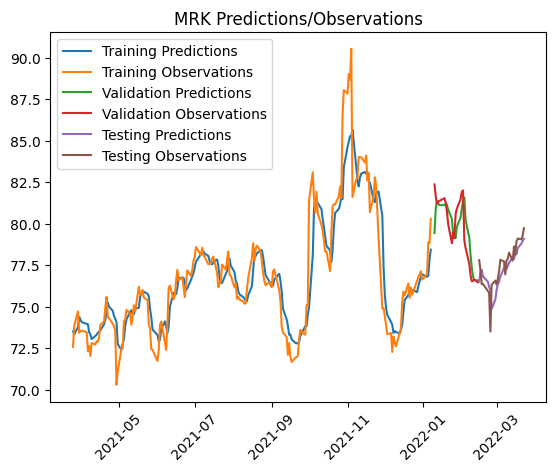
MAE Loss: 8.905188446044923



### 6.1.2 LSTM 1

LSTM 1 (Trained in Healthcare, predicting Healthcare):

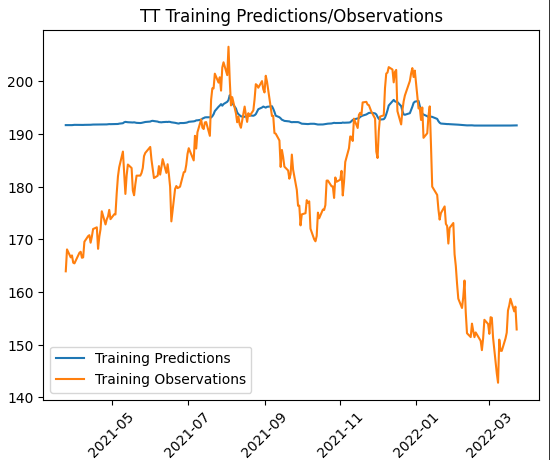
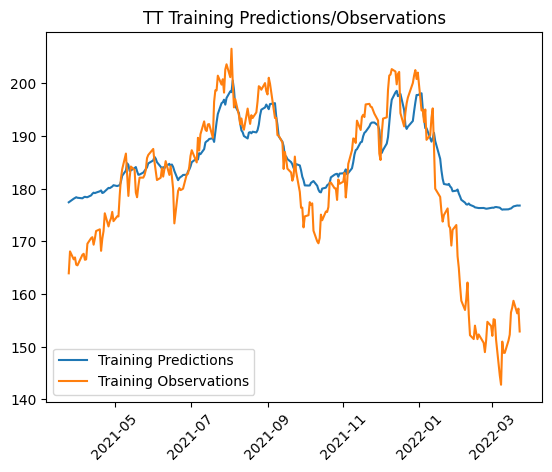
MSE Loss: 2.2735



### 

LSTM 1 (Trained in Healthcare, predicting Construction):

MSE Loss: 323.94366 Retrained MSE Loss: 99.03091



LSTM 1 Findings:The LLY model's results suggest that its relative success in predicting construction stocks, particularly compared to the ABBV and MRK healthcare models, may be due to an alignment between the price range and patterns in LLY's training data and those of the construction stock TT. This alignment likely enabled the model to generalize its learned relationships more effectively, capturing trends or shapes in the new data even when the industry contexts differ. The healthcare models' inability to fully predict construction stock behavior underscores the industry-specific nature of stock price movements, influenced by unique market dynamics, economic factors, and events. However, when used to predict healthcare industry stocks after being trained on such data, the LLY model performed much better, accurately predicting the general structure of stock prices with only minor discrepancies. It struggled to capture large spikes or falls in prices over short periods, potentially due to the large window of previous dates used as input. For less volatile price movements, the model excelled, indicating that stock price predictions are more straightforward when there are no intense fluctuations. This underscores the importance of considering data characteristics, such as price ranges and volatility, when applying machine learning models across different domains and industries.

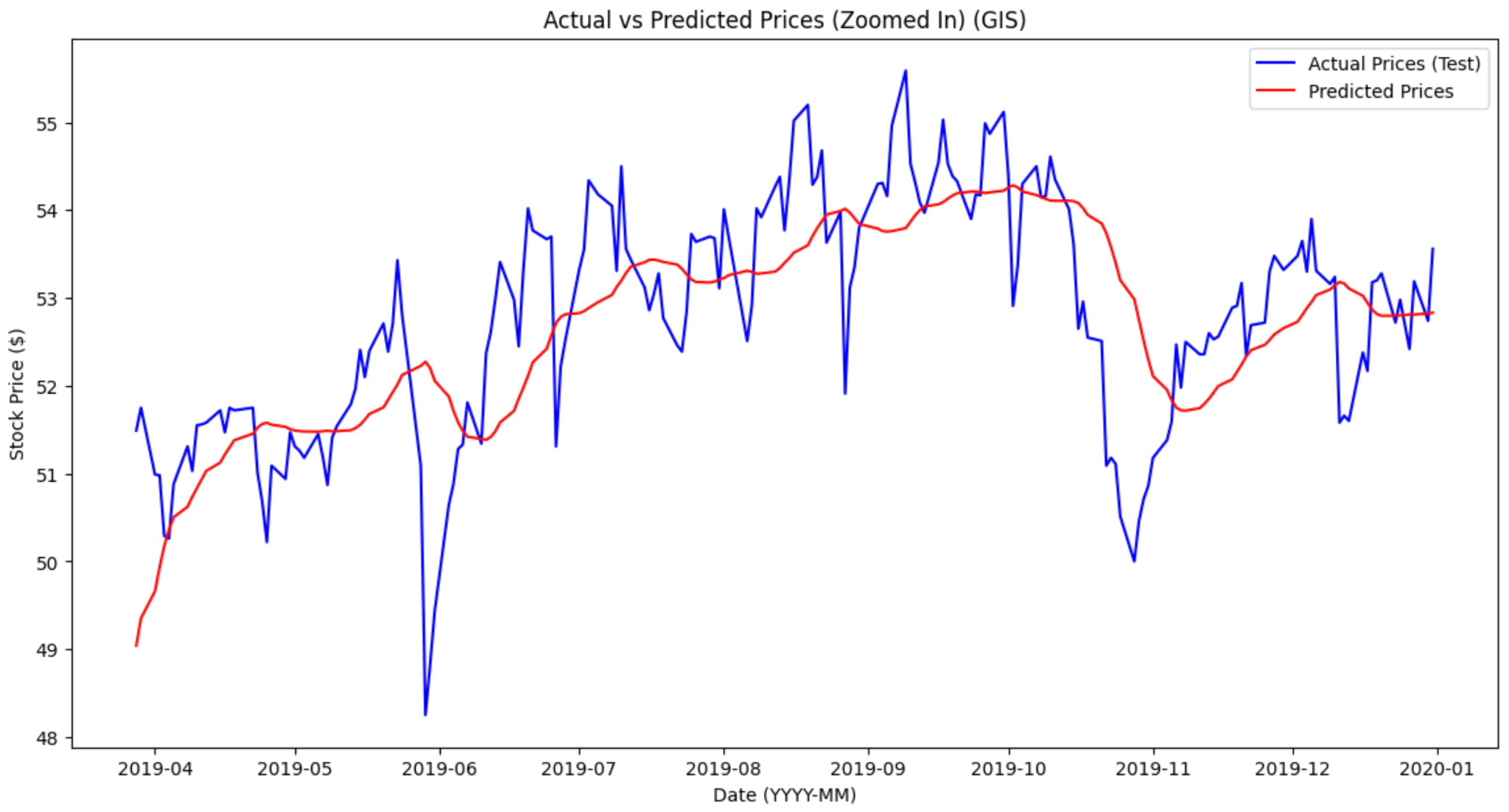
### 6.1.3 LSTM 3

LSTM 3 (Trained in Food, predicting Food):

Training Loss (MSE): 0.0031995901372283697

Testing Loss (MSE): 0.0008101946441456676

Mean Squared Error (Test Set): 1.0465153548809962

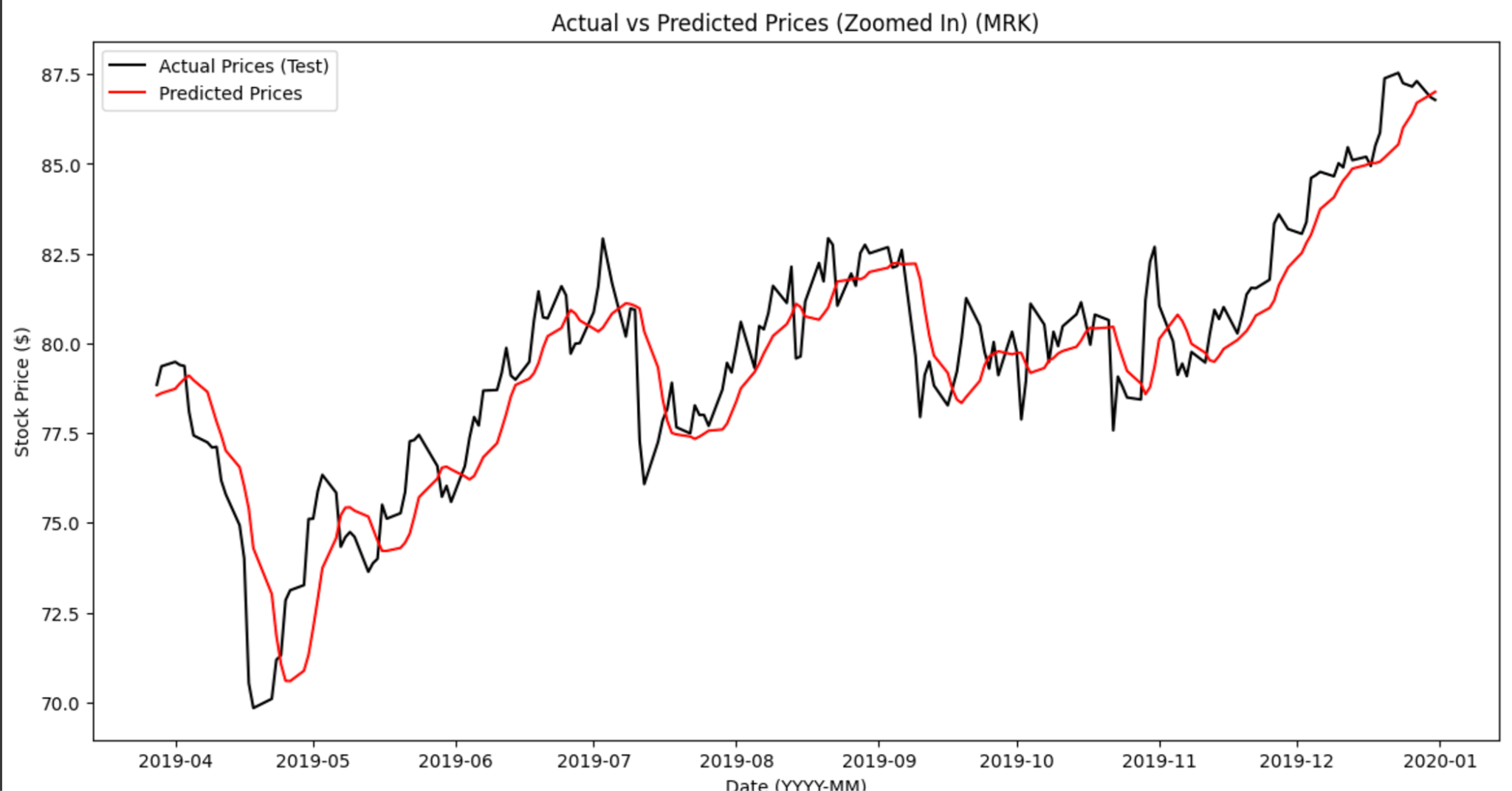


LSTM 3 (Trained in Healthcare, predicting Healthcare):

Training Loss (MSE): 0.001496272161602974

Testing Loss (MSE): 0.0024323591496795416

Mean Squared Error (Test Set): 2.117309065603221



LSTM 3 Insights: The LSTM 3 model highlights the benefits of regularization and optimization techniques in improving prediction accuracy. By incorporating dropout layers with a rate of 0.2 and leveraging EarlyStopping and ModelCheckpoint callbacks, the model effectively reduced overfitting and enhanced generalization. The architecture, consisting of two LSTM layers and dense layers, enabled the capture of complex temporal and nonlinear patterns in stock price data. While test predictions aligned closely with actual stock prices, small deviations indicate room for further optimization. Additionally, the reliance on historical data underscores the model’s limitation in accounting for external market factors like news or investor sentiment. Future efforts could focus on feature engineering, attention mechanisms, or integrating external data sources to refine accuracy. Overall, this iteration underscores the potential of LSTMs in time-series forecasting while emphasizing the need to balance model complexity with generalization.

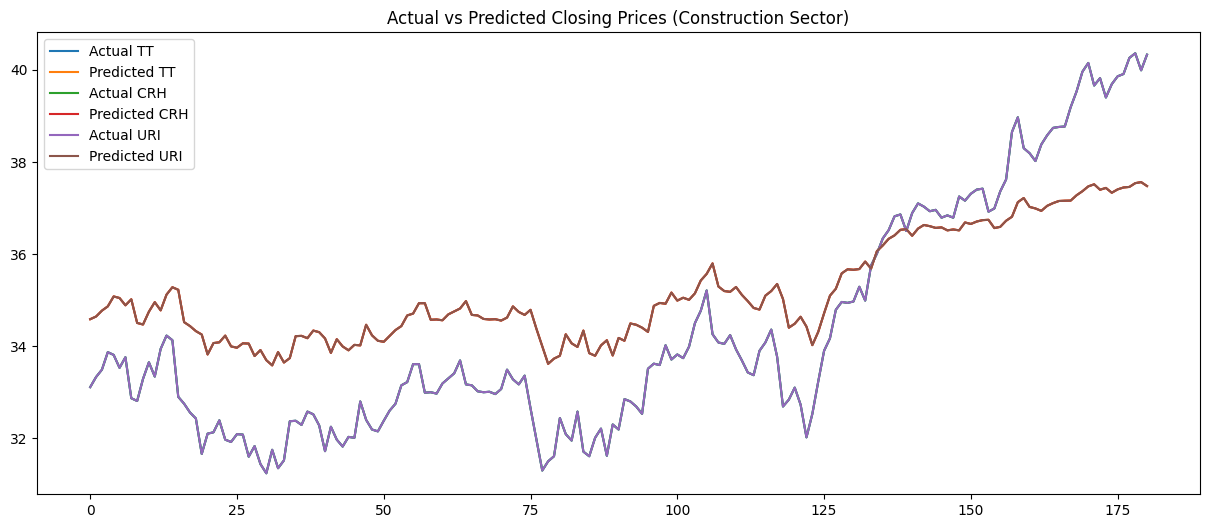
### 6.1.4 Transformer

Transformer (Trained on construction, predicting construction)

Test Loss: 0.008322750218212605

MSE: 2.6365912

MAE: 1.3823779



### 6.1.5 Baseline Model Performance

Linear Regression Model

* Baseline model for reference / testing pool of models
* Trained on Apple stock (Technology Industry) predicting on Apple Stock (Technology Industry)
* Mean Squared Error: 0.3855769089002136
* Mean Absolute Error: 0.42952491585062325

Random Forest Model

* Baseline model for reference / testing pool of models
* Trained on Apple stock (Technology Industry) predicting on Apple Stock (Technology Industry)
* Mean Squared Error: 186.5795595537429
* Mean Absolute Error: 10.054858566314604

XGBoost Regressor Model

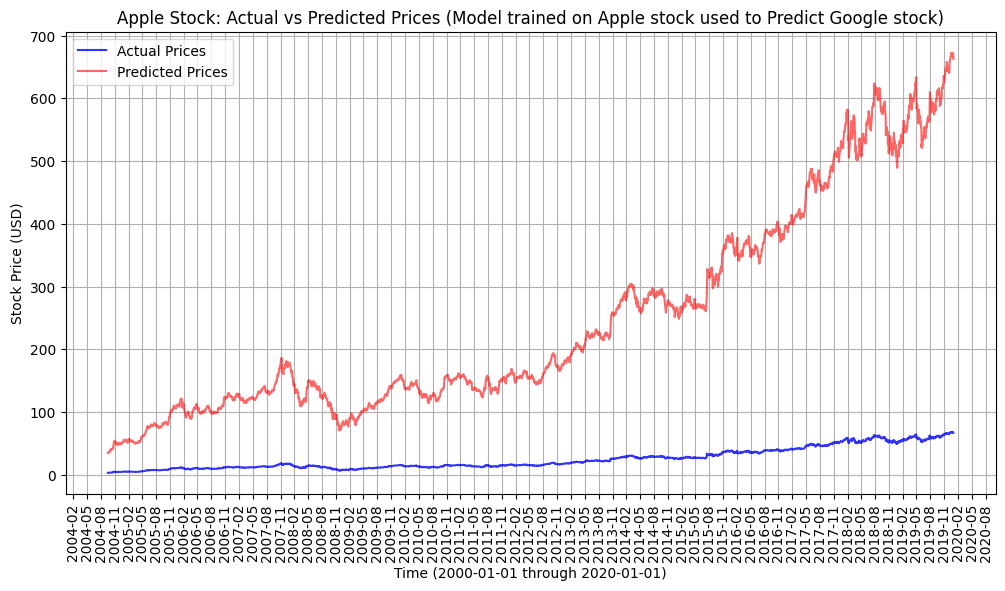
* Baseline model for reference / testing pool of models
* Trained on Apple stock (Technology Industry) predicting on Apple Stock (Technology Industry)
* Mean Squared Error (MSE): 184.37444927664396
* Mean Absolute Error (MAE): 9.991655056878315

### 6.1.6 Long Short-Term Memory Model (LSTM 2)

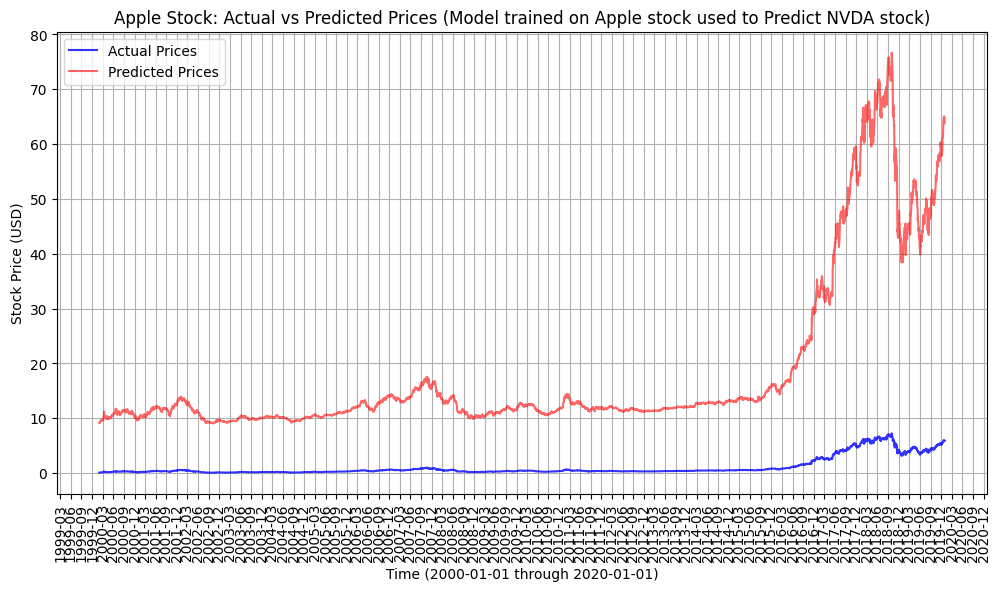
* Baseline model for reference / testing pool of models
* Trained on Apple stock (Technology Industry) predicting on Apple Stock (Technology Industry)
* Train MSE: 0.14920573391301184
* Validation MSE: 4.623605735044073

### 6.1.7 Multilayer Perceptron Model

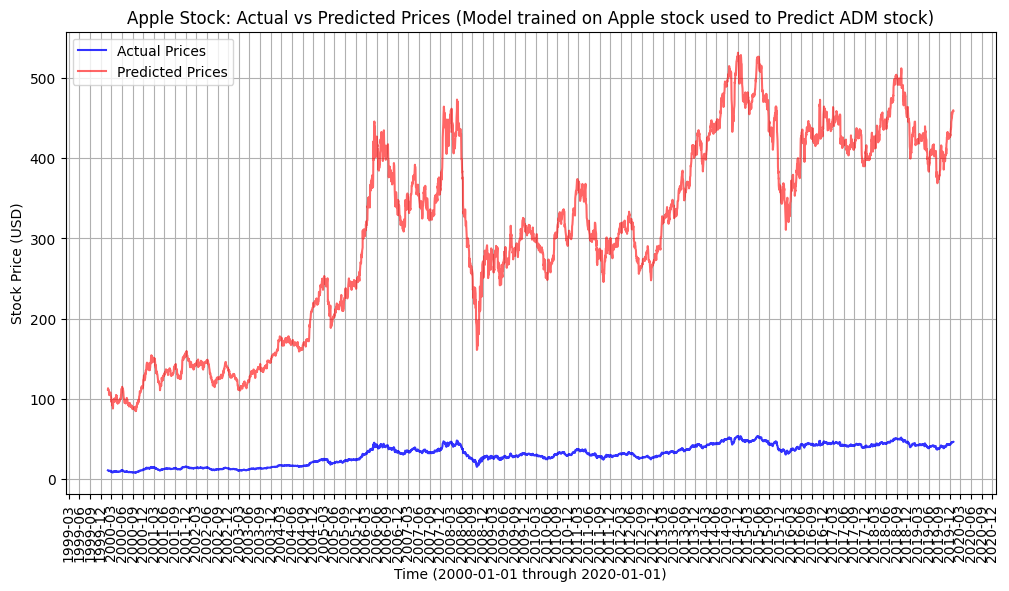
* Selected model to perform model comparison on different stocks
* Trained on Apple Stock (Technology Industry)
* Test MSE: 0.4465073164240513
* Test MAE: 0.46208631528814437
* Trained model (Technology Industry) predicting on Google Stock (Technology Industry)
  + Mean Squared Error (MSE): 71813.73031221946
  + Mean Absolute Error (MAE): 222.9393575257777



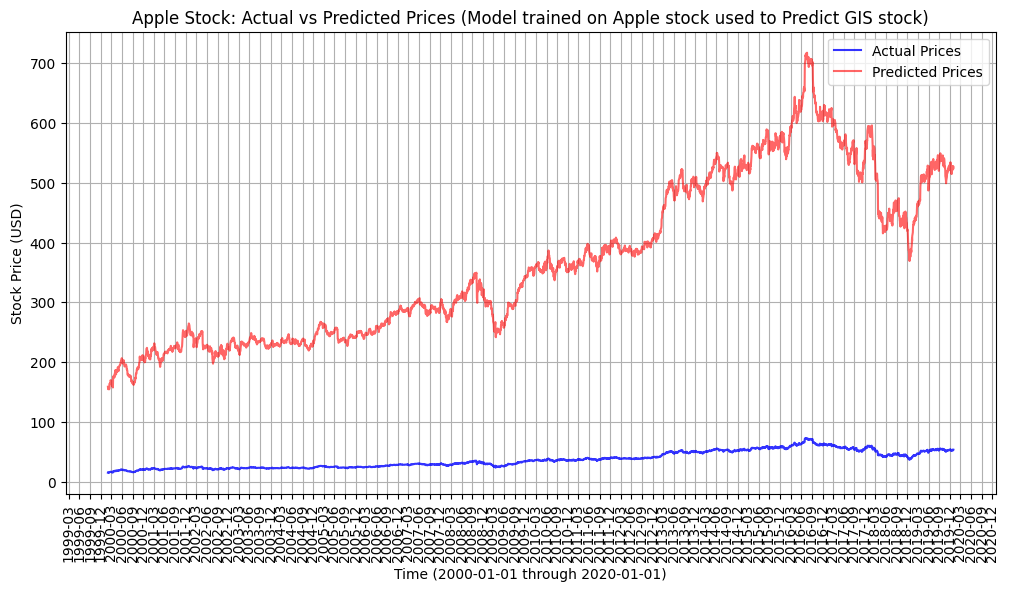
* Trained model (Technology Industry) predicting on Nvidia Stock (Technology Industry)
  + Mean Squared Error (MSE): 481.2389590100721
  + Mean Absolute Error (MAE): 17.24226981276846



* Trained model (Technology Industry) predicting on ADM Stock (Food Industry)
  + Mean Squared Error (MSE): 87714.49943855987
  + Mean Absolute Error (MAE): 275.91621841172974



* Trained model (Technology Industry) predicting on GIS Stock (Food Industry)
  + Mean Squared Error (MSE): 124301.7595811832
  + Mean Absolute Error (MAE): 330.9713993685328



# 

# 7. Conclusion and Future Considerations

## 7.1 CONCLUSION

After training various different models on stocks from different industries, we found that for predicting a single stock’s future closing price, the MLP model and Linear Regression performed the best with low MAE and MSE scores (MLP Test MSE: 0.4465, MLP Test MAE: 0.4621, Linear Test MSE: 0.3856, Linear Test MAE: 0.4295). When we tried to predict the prices of other stocks within the same industry as the stock that models were trained on, the models performed considerably worse with high error scores. The same was observed when predicting other stocks outside the industry of the stock that the model was trained on. Since some models were able to perform better predicting certain stocks within the same industries and some models were able to perform better predicting certain stocks in a different industry, there seems to be little correlation between the behavior of the stocks and their corresponding industries. However, regardless of industry, performance suffered when predicting other stocks that differed than the one that the model was trained on. For the stocks in separate industries, it was observed that when prices between the train and test stocks were similar, there was better performance (See ***Section 6***).

This is expected, as stocks in different industries tend to behave differently than stocks within the same industry, so models will struggle to predict prices in different industries accordingly. Furthermore, it is expected that models perform better in predicting the future prices of the stock that they are trained on compared to different stocks within the same industry. While the different stocks within the same industry may have similar characteristics, their behaviors are still different from one another, hence the lower accuracy when using certain models to predict other stocks other than those that it was trained on. In other words, these models were not able to generalize well to other stocks, but showed some better performance to stocks in the same industry compared to those that were not.

In addition, we noticed that the high volatility of certain stocks made it difficult to accurately predict large spikes and drops in closing prices. The stock market is truly random at times and these sudden unprecedented changes that deviate from patterns observed in the past are hard to predict. Furthermore, since we used a relatively small window of data looking at only the current day’s price or even just the past few days, our models were more likely to pick up rapid, short term.

In short, it appears that the best results can be achieved by training a model individually for each stock and having that model predict the future closing prices for that one stock rather than attempting to train a model that can be generalized to other stocks. However, perhaps further study could be completed and other methods could be explored to see if there could be more success across predicting different stocks with the same model.

## 7.2 FUTURE CONSIDERATIONS

For future considerations, we can explore models geared towards short term and long term investment trading as these two games have different rules to play by. Long term trading often involves analyzing a longer period of data and determining the growth or decline of the stock. This can suggest taking a longer moving window average, incorporating bollinger bands, stochastic oscillations, factoring in relative strength index (RSI), and other dynamic market influencers used by seasoned stock traders. Short term trading can take place rapidly with market volatility so a model that can respond and predict jumps in the data is important. We can consider using bollinger bands, smaller previous moving averages, average directional index, and MACD indicator. In dealing with short term trading, our model should be complex enough to predict sudden changes in the stock price as opposed to long term where we focus more on the trend of the stock throughout the years.

Another feature for our stock predictor model would be to consider multi-horizon prediction where the model predicts on multiple time scales (1 day, 5 days, 30 days, etc) to balance the need for short-term responsiveness and long-term pattern identification. This allows the user to view and analyze all the different time scales at once without having the need to go through each individually.