Stock Market Prediction Milestone Report

Predicting Stock Market prices can be incredibly difficult. There are no consistent patterns in stock market data that allow an analyst or investor to model stock prices over time with near-perfect accuracy. This is due to the high number of rapidly changing and difficult to forecast factors upon which the movement of stock prices can depend. The current advances in machine learning techniques and algorithms have made their implementation for prediction of stock prices a popular area of research. The results have been promising and several different techniques and algorithms have been used. In this project I have used four different time series machine learning models for predicting future rises or falls in stock market price movements. My initial forecast was predicted using an LSTM model that was then compared to a liner regression model, decision tree model, and finally an XGBoost model. As a final step, the one day moving average was also determined using a standard averaging technique and the exponential moving average. The purpose of this was to compare the long-range prediction accuracy to single step accuracy.

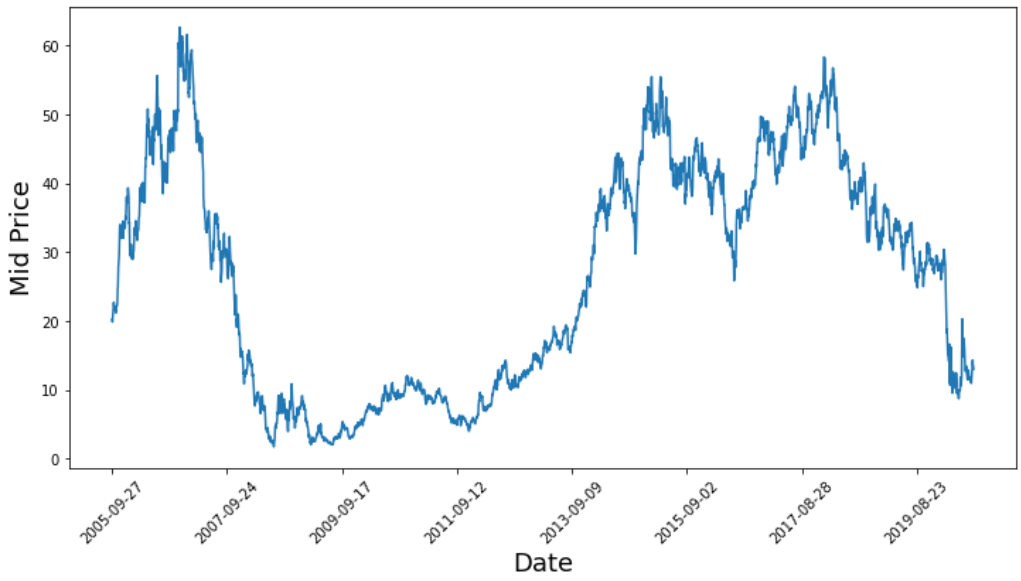
Why was it important to use time series modeling methods?

In order to accurately decide when to buy stocks and when to sell them to make a profit, it is necessary for the machine learning model to have the ability to receive as input the history of a sequence of data and use it to correctly predict what the future elements of the sequence will going to be.

Getting Data from Alphavantage

I obtained an API key from Alphavantage and wrote it into my code. I then set a ticker variable and a URL variable that can be edited to enable the code to be used on different stocks based on the user's assignment of the ticker and URL variable. Obtaining the data in this way returns a JSON file with all the stock market data for the ticker specified. The data was then be extracted from the JSON file and the date, open, high, low, and close values were converted into a pandas Data frame.

Initial Visualization



Initial data summary using df.info()

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3748 entries, 0 to 3747
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date    3748 non-null   object
 1   Low     3748 non-null   object
 2   High    3748 non-null   object
 3   Close   3748 non-null   object
 4   Open    3748 non-null   object
dtypes: object(5)
memory usage: 175.7+ KB
```

Based on the visualization and informational summary, there is some large variability and fluctuation in the data. There was a significant change in the pattern of behavior between 2007 and 2013 that needed to be accounted for during data normalization.

Data Pre-processing

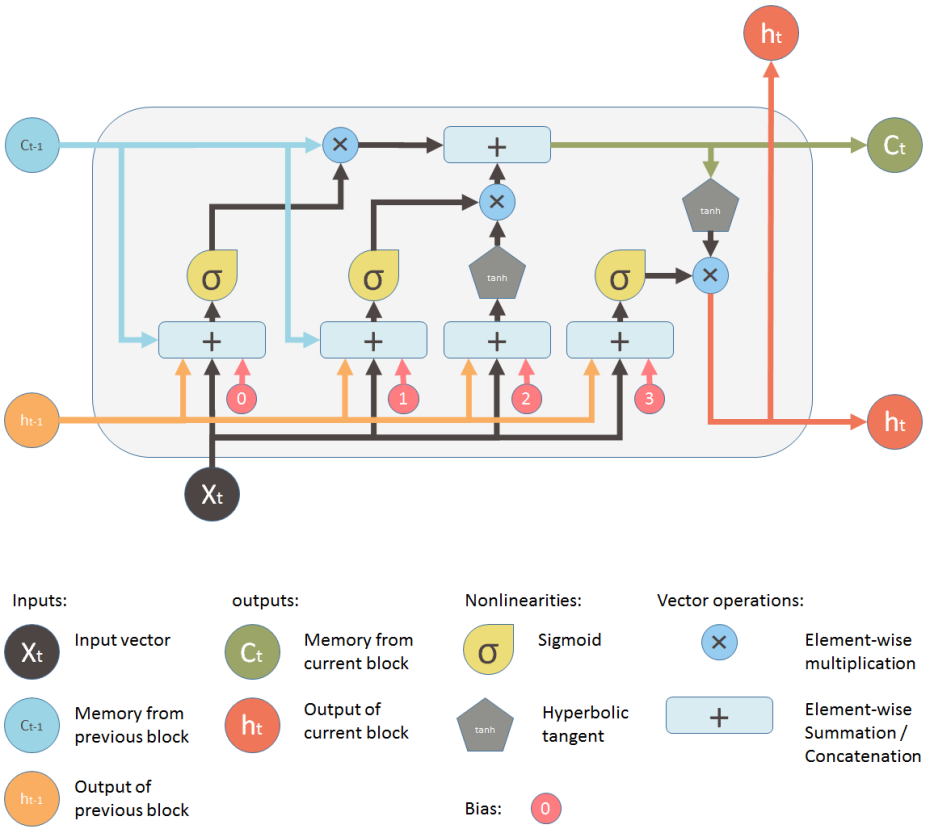
The data was split into a training and testing set using 3230 as the dividing index. The data was normalized between 0 and 1 using Scikit Learns MinMaxScaler function. Due to the large changes in the behavior of the data over time, the training data was normalized in batches to prevent the large values from overwhelming the smaller values and negating their influence of their patterns on the learning of our modeling. The normalization window size was chosen for the training data was 800 data points per window. The testing data was normalized in one large batch.

Review of LSTM Model Architecture

A recurrent neural network (RNN) can be thought of as a series of multiple copies of the same network, each passing a message to the next successor. This allows them to be able to connect previous information to the present task. LSTM neural networks are a special kind of recurrent neural network which work, for many tasks, much better than the standard RNN. LSTMs are explicitly designed to avoid the long-term dependency problem (remembering information for long periods of time).

The LSTM basic structure is based off the cell state. The cell state receives information from prior predictive layers based on the permissions granted from a series of regulatory gates. It then serves as the carrier of information into the next layer.

The following is a summary of the regulatory gates that determine the importance of information sent to the cell and determine it's inclusion into the cell state to be carried forward for formulating the final model prediction. The visual aid is included to aid the understanding of information flow and the points of decision.



https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714

$$
\begin{aligned}
\mathbf{i}_t &= \sigma\left(\mathbf{W}^{(xi)}\mathbf{x}_t + \mathbf{W}^{(hi)}\mathbf{h}_{t-1} + \mathbf{W}^{(ci)}\mathbf{c}_{t-1} + \mathbf{b}^{(i)}\right) \\
\mathbf{f}_t &= \sigma\left(\mathbf{W}^{(xf)}\mathbf{x}_t + \mathbf{W}^{(hf)}\mathbf{h}_{t-1} + \mathbf{W}^{(cf)}\mathbf{c}_{t-1} + \mathbf{b}^{(f)}\right) \\
\mathbf{c}_t &= \mathbf{f}_t \bullet \mathbf{c}_{t-1} + \mathbf{i}_t \bullet \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right) \\
\mathbf{o}_t &= \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right) \\
\mathbf{h}_t &= \mathbf{o}_t \bullet \tanh\left(\mathbf{c}_t\right),
\end{aligned}
$$

Inputs to the current cell in the recurrent chain include the output from the previous cell (ht), the new input of the current time step (xt), and the memory from the previous cell (ct).

The first gate is the forget gate. The forget gate is responsible for receiving ht, ct, and xt and determining this information's importance. The information that is determined non-contributory to the strength of the LSTMs predictive power is removed via multiplication by a filter. Removal of inert information is required for optimizing the performance of the LSTM network. The equation for this procedure is outlined above. If the sigmoid function outputs a 0, the information is forgotten, if the sigmoid function outputs a 1, the information is retained and passed forward.

The second gate is called the input gate. This gate receives the same inputs as the forget gate and ensures new information added to the cell state is important and is not redundant. The general equation for this procedure is also outlined above. This addition of information is basically three-step process as seen from the diagram.

The third and final gate is the output gate. The job of the output gate is the generation of the output of the current LSTM cell and the selection of useful information from the current cell state that should be passed as output to the next LSTM cell. The general equations for this process are also above.

To summarize

- **The input gate:** The input gate adds information to the cell state
- **The forget gate:** The forget gate removes the information that is no longer required by the model
- **The output gate:** The output Gate selects the information to be shown to the next step in the sequence

Data Generation

The first step was to implement a data generator whose output was a set of N/b batches of input data obtained sequentially. Each batch in the batch set had a set batch size (b). Each batch of input data had a corresponding output batch of data of the same size. The corresponding output batch to each input batch was generated by randomly selecting one value from the original data set that was between x+b and x+b+5 values ahead. The purpose of generating the output batch using this method is to reduce the risk of overfitting to the training data. The visual representation of this looks like:

Input Data:
[x0,x10,x20,x30],[x1,x11,x21,x31],[x2,x12,x22,x32]

Corresponding Output Data:
[x1,x11,x21,x31],[x2,x12,x22,x32],[x3,x13,x23,x33]

Initial Hyperparameter settings

There were 5 hyperparameters that needed to be determined

- The dimensionality of the data, which is always 1
- The number of continuous time steps considered for each optimization step, initialized at 50
- The number of data points in each batch (or time step), initialized at 500
- The number of LSTM layers, 3 are used for this model
- The number of nodes in each of the three layers of the LSTM cell, initialized at [200, 200, 150]
- The dropout percent, initialized at 20%

Loss and Optimization

The mean squared error (MSE) of a model prediction model is defined as the average of the squares of the errors. It is the average squared difference between the predicted values and the actual value. The loss function chosen to monitor the LSTM's model performance was the Mean Squared Error. The advantage of choosing the Mean Squared Error is the ability for this error metric to identify extreme values during training and allow for minimizing the impact of extreme values on the final model. For each batch of predictions and corresponding observed outputs, the Mean Squared Error was determined by adding the mean squared losses of all predicted values in the batch together.

The optimizer chosen to optimize the neural network was Adam. This is a widely used and very well-performing optimizer.

Running the LSTM

Training and prediction of stock price movements using the LSTM was performed for 3 epochs using the following procedure:

Define a test set of starting points on the time series to evaluate the model on

For each epoch

1) For full sequence length of training data
    - Unroll a set of batches
    - Train the neural network with the unrolled batches

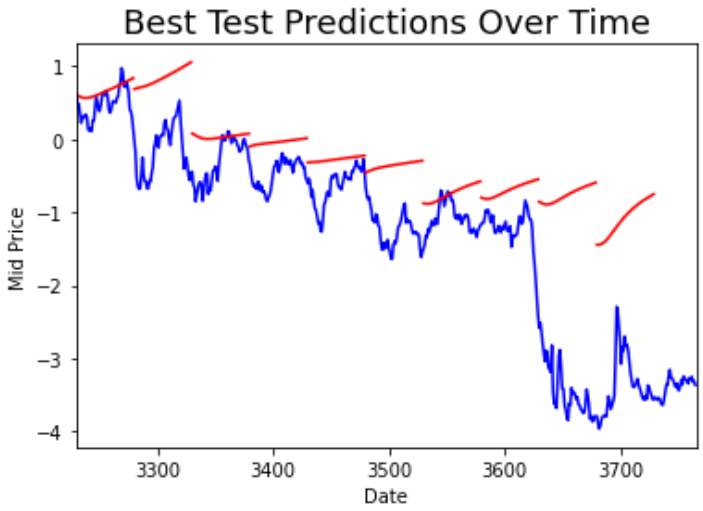
2) Calculate the average training loss

3) For each starting point in the test set
    - Update the LSTM state by iterating through the previous data points found before the test point
    - Make predictions for 50 steps continuously, using the previous prediction as the current input
    - Calculate the MSE loss between the 50 points predicted and the true stock prices at those time stamps

Initial results

After 30 epochs, the epoch with the best Mean Squared Error score was epoch thirty. The MSE at epoch thirty was 58.598 and the validation loss was 8.397. These initial results are very good. My next strategy will be to perform further hyperparameter tuning. I will also test the generalizability of the model by testing it on different tickers from the Alpha Vantage API.

Visualization of Initial Results



Linear Regression for Stock Market Prediction

The most basic machine learning algorithm that can be implemented on this data is linear regression. The linear regression model returns an equation that determines the relationship between the independent variables and the dependent variable. In the short term, stock prices are considered random and fluctuate with a large amount of noise. In the long term, stock prices tend to develop linear relationships. It is possible to frame the problem of predicting future stock prices with a regressor algorithm**.**

The equation for linear regression can be written as:

$Y = w0x0 + w1x1+w2x2+……..+wnxn+b$

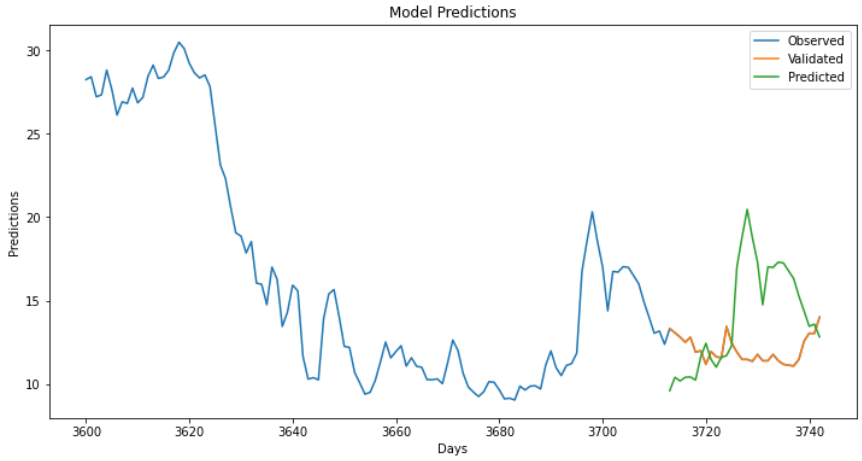
Where, x_1, x_2,….x_n represent the independent variables while the coefficients w_1, w_2, …. w_n represent the weights.

In time series prediction, the time series are typically expanded into three or higher-dimensional space by restructuring the series into a data frame to look like a supervised learning problem. This can be done by using previous time steps as input variables and assigning future time step as the output variable. The use of prior time steps to predict future steps is called the sliding window method.

The first step in preparing the data imported from Alpha Vantage for linear regression was to create a new data frame containing the close prices as features and the corresponding price at 30 days into the future as targets. The close price column was then isolated as a Numpy array of inputs (X) and the column of 30 day offset prices was isolated as a Numpy array of targets (y). The dataset was split into an 80% training set for creation of the model and 20% testing set for model validation.

The loss function chosen was Mean Squared Error. The results of training a Linear Regression model on the training set and validating on the test set was a mean squared error of 19.114. The results of predicting on the final 30 days of the dataset using the trained model was a Mean Squared Error of 0.191. This rivaled the results of the output on the LSTM model.

Below is a visualization of the 30-day prediction vs the validated prediction.

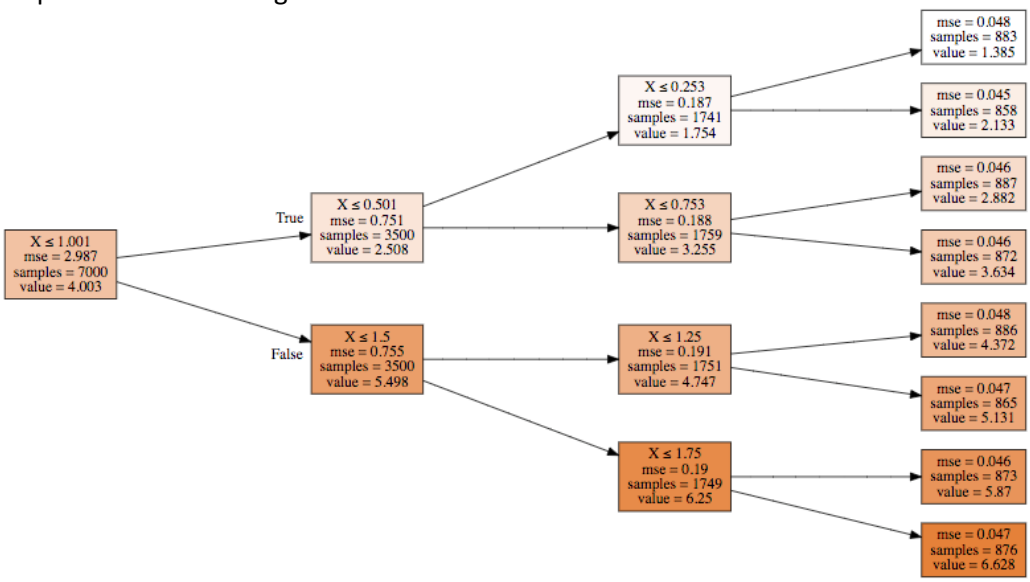


Decision Tree Regression for Stock Market Prediction

Another commonly implemented regression algorithm is the Decision Tree regression model. For regression trees, the value of terminal nodes is the mean of the observations falling in that node. Therefore, if an unseen data point falls in the range of values in the terminal node, we predict using the mean value.

Initially, all values in the training set are grouped into one large node. The algorithm then begins allocating the data into the first two partitions or branches, using every possible binary split on every field. The algorithm selects the split that minimizes the sum of the squared deviations from the mean in the two separate partitions. The same splitting rule is then applied to each of the new branches. This process continues until each node reaches a user-specified minimum node size and becomes a terminal node.

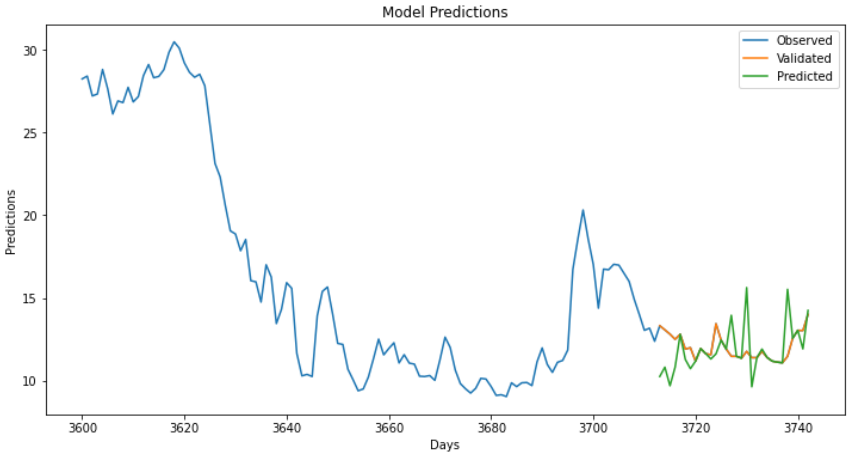
Below is a visual representation of the algorithm:



The same data preparation steps used for the Linear Regression model were used for training, validating, and predicting with the regression tree. See above for details

The loss function chosen was Mean Squared Error. The results of training a Regression Tree model on the training set and validating on the test set was a mean squared error of 35.598. The results of predicting on the final 30 days of the dataset using the trained model was a Mean Squared Error of 3.755. The regression tree was significantly outperformed by the linear regression and LSTM models. Next, I will try to improve the performance of the decision tree model with hyperparameter tuning.

Below is a visualization of the 30-day prediction vs the validated prediction. The model appears to be overfitting the training data. This is a common problem with decision tree algorithms.



XGBoost for Stock Market Prediction

Extreme Gradient Boosting, also called XGBoost, is a machine learning model that has attracted a lot of attention from the machine learning practitioners attempting to analyze and forecast stock markets. Extreme Gradient Boosting (XGBoost) models are an implementation of the gradient boosting framework. Gradient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction

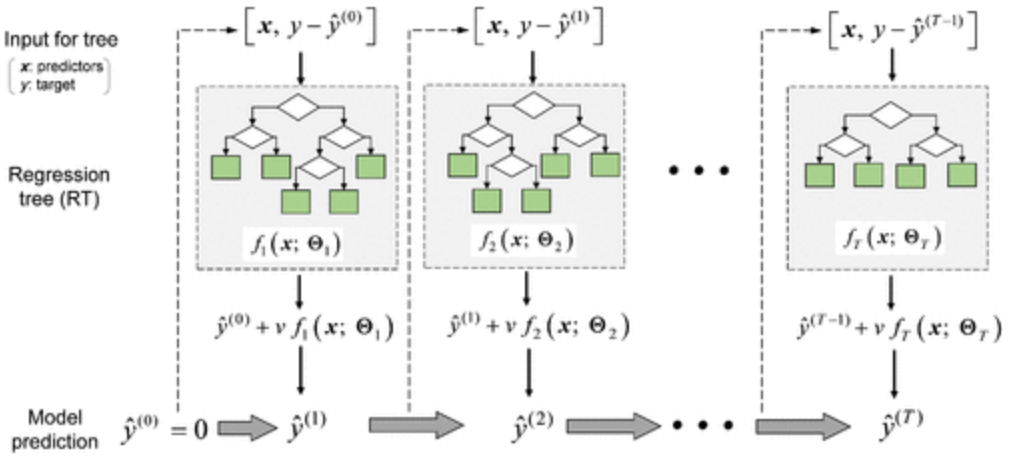
The objective of the XGBoost model is given as:

$$Obj = L + \Omega$$

Where, L is the loss function which controls the predictive power, and Ω is regularization component which controls generalizability and overfitting.

The loss function (L) which needs to be optimized can be Root Mean Squared Error for regression. The regularization component (Ω) is dependent on the number of leaves and the prediction score assigned to the leaves in the tree ensemble model.

The difference between gradient boosting and XGBoost is, XGBoost is a more regularized form of Gradient Boosting. XGBoost uses advanced regularization (L1 & L2), which improves model generalization capabilities. XGBoost delivers higher performance as compared to Gradient Boosting. Below is a visual diagram of the general concept



For this project I created three different models, with being an improvement of the previous model. The same data used for the Linear Regression and Decision Tree Regression models was used for the XGBoost modeling, so the only additional pre-processing necessary was the conversion of the training and validation datasets to DMatrices. This is a requirement for input into the XGBoost infrastructure.

The first model was a basic XGBoost model with default parameters and 5 boosting rounds specified. The resulting Mean Squared Error was 29.536.

To improve on the first model I created a cross validated XGBoost model with the following parameters:
Max tree depth: 4
Number of validation folds: 4
Number of boosting rounds: 5

The results of this model are outlined in the below table. The cross validation did not have improved results over the defaulted XGBoost model.

	train-rmse-mean	train-rmse-std	test-rmse-mean	test-rmse-std
0	21.986686	0.136333	21.987640	0.446506
1	15.719848	0.093633	15.735410	0.340168
2	11.434488	0.061982	11.470236	0.257726
3	8.569930	0.040131	8.630230	0.185066
4	6.721699	0.025848	6.809362	0.138029

The final attempt at improving the performance of the XGBoost model was to further tune the hyperparameter choices. I performed Hyperparameter tuning using the Randomized Search Cross Validation method. The parameter ranges I searched over were:

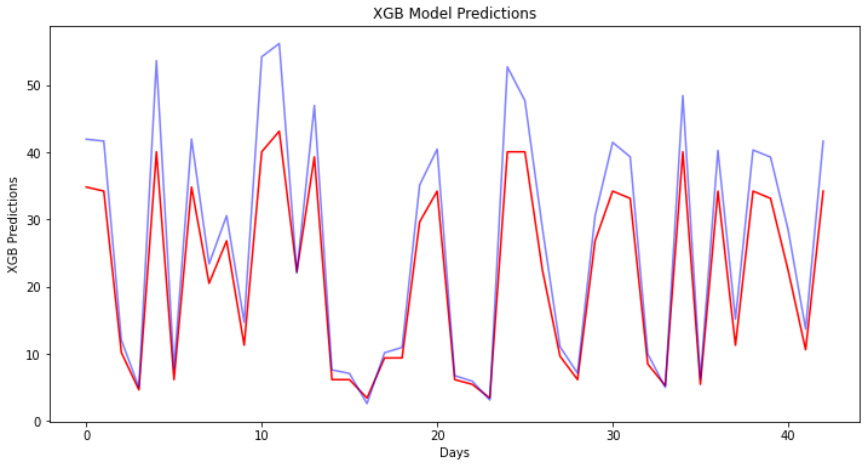
Samples per tree: [0.2, 0.3, 0.4, 0.5, 0.6, 0.7]
Learning Rate: [0.04, 0.06, 0.08, 0.1, ……, 0.98, 1.0]
Maximum Tree Depth: [3,4,5,6,7,8,9,10]
Number of Estimators: [50, 100, 150, 200]

I trained the Randomized search for 5 different iterations cross validated for 4 folds. The best parameters found were:

Samples per tree: 0.2
Learning Rate: 0.32
Maximum Tree Depth: 3
Number of Estimators: 100

The lowest RMSE found was 5.344. This is a small improvement over the default and cross validated XGBoost models.

Using the best parameters found, I validated the model on unseen data using the test set. The resulting Mean Squared Error was 29.470. Below is a visualization of the results



The XGBoost model was outperformed by all prior models created, even with a robust hyperparameter tuning algorithm to find the optimal model.

One Day Ahead Prediction Modeling

Despite the challenges of predicting stock market price movement for time periods far into the future, predicting day to day stock market movement (one step ahead), has been much more successful. Out of curious comparison, I created 2 models to predict single day market movements and compared the performance to the future prediction models. First, I created a model to predict the simple moving average. I then created a model to predict the exponential moving average.

A simple moving average (SMA) is an arithmetic moving average calculated by adding recent prices and then dividing that by the number of time periods in the calculation average. The SMA is a technical indicator that can aid in determining if an asset price will continue or reverse a bull or bear trend. The equation for calculating the SMA Is:

SMA = (A_1 + A_2 + ... + A_n)/n

where:
A_n = the price of an asset at period n

N = the total number of periods

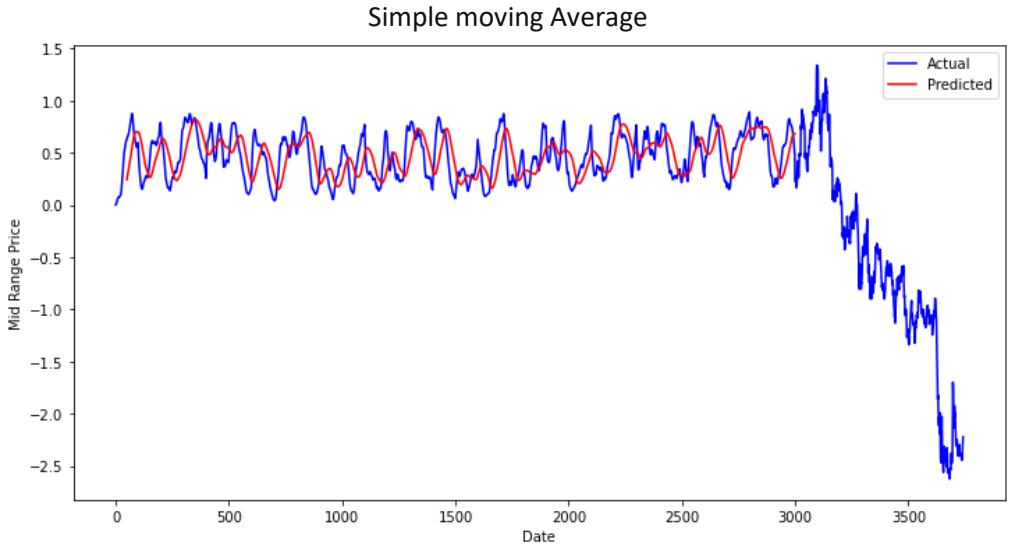
The exponential moving average (EMA) is a type of moving average that places a greater weight and significance on the most recent data points. An exponentially weighted moving average reacts more significantly to recent price changes than a simple moving average, which applies an equal weight to all observations in the observation time frame. The equation for calculating the EMA is:

*EMA*Today = (ValueToday∗(Smoothing/1+Days)) + *EMA*Yesterday∗(1−(Smoothing/1+Days))

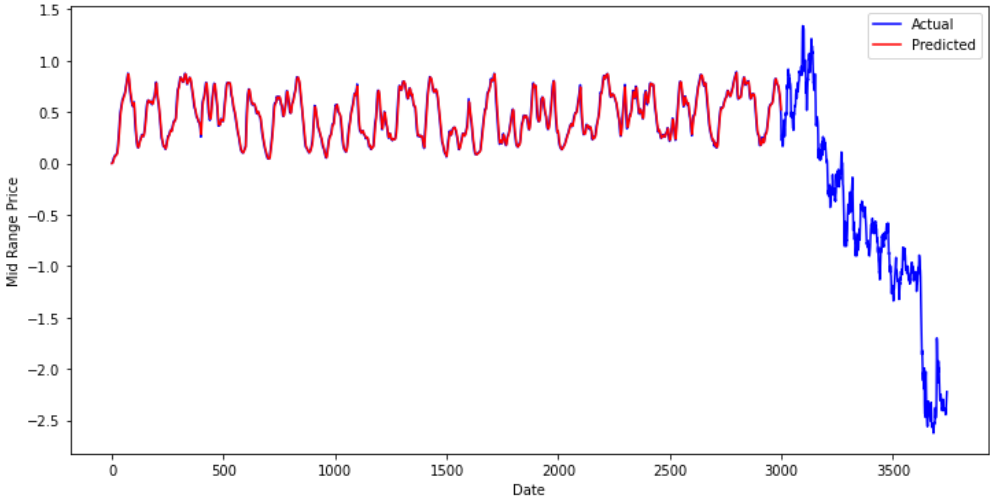
The major difference between the exponential moving average (EMA) and the simple moving average is the sensitivity each one shows to changes in the data used in its calculation. The EMA gives a higher weighting to recent prices, while the SMA assigns equal weighting to all values. Both averages are interpreted in the same manner and are both commonly used by traders to smooth out price fluctuations. Since EMAs place a higher weighting on recent data than on older data, they are more reactive to the latest price changes than SMAs.

The simple moving average was calculated with a window size of 50 days and resulted in an MSE error of 0.410. The exponential moving average was also calculated with a window size of 50 days and a decay rate of 0.5. The resulting MSE error was 0.007.

Below are visualizations for the simple moving average and the exponential moving average predictions



Exponential Moving Average



These models are highly successful for short term predictions, however, when attempts are made to predict for longer periods into the future, their predictions become repetitive and accuracy declines.

Initial Conclusions and Next Steps

The progress so far demonstrates the models with the best predictive power to be the Linear Regression model and the LSTM model. The success of tree-based models has been poor, however this may be a matter of more thoughtful hyperparameter tuning. The next steps in this project will be improvement of each model's accuracy through hyperparameter tuning. I will also test the generalizability of all models by testing them on multiple tickers to expose them to new data.

Phase 2 Hyperparameter Tuning for Linear Regression and Decision Tree Regression

Linear Regression with Ridge Regularization

Ridge Regression is a technique for analyzing multiple regression data that suffer from multicollinearity. When multicollinearity occurs, least squares estimates are unbiased, but their variances are large so they may be far from the true value. By adding a degree of bias to the regression estimates, ridge regression reduces the standard errors. It is hoped that the net effect will be to give estimates that are more reliable.

The ridge regression algorithm works by imposing a penalty on the size of the coefficients according to the following equation:

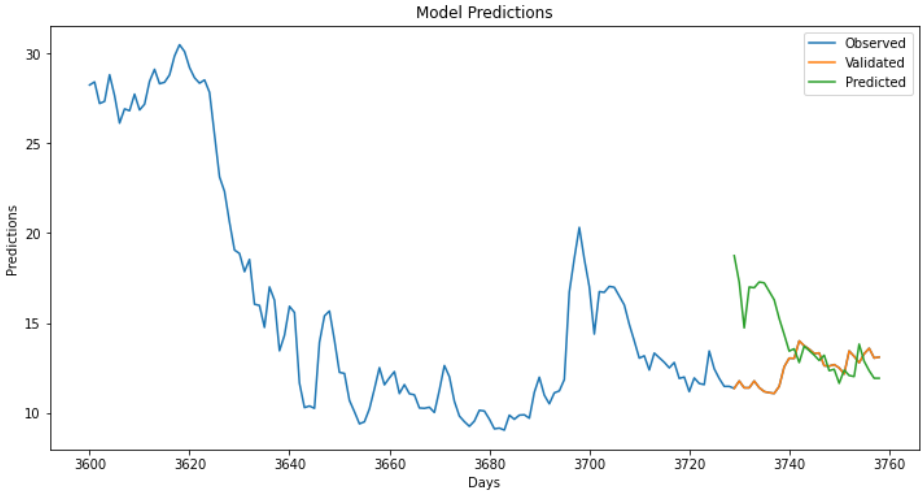
$$\min_{w} ||Xw - y||_2^2 + \alpha ||w||_2^2$$

The complexity parameter $\alpha \geq 0$ controls the amount of coefficient shrinkage. The larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to collinearity.

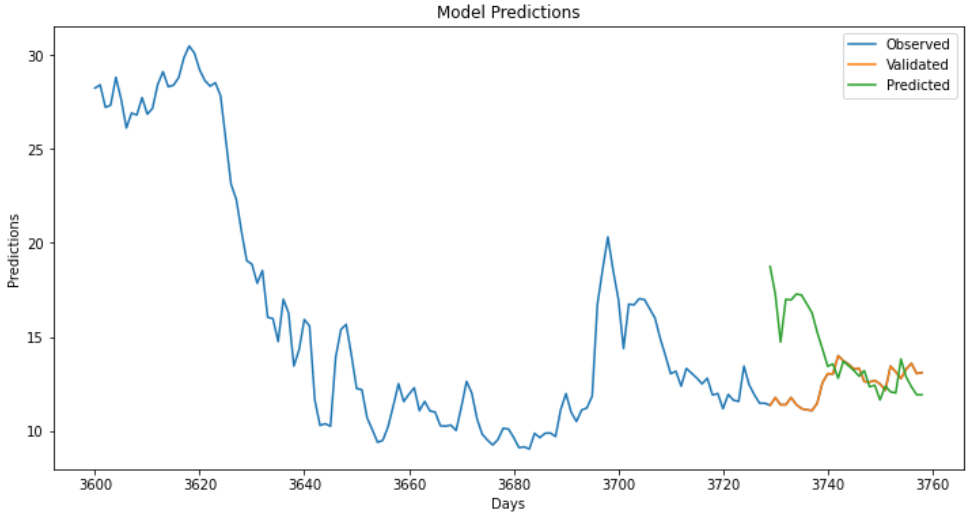
In this project I attempted 2 different version of the ridge regression model. The first was a basic ridge regression model and the second was a cross validated ridge regression model. The basic ridge regression model was given a static alpha of 0.5. The cross validated ridge regression with an alpha search range between 10^{-6} and 10^{6}.

Both the basic ridge regression model and the cross validated ridge regression models returned MSE values of 0.190 on future predictions. The visualized results of both models are below. You can see there is very little difference in the results of both models. Conclusively, the ridge regression model offered no additional benefit over the simple ordinary least squares model. All three versions of linear regression attempted to this point in the project returned MSE values of 0.190 on future predictions.

Basic Ridge Regression



Ridge Regression with 10 Fold Cross Validation



Linear Regression with Lasso Regularization

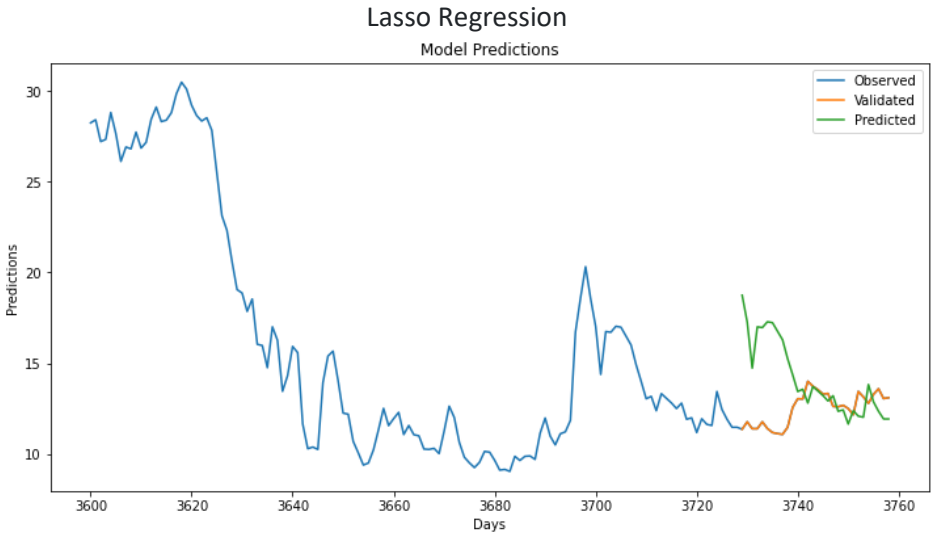
Lasso regression stands for Least Absolute Shrinkage and Selection Operator. Like ridge regression, the algorithm adds a penalty term to the cost function. As the value of coefficients increases from 0 the absolute sum term increases the penalty and decreases the value of coefficients to reduce loss. Unlike ridge regression, the penalty term for lasso regression is the absolute sum of the coefficients multiplied by alpha. The difference in effect between ridge and lasso regression is that the lasso algorithm can eliminate some features by reducing coefficients to absolute zero. Ridge regression which never sets the value of coefficient to absolute zero.

The formula for lasso regression is below. Mathematically, it consists of a linear model with an added absolute value regularization term. The objective function to minimize is:

$$\min_{w} \frac{1}{2n_{\text{samples}}} ||Xw - y||_2^2 + \alpha||w||_1$$

where α is a constant and ||w||1 is the ℓ1-norm of the coefficient vector.

In this project the lasso regression model was built with an alpha value of 0.1. This regression model returned a MSE value of 0.195 on future predictions. The visualized results are below. You can see there is very little difference in the results of this model in comparison to the ordinary least squares regression model. Taken into the context of all linear models attempted in this project, all four versions of linear regression returned MSE values of 0.190 – 0.195 on future predictions with almost identical results.



Decision Tree Regression with Randomized Hyperparameter Search

The poor performance of the decision tree model above appears to be resulting from the tendency of the algorithm logic to overfit on specific data points rather than fitting to trends in the data. This leaves a lot of opportunity for hyperparameter tuning to control overfitting and improve the accuracy of predictions. Choosing specific hyperparameter combinations to optimize the performance of the model can be a guessing game and the best way of efficiently finding the best combination of hyperparameters is using a Cross Validation Randomized Parameter search similar to the technique used for the XGBoost model. The parameter grid used to search within included:

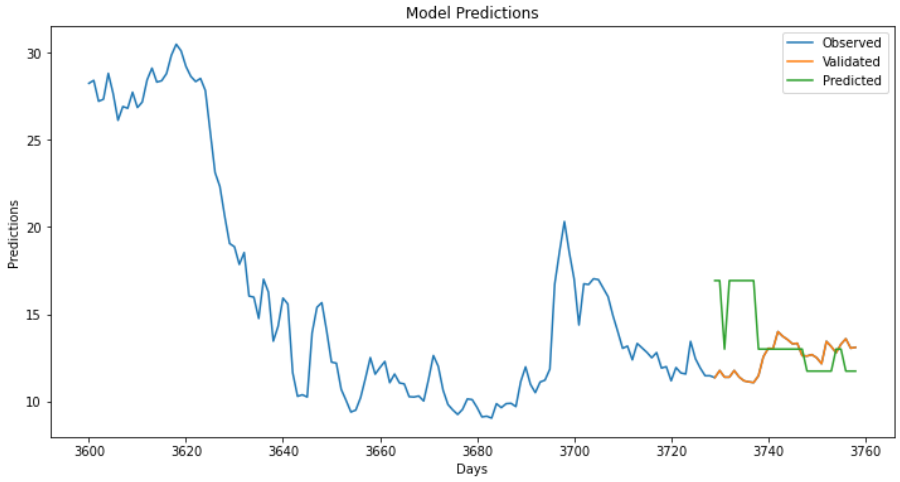
Maximum Tree Depth: [2,3,4,5,6,7,8,9,10]
Maximum Features to Consider for Best Split: [auto , sqrt]
Quality of Split Criterion : [mse, mae]
Node Split Strategy: [best, random]
Minimum Samples per Leaf: [5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

I trained the Randomized search for 10 different iterations cross validated for 5 folds. The best parameters found were:

Maximum Tree Depth: 6
Maximum Features to Consider for Best Split: sqrt
Quality of Split Criterion : mean squared error
Node Split Strategy: best
Minimum Samples per Leaf: 5

This decision tree regression model returned a MSE value of 0.368 on future predictions. The visualized results are below. This is a dramatic improvement over the prior decision tree prediction of 3.755.

Decision Tree Regression after Randomized Hyperparameter Search



Consolidated Results

The table below shows the mean squared error for future predictions made by all the models built

Model	Mean Squared Error
One Day Ahead, Exponential Moving Average	0.007
Ordinary Least Squares Linear Regression	0.190
Ridge Regression	0.190
Lasso Regression	0.195
Decision Tree with Hyperparameter Search	0.368
One Day Ahead, Simple Moving Average	0.410
Decision Tree	3.755
XGBoost with Hyperparameter Search	29.470
XGBoost, default	29.536
XGBoost with Cross Validation	46.362
LSTM Neural Network	58.598

Initial Conclusions

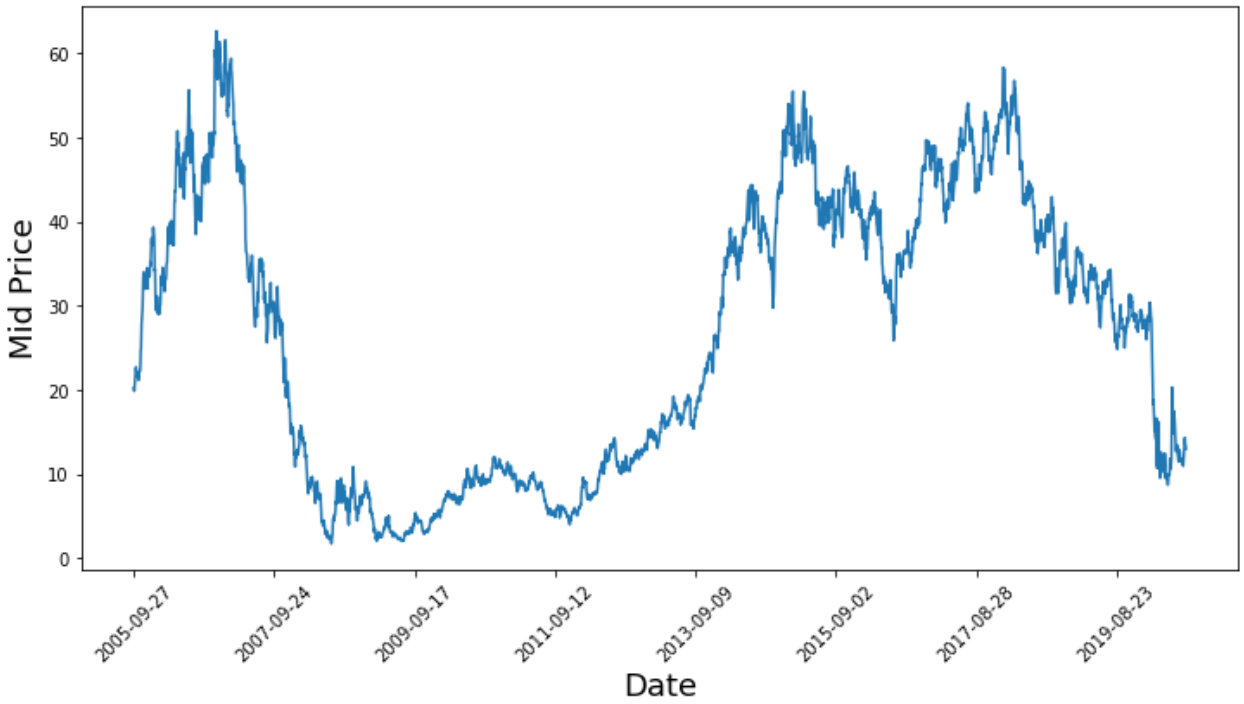
- Ordinary Least Squares Regression provides the best results for future predictions
- One day ahead predictions are the most reliable but are limited in use for some purposes that require longer range insights
- Models with a tendency towards overfitting are less effective for stock market prediction even after robust hyperparameter tuning

Generalizability

Another test of a model's ability to predict future stock market movement is to investigate its performance on multiple companies with different historical trends. The historical trend variations between companies are as unique as the companies themselves. A robust model would have the ability to accurately predict stock market movement reliably regardless of the shape of the time series graph. This would imply the model's ability to identify relationships in the data that are not easily detectable or obvious. For this project, the entire model series was trained and tested on historical stock market data for AAL, GOOGL, AAPL, and TSLA.

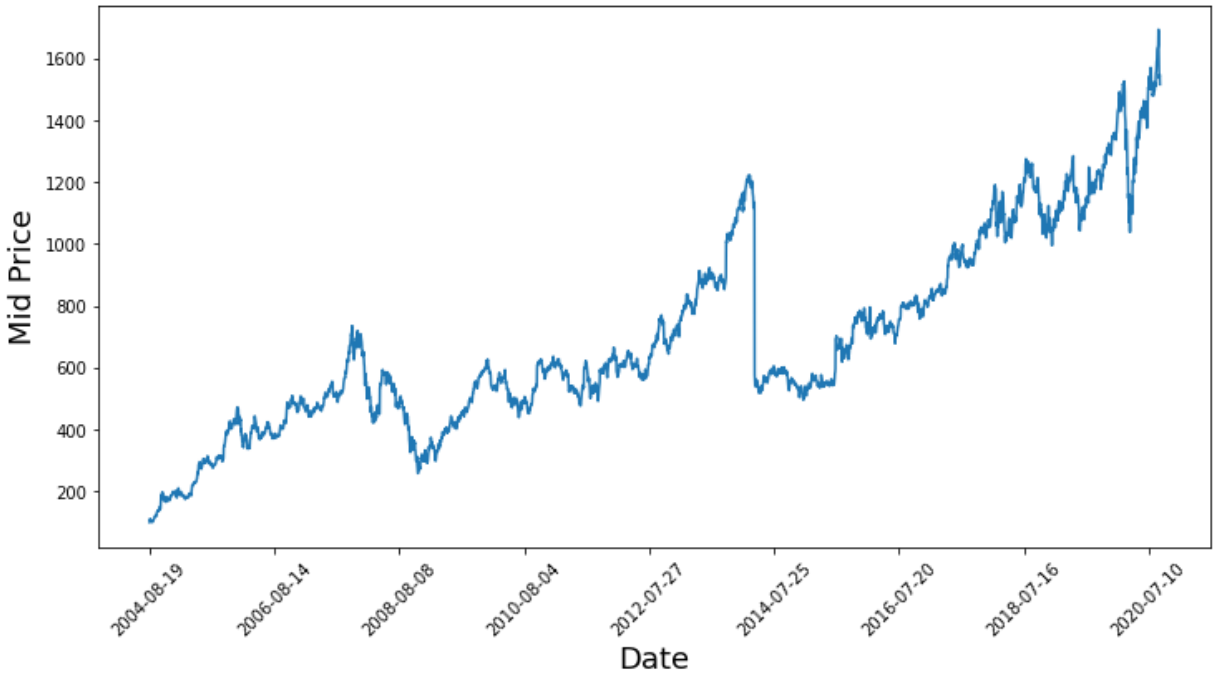
AAL

Model	Mean Squared Error
One Day Ahead, Exponential Moving Average	0.007
Ordinary Least Squares Linear Regression	0.190
Ridge Regression	0.190
Lasso Regression	0.195
Decision Tree with Hyperparameter Search	0.368
One Day Ahead, Simple Moving Average	0.400
Decision Tree	3.755
XGBoost with Hyperparameter Search	29.470
XGBoost, default	29.536
XGBoost with Cross Validation	46.362
LSTM Neural Network	58.598



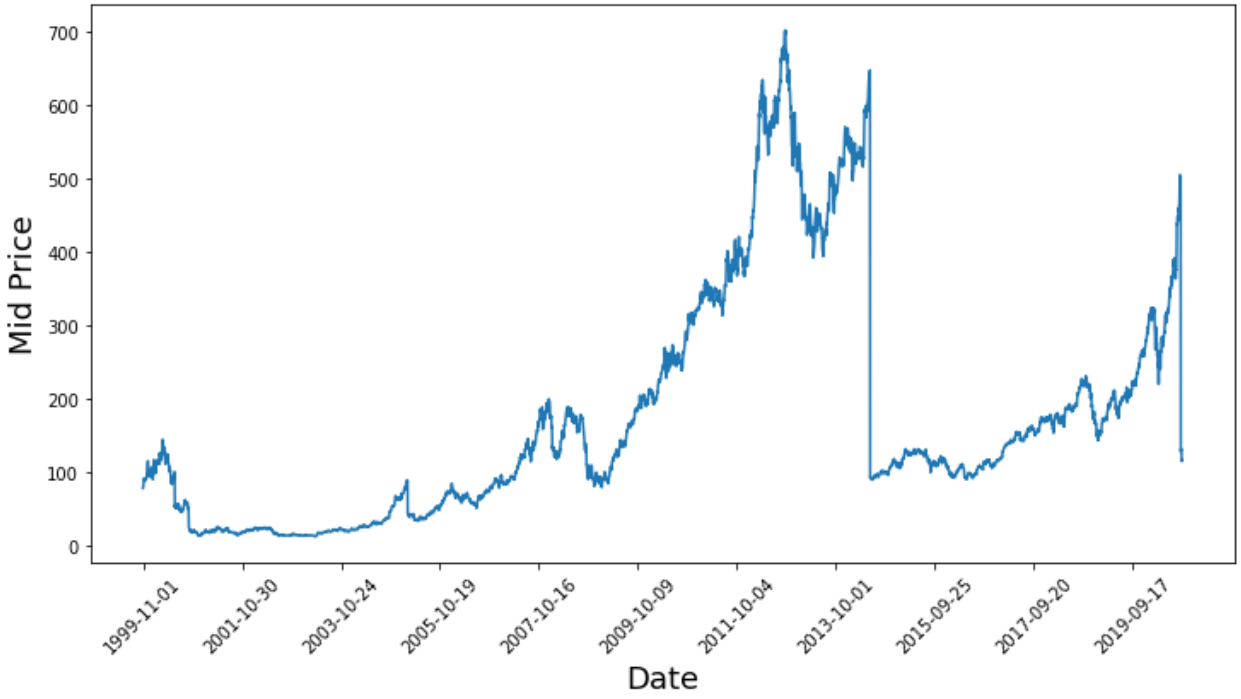
GOOGL

Model	Mean Squared Error
One Day Ahead, Exponential Moving Average	5.000
Ordinary Least Squares Linear Regression	10.697
Ridge Regression	10.697
Lasso Regression	10.702
One Day Ahead, Simple Moving Average	40.000
Decision Tree with Hyperparameter Search	1021.567
Decision Tree	6990.854
XGBoost, default	14913.590
XGBoost with Hyperparameter Search	14970.071
XGBoost with Cross Validation	23545.061
LSTM Neural Network	38460.100



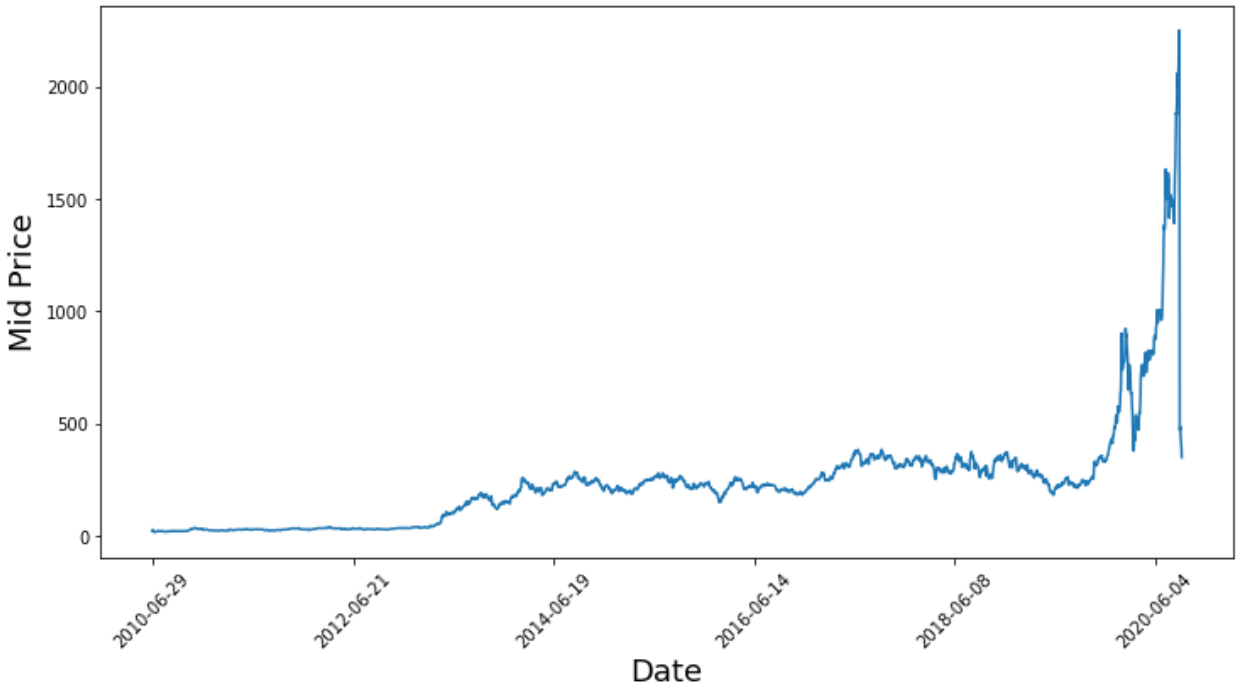
AAPL

Model	Mean Squared Error
One Day Ahead, Exponential Moving Average	0.100
One Day Ahead, Simple Moving Average	6.000
Ordinary Least Squares Linear Regression	39.460
Ridge Regression	39.460
Lasso Regression	39.470
Decision Tree with Hyperparameter Search	570.927
LSTM Neural Network	1902.000
XGBoost, default	2075.320
XGBoost with Hyperparameter Search	2143.251
XGBoost with Cross Validation	3469.328
Decision Tree	17819.173



TSLA

Model	Mean Squared Error
One Day Ahead, Exponential Moving Average	1.000
One Day Ahead, Simple Moving Average	80.000
LSTM Neural Network	1592.000
XGBoost with Hyperparameter Search	2316.076
XGBoost, default	2853.577
XGBoost with Cross Validation	11024.58
Lasso Regression	17793.782
Ridge Regression	17794.482
Ordinary Least Squares Linear Regression	17794.484
Decision Tree with Hyperparameter Search	209766.112
Decision Tree	467594.970



Referrences:

https://www.programmersought.com/article/5487636739/

https://wingkiwong.wordpress.com/2019/01/13/long-short-term-memory-in-time-series-data/

https://towardsdatascience.com/using-lstms-for-stock-market-predictions-tensorflow-9e83999d4653

https://www.thepythoncode.com/article/stock-price-prediction-in-python-using-tensorflow-2-and-keras

https://analyticsindiamag.com/hands-on-guide-to-lstm-recurrent-neural-network-for-stock-market-prediction/

https://blog.usejournal.com/stock-market-prediction-by-recurrent-neural-network-on-lstm-model-56de700bff68

https://github.com/jaungiers/LSTM-Neural-Network-for-Time-Series-Prediction

http://colah.github.io/posts/2015-08-Understanding-LSTMs/