Cover Page

Course:	EE590 Project Report
Name:	Nikhilesh Sankara Narayanan
Student ID:	3060376551
Term:	Spring 2020
Project Title:	Comparison of Stock Price Prediction Techniques
Date:	5/1/2020

Abstract

One of the most challenging problems in the Stock Market is predicting how stocks will perform in the future. There are many different factors that affect share prices, and so predicting their movement with high accuracy is difficult. In addition to the large number of unknown factors, a lot of noise is inherent in the data as well. Therefore, different samples of data from various timeframes need to be analyzed to conclusively evaluate the effectiveness of any particular quantitative methodology. This paper will be looking at a few different statistical and learning methods applied to this problem and attempt to benchmark their performance against real-world data. Each of the techniques described was implemented in Python using statistical library tools. The best technique was measured using mean-squared error against observed data, and it was identified to be the most complex model implemented using a neural network.

Introduction

This paper analyzes and compares different quantitative techniques that can be used in stock price prediction. They are compared based on their error rate when compared to real data, and after comparing them across multiple sets of stock price data a conclusive judgement of the effectiveness of each technique can be formed. Since stock prices fluctuate daily and are subject to many different factors (many of which are unknown/not measurable) we only consider the past history of a stock's price and try to use that as a guide to its future values. The hope is that there is an identifiable pattern/set of patterns within the price movement that can be determined, to aid us in properly predicting future prices. Finally, once all predictions are available, they can be compared by their error rates (when predicting) and profitability when used to build a portfolio.

Being able to produce accurate results is extremely important to investors to be able to better improve the return of their investment portfolios and to safeguard themselves against risk due to price volatility. Understanding the difference between different quantitative analysis techniques is also important to gauge in which situations/market conditions a particular model is most effective. It is possible (and often the case) that a single technique cannot accurately capture the bigger market picture, and so relying on a combination of techniques is most advisable.

The four different techniques that were implemented, tested and compared are:

1. <u>Linear Regression:</u>

This is the simplest learning method. We will use the dates in the time-series as the independent variable and try to fit a line to the predict stock prices in future dates.

2. <u>k-Nearest Neighbors Regression (kNN):</u>

This is a simple Machine Learning algorithm that given a collection of data points, assigns a group that each data point belongs to based on the training data. Once trained, it can be used to predict future, unseen data. Again, this can be back-tested to ensure that the classification has low error.

3. <u>Auto-Regressive Integrated Moving Average (ARIMA):</u>
We can, with some adjustments to data (to ensure stationarity, etc.) model the stock prices as an ARIMA model, which is a type of model where a given time series is based on it's own past values (with a "lag") and is used to forecast future values. We will have to find the characteristics terms of the model (p, q, d values) to fully specify it [6]. This can be done using an exhaustive Grid Search.

4. <u>Neural Network - Multi-Layer Perceptron (MLP):</u>
Since Neural Networks can be thought of as an approximation for complex mathematical functions/equations, just with much higher number of weights and complexity, they can similarly be used to model stock prices [5]. Mostly, Recurrent Neural Networks are more appropriate for stock price prediction, such as their use in the Elman & Jordan Networks [2]. Since there is a very large amount of financial data to learn from, the size and complexity of modern Neural Networks make them a good model to accurately capture the "big picture" i.e. get more accurate predictions. In this project, we implement a Multi-Layer Perceptron (MLP), a type of simple Artificial Neural Network.

The results are mostly open to interpretation; however a clear trend seems to emerge. Very simple model (Linear Regression) performs reasonably well in most situations, as it is fitting a straight line to the data. Therefore, stocks that generally rise/fall steadily over time are easily modeled by it. However, more drastic changes are not properly captured.

The statistical technique, ARIMA performs better on average than Linear Regression, and in some cases outperforms the Machine Learning models (k Nearest Neighbors Regression and Neural Network). This captures patterns in the data more accurately even when there are non-linear fluctuations.

The Machine Learning models both perform quite well with a low rate of error, as the training data is quite easy to learn from and fit a model too. However, over very long timeframes the performance of the models starts to break down. In addition, these models are specifically trained per stock and so are not applicable to different stocks and timeframes without major refitting and training – in contrast, a Linear Regression or ARIMA model can be quickly refitted to a new dataset as the parameters are far fewer. In addition, a key point to note is that models like ARIMA are expected to do better in the short term, whereas Neural Networks can possibly capture long-term trends more accurately.

The performance of each model is described further in the results section. Each model was tested on the same stock data across 10 companies, and over a period of 1 year, 5 years and 10 years of daily data.

The conclusion is that which model to use really depends on the timeframes, and the specific stocks that are being analyzed. Either ARIMA model or Neural Network model perform reasonably well in most situation and can be a good guideline to base investment decisions off of. Models like Linear Regression are too simple to capture major fluctuations in price and so are more helpful for a very long-term perspective of steadily growing stocks (e.g. Coco-Cola). On the other hand, slow training models like neutral network model can provide some additional insights, but overall do not provide much performance improvement despite added complexity and difficulty to build, train, tune parameters and test.

PROBLEM STATEMENT

Given any time-series data over a specified timeframe, there are many different ways to analyze that data and possibly forecast future data based on it. In this paper, we consider 4 specific techniques, increasing in complexity from a simple linear system to a multi-layer neural network. The data we are analyzing is a set of companies' stock prices over multiple timeframes, and the goal is to

1. Extract stock price information for a chosen set of companies and prepare it for training.
2. Implement the four quantitative models (as described above).
3. Train each predictive model on existing training data with best parameters.
4. Predict future prices for each model, and compare to known observations (that are not known to the model).
5. Measure performance of each model and compare them against each other, to identify when and which model is more effective.

This paper is mainly trying to achieve point 5 – to understand which model is more consistently accurate over multiple datasets. That model inspires greater confidence when investing with real money, as the potential or profit or loss is rather great.

Such a problem makes sense from a theoretical perspective, but there are a number of real-world factors to consider when running such models. The major issues with the problem we are trying to solve are as follows:

1. The fluctuations in stock prices are not known to be anything but a random walk. Therefore, while we may attempt to model its movement and predict future values, there is a fundamental assumption that some learnable patterns do exist in the time-series data.
2. Complex models like Deep Learning models can be time consuming due to their complexity, but not perform much better than more straightforward models like ARIMA/Linear Regression. There is significant additional time and effort to build complex models, process data, tune hyperparameters and train them on the given data. However, this does not always yield much benefit over more conventional methods.
3. Quantitative Technical Analysis cannot properly account for catastrophic events such as an Economic Recession, or a global pandemic that would affect severely stock market prices. They are not capable of incorporating such events as they are mostly random, and so the timeframes chosen do not have any major recessions (in USA) occurring during them.
4. It is not entirely know upon which factors stock prices depend, or in fact whether different data points in the same timeseries depend on the same factors. For example, changes in laws and regulations may indirectly affect the distribution of stock prices, such that even within a particular dataset, there is not one single distribution that can describe it completely.

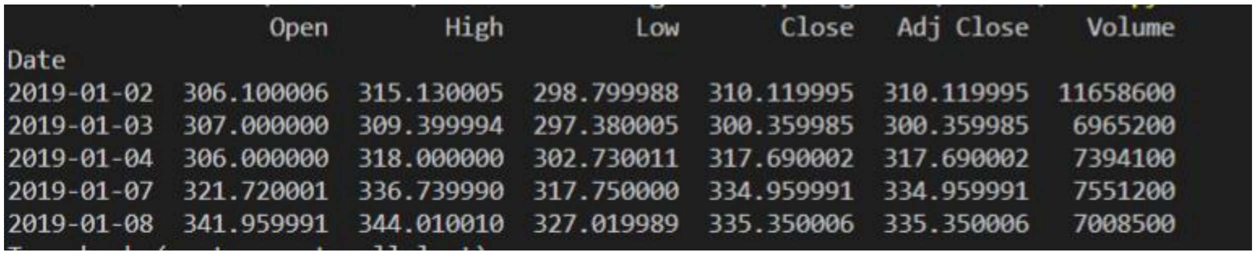
DATA COLLECTION

The data required is time-series data over different time periods, and for a set of companies. This data is easily found on Yahoo! Finance and similar financial information websites [1]. Each of the companies' Daily stock price data is downloaded for timeframes: 1 year, 5 years and 10 years using Python code, as this makes it much easier to quickly specify Ticker Symbols and Timeframes without having to manually download each file. Yahoo! Finance was chosen as it is a reliable website that's commonly used for similar projects, and it has a public API that allows us to download data files directly into our program through the use of external libraries.

The library used in this project is the **yfinance** library [7], available in the **pip** installer for Python. It can be installed from the Python terminal using the command

```
$ pip install yfinance --upgrade --no-cache-dir
```

The benefit of this package is that it avoids having to manually visit the Yahoo! Finance website, enter the stock ticker symbol, manually set start and end dates and download and Excel file. It allows direct download of data into a Pandas DataFrame object which can then be further processed to extract the necessary info. The raw downloaded data appears in the following format:



```
                  Open        High         Low       Close   Adj Close    Volume
Date
2019-01-02  306.100006  315.130005  298.799988  310.119995  310.119995  11658600
2019-01-03  307.000000  309.399994  297.380005  300.359985  300.359985   6965200
2019-01-04  306.000000  318.000000  302.730011  317.690002  317.690002   7394100
2019-01-07  321.720001  336.739990  317.750000  334.959991  334.959991   7551200
2019-01-08  341.959991  344.010010  327.019989  335.350006  335.350006   7008500
```

Figure 1: Expected format of raw stock price data downloaded using the yfinance API

The data we require for this project is simply the Daily Close Price, as well as the Data column. The data does have to be reorganized a little bit, as the Date values are presented as the Index and not as a separate column – we can therefore create a new column labelled "Date" and set it to the index. This creates the Date column which will be used as the Independent ('x') variable for each of the models. Since the data is already formatted as a DataFrame object, extracting the "Date" and "Close" columns becomes trivial. The column labels are then dropped, and then the DataFrames are reformatted into NumPy arrays, and reshaped into (-1,1) 1D arrays with one feature.

The data is then split into Training and Testing subsets, which the Test data simply being the last "X" datapoints. i.e. The models are trained on the first (totalObservations – X) data points, and then used to predict the next 'X' datapoints, finally comparing them to the observed values. This X is a hyperparameter that is set relatively low e.g. 7 days of predicting/testing data, for years of training data. The final data is simply 4 columns: Train and Test arrays for 'x' values (independent variable) and 'y' values (dependent variable).

One additional pre-processing step is required for the Linear Regression model, as it only works for numerical data as the independent variable. Therefore, the Date is converted into a numeric representation. For this we can use the Ordinal Representation of the date, using the toordinal() function in the datetime library. From the Python documentation: "*Returns the proleptic Gregorian ordinal of the* *date, where January 1 of year 1 has ordinal 1".* That is, is starts counting days from the date 01/01/01 and assigns value 1 to it, increasing by 1 for each subsequent day.

For the Machine Learning models (K Nearest Neighbors Regressor and Multi-layer Perceptron) the data is further scaled into the (0,1) interval using a Min-Max scaler, such that each datapoint is subtracted by the minimum and divided by the number of data points. This scaler was chosen because the stock price data tends to not have many outliers i.e. sudden large spikes in price between successive days. Daily prices are on a relatively similar scale with only minor fluctuations (this is not valid for weekly/monthly data). The benefit of such scaling is that it is easier for the Machine Learning algorithms to identify features on a smaller scale, while still maintaining the relative proportions of the dataset. They can also converge faster. [4]

The final, data-related constraint is that since the data is widely varied and fluctuates a lot, ensuring stationarity of data for the ARIMA model – and futher, stationarity of AR,and invertibility of MA coefficients - is impossible to guarantee. For this reason, a Grid Search algorithm is applied to the ARIMA model wherein it is tested for different values of (p,d,q) coefficients, and the one with the least error is chosen as the final model [13]. This gives it a slight performance edge over the other techniques, but it is necessary, as one consistent set of parameters cannot be used for all the different datasets. For example a (1,0,1) model might work well for one dataset, but not result in stationary coefficients for a different dataset – therefore each dataset needs to be tested against different possible configurations to ensure that it actually works. Further, using minimized error as selection criteria further improves the final objective of obtaining accurate predictions (although making ARIMA models comparatively better performing than alternatives).

To balance out this additional bias, grid search is applied to the hyperparameters of the MLP Regressor model as well. This is described further in later sections.

Finally, the data produced from each of the models is outputted to ".csv" (comma-separated values) files depending on which company's stock and timeframe was chosen. It is then combined with the training data to produce contiguous charts. This is further described in the Results section.

METHODS USED

The main libraries and packages used in this project are

- Pandas, for data input and processing [9]
- NumPy, for array manipulation/reshaping [10]
- yfinance, for downloading stock price data [7]
- Scikit-Learn and statsmodels, for implementing and training the models [11]

The first step is to download and preprocess the data, which is described in the section above. Once the data is processed and split into testing and training sets, each of the models is run to train on the training set, and calculate error based on the difference between the predicted output and actual observed test data. All code, along with necessary packages to be installed are attached with the report. Each of the 4 models' implementations is described in further detail:

1. Linear Regression Model

This is the simplest model, that accepts train and test datasets, with independent variable being converted from a DateTime value to a numeric value for use in the regression. It simply fits a line to the training data, i.e. identifies a slope and an intercept that produces a “best-fit” approximation of the data [3] . The result is a straight line that is the least distance from all training points, resulting in lowest possible error. The error calculation used is the mean squared error (of which we take the square root to get measure absolute distance of test data points from prediction line). The key issue with this model is that it will always output a straight line and so all predictions will be linearly increasing or decreasing. Therefore it is nearly useless for longer, fluctuating datasets that are not even close to linear.

Despite its poor performance, it still outputs reasonable values in the approximate range of real values, and so is useful as a baselines “predictor” to compare more advanced models against.

2. ARIMA Model

The ARIMA model ('Auto Regressive Integrated Moving Average') is useful for predicting time-series data that are dependent on their own past values. This can be analyzed to help predict future values in the time-series. A key requirement of ARIMA models is that the input data be stationary; if the data is non-stationary, a differencing step is usually applied where previous values in the series are subtracted from subsequent values, in order to eliminate the non-stationarity.

It relies on 3 key parameters (p, d, q) where

- p is the “order” (how much the model lags)
- d is the degree of differencing
- q is the moving-average order.

In this project, the implementation of ARIMA uses the statsmodel package [14] that provides a parameterized implementation of this method, where the input dataset and the values of (p, d, q) are provided as arguments to the ARIMA function. Since it is not possible to identify whether a dataset is stationary or not without prior testing, and furthermore difficult to difference properly until it is stationary, we use a Grid Search algorithm to identify the best values for each of the (p, d, q) parameters. Specifically, different combinations of values for (p, d, q) are tested, with ARIMA models being generated for each combination, and then forecasts produced and their error calculated against the historical test data. For each dataset, the ARIMA configuration with the lowest mean-squared error is chosen. The benefit of selecting the best performing model while training the data is that it helps achieve better stock price predictions on average , which is one of the goal this project is trying to solve. Therefore, even though this puts the ARIMA models at an advantage when comparing with the other three, it is still a worthwhile endeavor. In addition, configurations that lead to incompatible values for p, d, q are discarded (such as non-stationary ‘p’, or non-invertible ‘q’).

Furthermore, ARIMA models are generally only used to produce very short out-of-sample forecasts. The key reason is that since the entire model relies on using historical data to predict future data, the further into the future you try to predict without using observed data, the less accurate the prediction becomes. That is, while time-series data up to today can be used to predict tomorrow’s datapoint quite accurately, prediction of data into the next week is less accurate because there is no data of the one week in between.

It is also not useful to use predicted data, as this is not known to be accurate yet and can lead to the model overfitting itself.

Therefore, this project implements a Rolling Forecast ARIMA model [12] , where after each prediction is made, the actual observed data (of the date that was just forecast) is added to the training data and the model is re-created. In this way, after each data point is predicted the model is retrained until the end of the test data is reached. The following chart describes the Rolling Forecast technique:

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8
Regular								
	Training Data					Predictions		
Rolling								
	Training Data					Prediction		
	Training Data					Test Data	Prediction	
	Training Data					Test Data	Test Data	Prediction

Figure 2: Comparison of a regular Out-of-Sample Forecast and Rolling Forecast

Therefore, in a Rolling Forecast, the final prediction incorporates values from the test data set into the training data set, i.e. the model continues improving even when predicting. This helps forecasts that are more than a few time steps out-of-sample.

3. K Nearest Neighbors Regressor

The K Nearest Neighbors (kNN) algorithm is useful is finding the similarity between old and new data points, and uses that to predict "new" data points given "old" (training) ones. This was implemented in Python using the scikit-learn's Neighbors module [11]. The only parameter that needs to be optimized is the value of 'k' i.e. how many neighbors are considered for a single data point. This is also a Regression model similar to Linear Regression, however it has a greater capability to learn patterns from past data and can be used to forecast a bit further into the future with greater accuracy.

The data used for this regressor is first scaled using a Min-Max scaling algorithm, that maps all values to the range (0,1). This leads to faster convergence and hence less training time. [4]

Similar to parameter optimization in the case of ARIMA forecast, we can experiment with different values of 'k' parameter for each dataset. The benefit of this is that while the underlying algorithm is the same, different time-series datasets will have unique patterns and seasonality, and so optimizing parameters to each dataset such that we get the best performance, will help achieve the 'best performance' goal. The kNN model is re-run over the entire training set for different values of 'k', and the one with the lowest error over test data set is chosen as the final model for comparison with other models. Note that all models are actually predicting over the test set – we simply chose the one with a slight performance edge to account the final results for, to simply reporting and to have a robust criteria for parameter value selection.

4. Multi-layer Perceptron

The final model is a simple neural network, implemented as an MLP (multi-layer perceptron). The data used for this model is also scaled to the (0,1) interval using a Min-Max scaler. The MLP architecture is straightforward – it has an input layer, one hidden layer (with variable size) and an activation (output) layer. However, the key point here is that the size of the hidden layer and the activation function are learnt parameters as well, using the Grid Search Algorithm from before. The following image was created by me to illustrate the structure of the MLP used.

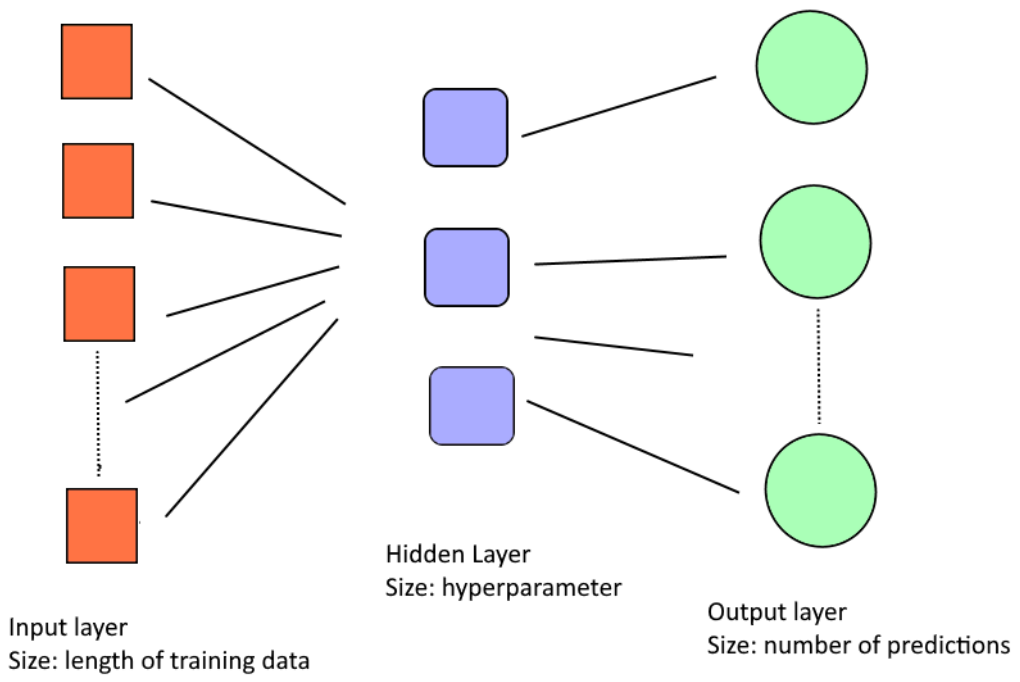


Figure 3: Internal structure of multi-layer perceptron

The hyperparameters chosen were:

- Maximum number of iterations – usually set to 5,000 or 10,000
- Hidden layer size – either 5 or 10 nodes. Significantly increasing this leads to far longer training time.
- Activation function – chosen from the following, depending on best fir to the data. If g() represents the activation function and z the input to that layer, then:
    - Identity function: $g(z) = z$
    - Logistic Function: $g(z) = \frac{1}{1+e^{-z}}$

- tanh function: $g(z) = \frac{(e^{-z} - e^{z})}{(e^{-z} + e^{z})}$
- reLU (rectified Linear Unit): $g(z) = \max\{0, z\}$

- Weight optimization solver function – this is the function that recalculates weights after each prediction during training process. The functions tested are the Stochastic Gradient Descent (which is the defacto standard), the "Adam Optimizer" [15] and the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm [16]. All three are popular optimization functions for neural networks and so we let their performance pick the best one.
- Regularization parameter – this model uses L2 regularization, which adds a penalty to the loss function, to disincentive the model when it makes an error. The parameter values are usually extremely small, and are chosen between 0.0005 and 0.005 for this model.

Generating results: Each of the models produces the same output – a list of predicted values for stock prices, as well as the total error between the predictions and the observed data. The error is calculated as the square root of the mean squared error between the list of predictions and the list of test data points. This list of predictions is then charted along with the original training data and test data, such that the deviation of predictions from actual data is clearly visible.

RESULTS

The format of these results needs to be explained first. First, for a given company's stock data, each model is trained on 1 year, 5 years, and 10 years of data, after which they produce a forecast of the next 1 week's and 1 month's stock prices. The reason for choosing such a small output timeframe compared to the training data is that forecasting too far into the future will be less accurate than forecasting immediately after the end of training data. The results are charted, so that the difference between the actual observed data and the predicted data is clearly visible. Finally, the error rate of each model is also charted to observe which model performed best for that company's stock, and for which timeframes.

For the purpose of charts, I will display the results of 3 company's stock predictions - that alone results in 4 (models) x 3 (timeframes) x 2 (companies) = 24 different plots to be made. To simplify the graph, I will composite the results of the 4 models for a given timeframe onto one graph so their relative performance can be seen clearly. The error reporting is much more simple and can be performed for a greater number of companies. The code used to generate the numeric output is available and can be run to verify the authenticity of these results.

Company: Tesla (TSLA)

The following chart is simply the observed stock price of Tesla over the past 1 year, taken from Yahoo! Finance [1]. This is just to illustrate the general trends in its price as the entire history will not be visible when looking at the finer details later on:

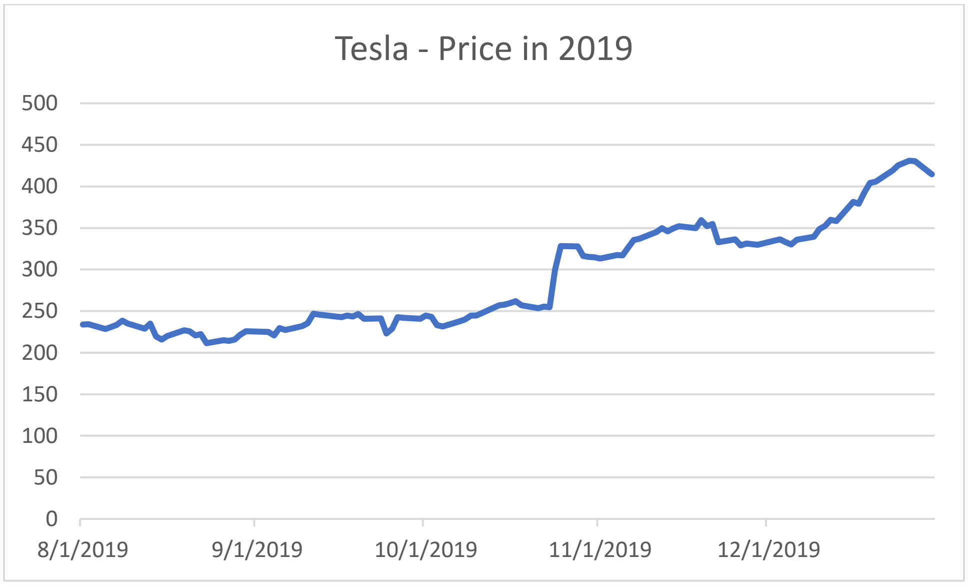


Figure 4: Stock price data for Tesla from August-December 2019

Now we can look into the last month of data for a much closer look at how each model performed:

Using 1 year of data:

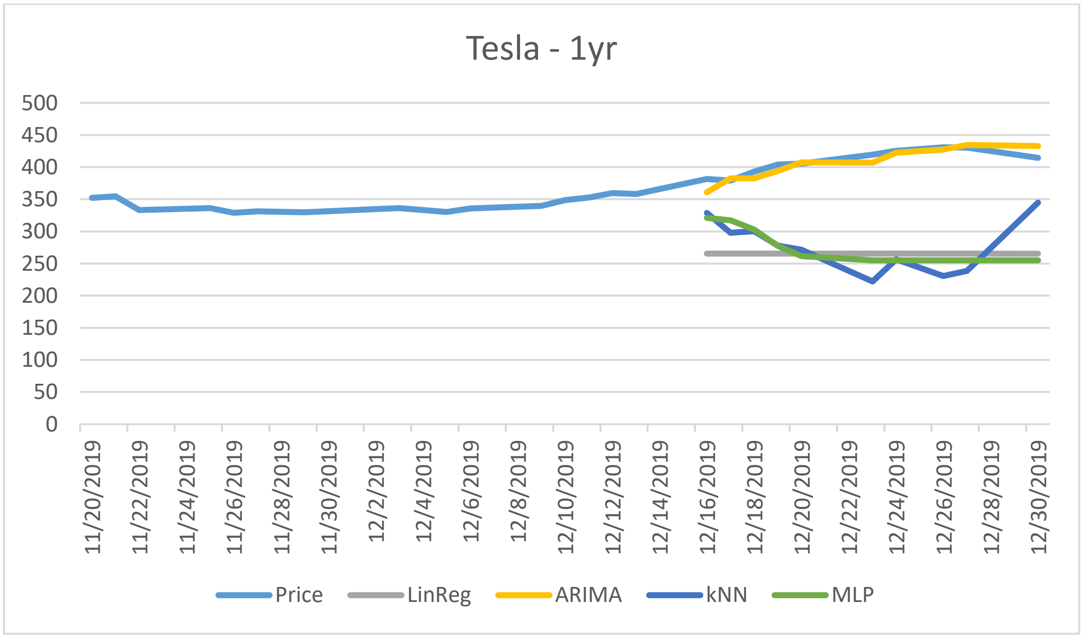


Figure 5: Stock price predictions for TSLA between 16-30 December 2019 with 1 year of training data

At first glance, it is abundantly clear that Linear Regression cannot solve this problem, as it just produces a straight line through the majority of data. The ARIMA model is an excellent fit, however. It captures the overall rising trend perfectly. The two Machine Learning models are more reasonable than the Linear Regression, but they both predicted s sharp downtrend which is not observed. Therefore, so far the ARIMA model is outperforming. The absolute error of each model over all predictions is:

Linear Regression Error	ARIMA Error	kNN Error	MLP Error
144.1212	10.81961	141.8296	139.8405

Using 5 years of data:

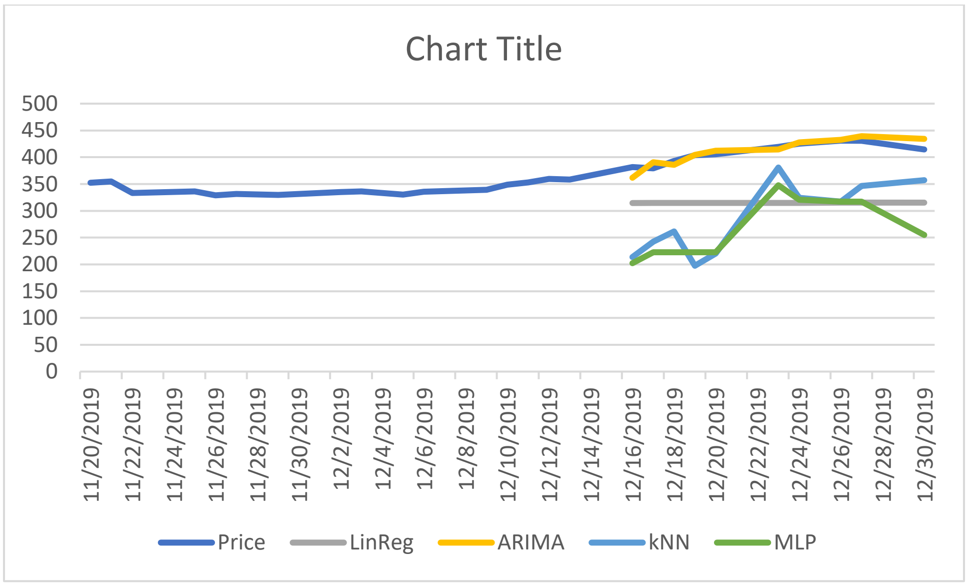


Figure 6: Stock price predictions for TSLA between 16-30 December 2019 with 5 years of training data

Error:

Linear Regression Error	ARIMA Error	kNN Error	MLP Error
95.2346	10.54167	132.6754	147.942

<u>Using 10 years of data:</u>

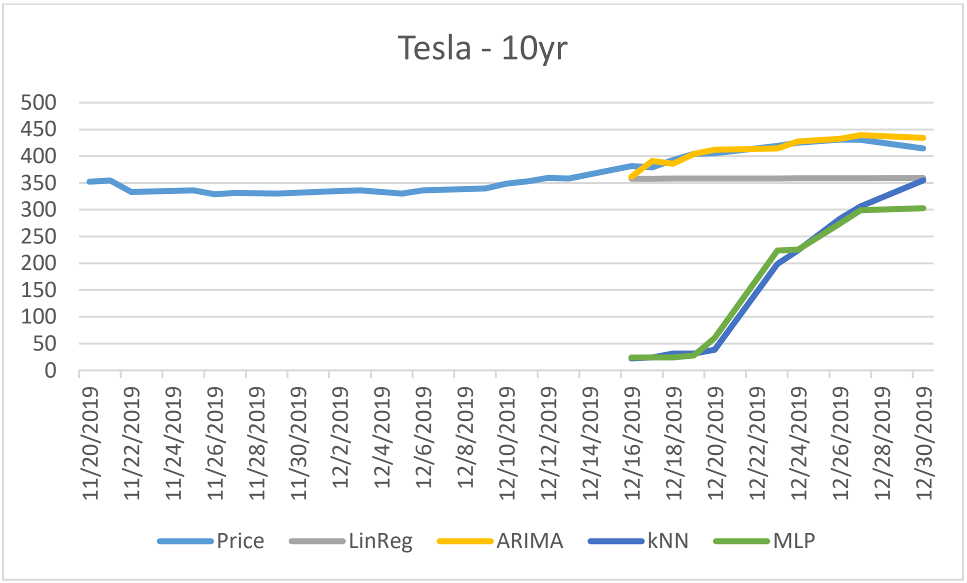


Figure 7: Stock price predictions for TSLA between 16-30 December 2019 with 10 years of training data

Linear Regression Error	ARIMA Error	kNN Error	MLP Error
53.00548	10.56595	280.9965	279.8241

Looking at both the charts and the error values, it’s clear that the ARIMA model outperforms all others significantly. This is in part due to the rolling forecast approach used for it (which is not appropriate for the other models given their design). Therefore we can conclude that for a very short out-of-sample forecast, the ARIMA model performs well relative to the other models considered, and very well in absolute sense.

One key point to note about the Machine Learning models is that they break down when training data gets longer and longer. This is contrary to my initial expectations of them performing better in the long run. I hypothesize that especially 10 years ago, stock prices were much lower for TSLA than they are today, and that data is weighted too heavily into future predictions. Both the kNN and MLP model suffer from this and are almost completely inaccurate over the 10-year timeframe (the error in prediction is nearly equal to the stock price itself).

Company: Microsoft (MSFT)

The following chart is simply the observed stock price of Microsoft over the past 1 year from Yahoo! Finance [1], to gain an overall picture of its stock price movements:

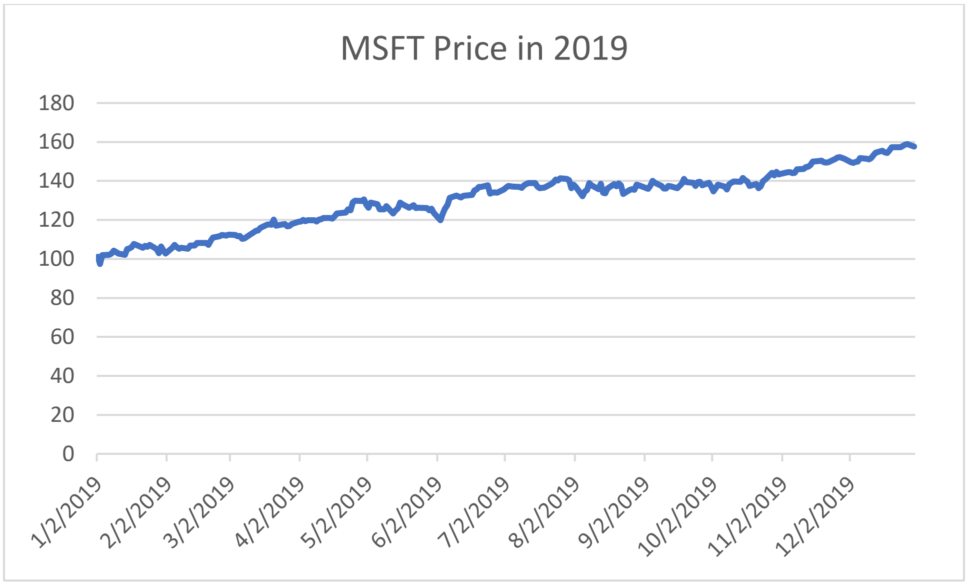


Figure 8: Stock price data for Tesla for the entire year 2019

Using 1 year of data:

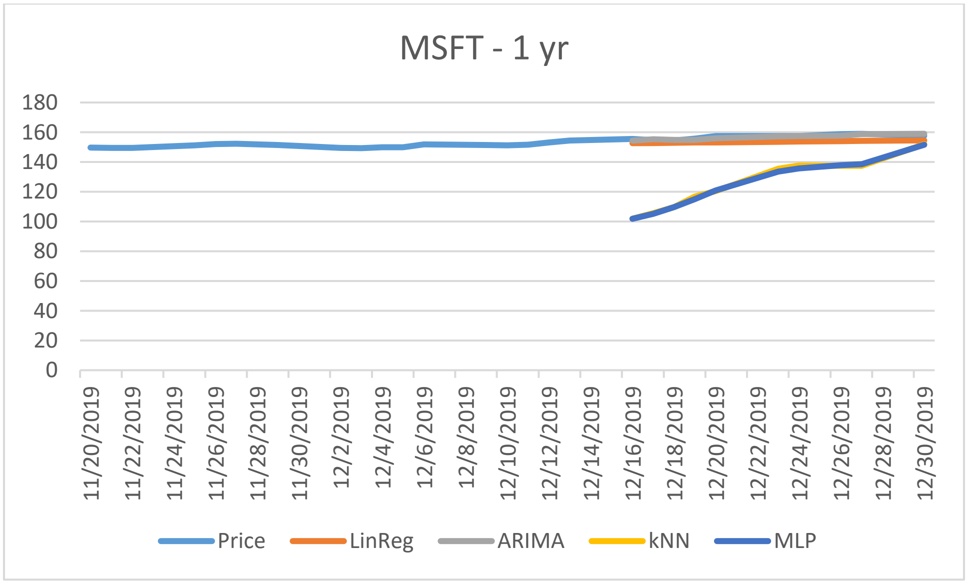


Figure 8: Stock price predictions for MSFT between 16-30 December 2019 with 1 year of training data

Errors:

Linear Regression Error	ARIMA Error	kNN Error	MLP Error
3.467636	0.919477	34.61395	35.00484

<u>Using 5 years of data:</u>

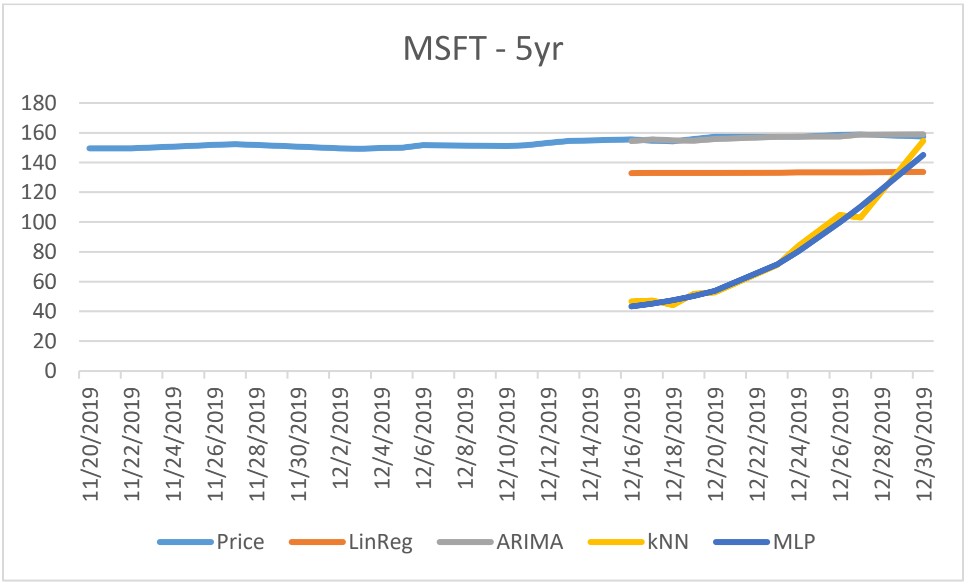


Figure 9: Stock price predictions for MSFT between 16-30 December 2019 with 5 years of training data

Errors:

Linear Regression Error	ARIMA Error	kNN Error	MLP Error
23.58869	0.944139	87.21067	87.78028

<u>Using 10 years of data:</u>

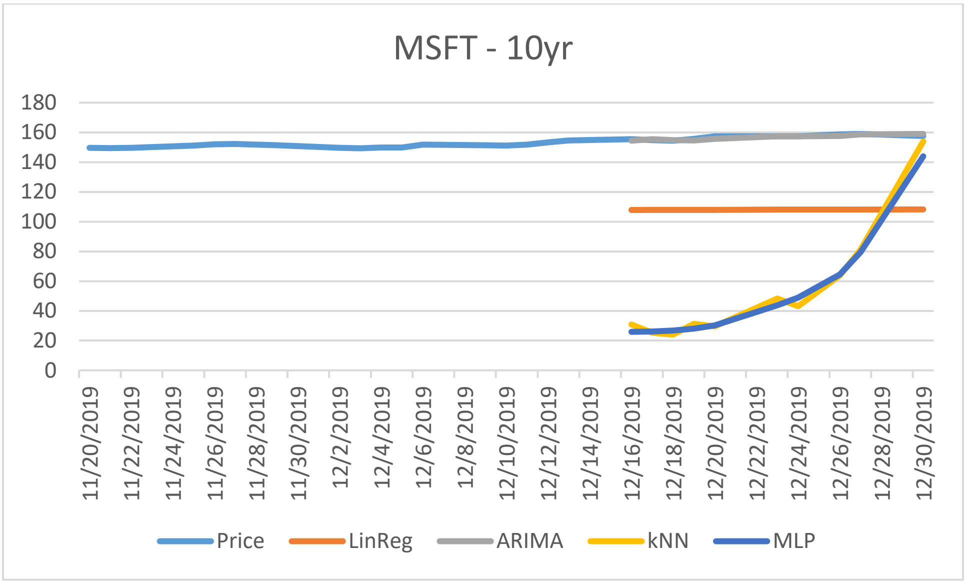


Figure 10: Stock price predictions for MSFT between 16-30 December 2019 with 10 years of training data

Errors:

Linear Regression Error	ARIMA Error	kNN Error	MLP Error
48.8016	0.958547	110.0443	110.5016

As can be clearly seen from these graphs, our models are all within a reasonable scale of stock prices, and in some cases are not too far from observed data. This lends some confidence to their efficacy. However, as in the case of TSLA, the Machine Learning models breakdown at longer timeframes, while the ARIMA model is consistently well performing with low error and good fit.

Therefore, it can be concluded that this project's goals have been achieved. Though the performance of some models was not upto the standard expected at the onset, the results are consistent in declaring the ARIMA model most reliable for stock price prediction of the given companies.

The results are indeed positive – however, not completely as expected. The fact that the data shows that the Machine Learning models are so poor at long timeframes is surprising. It is possible that there are other issues with the model, such a simplicity and unoptimized parameters, but the surprising fact is the variation in performance between short and long timeframes. This is definitely an avenue for future research and improvement of the Machine Learning models.

CONCLUSIONS

The objective of this project was to implement and compare four different quantitative techniques and measure them based on their performance individually, and relative to each other. It also measures which model produces the best results when attempting to predict future stock prices in short-term and long-term timeframes.

On the basis of these results, we can conclude that an ARIMA model works well at all timeframes of training data. Models like Linear Regression are nearly useless in practically predicting real world data, and more complex models like kNN Regression and Multi-Layer Perceptron tend to perform reasonably well across shorter time-frames, but breakdown at longer ones. The program setup allows these tests to be conducted on multiple different company stocks and timeframes with ease, but the trend we see from the companies analyzed is most likely to hold.

The results of this project are significant because it describes the structure of a general purpose Predicting and Benchmarking program, that can be easily extended with further quantitative models. As long as their input and output format are maintained, any number of forecasting techniques can be easily implemented and the program will incorporate them into its tests. Secondly, it gives a rather good picture of the drawbacks of each of the models analyzed, and can be useful in understanding when NOT to use a particular technique. All in all, this project should serve as a good starting point to further study and analyses of the various techniques one can use in stock price prediction.

References

1) Publicly traded stock data, *Yahoo! Finance*. Retrieved from http://finance.yahoo.com

2) "Stock Market Prediction." *Wikipedia*, Wikimedia Foundation, 30 Aug. 2019, en.wikipedia.org/wiki/Stock_market_prediction.

3) Ng, Yibin. "Machine Learning Techniques Applied to Stock Price Prediction." *Medium*, Towards Data Science, 3 Oct. 2019, https://towardsdatascience.com/machine-learning-techniques-applied-to-stock-price-prediction-6c1994da8001

4) Singh, Aishwarya. "Predicting the Stock Market Using Machine Learning and Deep Learning." *Analytics Vidhya*, 4 Sept. 2019, www.analyticsvidhya.com/blog/2018/10/predicting-stock-price-machine-learningnd-deep-learning-techniques-python/.

5) Thawornwong, S, Enke, D. Forecasting Stock Returns with Artificial Neural Networks, Chap. 3. In: Neural Networks in Business Forecasting, Editor: Zhang, G.P. IRM Press, 2004.

6) Brownlee, Jason. "How to Create an ARIMA Model for Time Series Forecasting in Python." *Machine Learning Mastery*, 17 Sept. 2019, https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/

7) Yfinance Python Library, *PyPI*, www.pypi.org/project/yfinance.

8) Official Python Documentation, https://docs.python.org/3/library/index.html

9) Official Pandas Documentation, https://pandas.pydata.org/

10) Official NumPy Documentation, https://numpy.org/

11) Scikit–Learn package Documentation, https://scikit-learn.org/stable/user_guide.html

12) "Variations on Rolling Forecasts." *Portrait of the Author*, 15 July 2014, robjhyndman.com/hyndsight/rolling-forecasts/.

13) "How to Grid Search ARIMA Model Hyperparameters", Jason Brownlee, January 18, 2017, https://machinelearningmastery.com/grid-search-arima-hyperparameters-with-python/

14) Statsmodels v0.11.1 Documentation, https://www.statsmodels.org/stable/user-guide.html

15) Kingma, et al. "Adam: A Method for Stochastic Optimization." *ArXiv.org*, 30 Jan. 2017, https://arxiv.org/abs/1412.6980

16) "Iterative Methods for Optimization." *Frontiers in Applied Mathematics*, https://epubs.siam.org/doi/abs/10.1137/1.9781611970920.ch4

Appendix

Software Used: Python (version 3.0 and above)

Packages and imports required:

Pandas:	https://pandas.pydata.org/
NumPy:	https://numpy.org/
Scikit-Learn:	https://scikit-learn.org/stable/
statsmodels:	pip install statsmodels
yfinance:	pip install yfinance

All the code has been compiled into one file, with separate functions for data processing and running each model.

To change the Company Stock that is analyzed and the timeframe, edit the calls to “runModels” in main() function.

```
import pandas as pd
import math
from functools import reduce
# from pandas import datetime
import numpy as np
import yfinance as yf
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from sklearn import neighbors
from sklearn import neural_network
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARIMA

#Note - need to run python -
m pip install statsmodels  and pip install yfinance

def downloadData(ticker, start, end):
    # df = yf.download('TSLA', start='2019-01-01', end='2019-12-
31', progress=False)
    df = yf.download(ticker, start, end)
    df["Date"] = df.index
    return df

def processInput(input):

    numRows = input.shape[0]     #not counting labels
    numCols = input.shape[1] - 2 #not counting Number and Date

    TEST_OFFSET = 10
    x = input["Date"]
    y = input["Close"]
    x_linreg = x.map(pd.datetime.toordinal)

    x_train = np.asarray(x[0:numRows - TEST_OFFSET ]).reshape(-1,1)
    y_train = np.asarray(y[0:numRows - TEST_OFFSET]).reshape(-1,1)
    x_test = np.asarray(x[numRows - TEST_OFFSET:numRows+1]).reshape(-
1,1)
    y_test = np.asarray(y[numRows - TEST_OFFSET:numRows+1]).reshape(-
1,1)

    x_train_linreg = np.asarray(x_linreg[0:numRows - TEST_OFFSET ]).re
shape(-1,1)
    x_test_linreg = np.asarray(x_linreg[numRows - TEST_OFFSET:numRows+
1]).reshape(-1,1)

    return x_train, y_train, x_test, y_test, x_train_linreg,  x_test_l
inreg


def linreg(x_train, y_train, x_test, y_test):
    model = LinearRegression().fit(x_train, y_train)
    # print(model.coef_, model.intercept_)

    y_pred = model.predict(x_test)
    error = math.sqrt(mean_squared_error(y_test, y_pred))
    # print('Mean squared error: %.2f' % error)
    return y_pred, error

def arima(x_train, y_train, x_test, y_test, arimaOrder = (1,1,1)):
    historical = [i for i in y_train]
    errors = list()
    #predict future len(y_test) observations
    predictions = list()
    for t in range(len(y_test)):
        model = ARIMA(historical, order=arimaOrder)
        model_fit = model.fit(disp=0)
        output = model_fit.forecast()
        y_pred = output[0]
        predictions.append(y_pred)
        obs = y_test[t]
        historical.append(obs)
        # print('predicted=%f, expected=%f' % (y_pred, obs))
        error = math.sqrt(mean_squared_error(y_test[t], y_pred))
        errors.append(error)
    #get final error
    finalError = math.sqrt(mean_squared_error(y_test, predictions))
    return predictions, finalError

def arimaGridSearch(x_train, y_train, x_test, y_test, ps, ds, qs):
    minError, bestOrder = float("inf"), None
    bestPreds = None
    for p in ps:
        for d in ds:
            for q in qs:
                arimaOrder = (p,d,q)
                try:    #prevent ValueError if coefficients not approp
riate
                    preds, error = arima(x_train, y_train, x_test, y_t
est, arimaOrder)
                    if minError > error:
                        minError = error
                        bestOrder = arimaOrder
                    bestPreds = [pred for pred in preds]
                except:
                    continue    #skip if coeffs give error
    return bestPreds, minError, bestOrder

def knn(x_train, y_train, x_test, y_test, k):
    #re-scale features to 0->1 range
    mmscaler = MinMaxScaler(feature_range=(0,1))
    x_train = mmscaler.fit_transform(x_train)
    x_test = mmscaler.fit_transform(x_test)

    #build model
    model = neighbors.KNeighborsRegressor(n_neighbors = k)
    model_fit = model.fit(x_train, y_train)  #fit the model
    y_pred = model.predict(x_test)
    #rmse error
    error = math.sqrt(mean_squared_error(y_test, y_pred))
    return y_pred, error

def knnOptim(x_train, y_train, x_test, y_test, maxK):
    minError = float("inf")
    bestPreds = None
    bestK = 1
    for k in range(1, maxK+1):
        preds, error = knn(x_train, y_train, x_test, y_test, k)
        # print(preds, error)
        if minError > error:
            minError = error
            bestPreds = [pred for pred in preds]
            bestK = k
    return bestPreds, minError, bestK

def mlp(x_train, y_train, x_test, y_test):
    #re-scale features to 0->1 range
    mmscaler = MinMaxScaler(feature_range=(0,1))
    x_train = mmscaler.fit_transform(x_train)
    x_test = mmscaler.fit_transform(x_test)

    #train model
    model = neural_network.MLPRegressor(max_iter=10000)

    #optimize params with grid search
    params = {
        "hidden_layer_sizes": [5,10],
        "activation": ["identity", "logistic", "tanh", "relu"],
        "solver": ["lbfgs", "sgd", "adam"],
        "alpha": [0.0005,0.005]
        }
    gsModel = GridSearchCV(estimator=model, param_grid=params)

    #fit the model
    # model_fit = model.fit(x_train, np.ravel(y_train))
    gsModel.fit(x_train, np.ravel(y_train))
    y_pred = gsModel.predict(x_test)
    # print("Params chosen: ", model.get_params())
    error = math.sqrt(mean_squared_error(y_test, y_pred))
    return y_pred, error
```

```
def runModels(tckr, start, end, yr):


    #1 year
    input = downloadData(ticker=tckr, start=start, end=end)
    x_train, y_train, x_test, y_test, x_train_linreg, x_test_linreg = 
processInput(input)

    #LinReg
    lrPreds, lrError = linreg(x_train_linreg, y_train, x_test_linreg, 
y_test)
    lrPreds = reduce(np.append, lrPreds)
    #ARIMA
    arimaPreds, arimaError, bestOrder = arimaGridSearch(x_train_linreg
, y_train, x_test, y_test, range(0, 3), range(0, 3), range(0, 3))
    arimaPreds = reduce(np.append, arimaPreds)
    # kNN
    knnPreds, knnError, knnK = knnOptim(x_train_linreg, y_train, x_tes
t, y_test, maxK=10)
    knnPreds = reduce(np.append, knnPreds)
    #mlp
    mlpPreds, mlpError = mlp(x_train_linreg, y_train, x_test, y_test)
    mlpPreds = reduce(np.append, mlpPreds)

    df = pd.DataFrame()
    #add original training data
    # df["Price"] = input["Close"]
    #add model predictions and errors
    df["LinReg"] = lrPreds
    df["LinRegError"] = lrError
    df["ARIMA"] = arimaPreds
    df["ARIMAError"] = arimaError
    df["kNN"] = knnPreds
    df["kNNError"] = knnError
    df["MLP"] = mlpPreds
    df["MLPError"] = mlpError

    df.to_csv(""+tckr+"_"+yr+".csv", index=False)



def main():

    start10y = "2010-01-01"
    start5y = "2015-01-01"
    start1y = "2019-01-01"
    end = "2019-12-31"

    #TSLA charts
    runModels("TSLA", start1y, end, "1y")
    runModels("TSLA", start5y, end, "5y")
    runModels("TSLA", start10y, end, "10y")

    #MSFT charts
    runModels("MSFT", start1y, end, "1y")
    runModels("MSFT", start5y, end, "5y")
    runModels("MSFT", start10y, end, "10y")
```

```
main()
```