Machine Learning Engineer Nanodegree Capstone Project

Predicting Stock Prices

Franz Williams

August 9, 2018

I. Definition

Project Overview

The history of the stock market goes back to the early 1600s[1], where investors would purchase shares of multiple companies instead of investing in a single risky trade voyage. In its simplest form, investors buy shares of a company's stock with the intent of selling them later, preferably at a higher price to realize a profit. Historically, the act of buying and selling stock was performed directly by humans. More recently however, investors are turning to computers for assistance in the stock trading process. Some examples are: performing analyses of stock price history or market sentiment, automated order placement (algorithmic trading[2]), and utilizing low-latency connections to exchanges to place orders before competitors. In this project I will be attempting to predict future stock price changes based on historical price data, a process known as time series forecasting[3].

For this project, I'll be using daily historical stock prices from several large technology companies: Apple (APPL), Amazon (AMZN), Alphabet/Google (GOOGL), Microsoft (MSFT), and Facebook (FB). Each of these datasets was acquired using Alpha Vantage's[4] free API.

Problem Statement

The goal of this project is to use historical price data to predict the price of a stock several days into the future. The solution I propose is to apply deep learning techniques, specifically a deep feed-forward neural network to accomplish this task. I will be using Keras[5] and scikit-learn[6] to facilitate building and training the neural network model, as well as NumPy and Pandas for data preprocessing. Ideally, our model's prediction will be within 5% of the price 7 days in the future, and also outperform the random walk benchmark described later.

The following is an outline of the steps I intend to take:

- Sanitize and preprocess data
  - Check that our datasets contain sane values; especially since our evaluation metric does not tolerate zero values.
  - Account for the fact that `open`, `high`, and `low` prices were not adjusted for stock splits/dividends, and as a result contain large price gaps.
  - Create our training data: input windows and target output price changes
- Train network and tune hyperparameters
- Evaluate final network performance and compare to our benchmark

[1] https://bebusinessed.com/history/history-of-the-stock-market/
[2] https://www.investopedia.com/terms/a/algorithmictrading.asp
[3] https://en.wikipedia.org/wiki/Time_series#Prediction_and_forecasting
[4] https://www.alphavantage.co/documentation/#dailyadj
[5] https://keras.io/
[6] http://scikit-learn.org/

Metrics

MSE (Mean Squared Error[7]) will be used as the loss function during training. During evaluation however, we will use MAPE (Mean Absolute Percentage Error[8]) to compare both our trained model's and the benchmark's predictions to our ground truth prices. MAPE has several drawbacks to be aware of: it does not allow zero values, and it's biased, since predictions below the ground truth can have up to 100% error while the error for predictions above the ground truth is unbounded. Since we are directly comparing the MAPEs of two predictions, and because our data should not contain any zero values, MAPE should still be a sufficient evaluation metric for this task. MAPE is also intuitive and simple to implement:

$$MAPE = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{P_i - F_i}{P_i} \right|$$

where P_i and F_i are the actual price and forecasted price for each test i, respectively.

II. Analysis

Data Exploration

Each company's dataset is composed of historical price data with following features:

- `timestamp` - the timestamp of the sample; originally in dd/mm/yyyy format
- `open` - the price of the stock when the market opened for the day
- `high` - the highest price achieved during the day
- `low` - the lowest price during the day
- `close` - the price at market close
- `adjusted_close` - the closing price adjusted for stock splits and dividends
- `dividend_amount` - dividend amount paid to shareholders
- `split_coefficient` - this should be > 1 when a stock split occurs, describing the split ratio (2:1, 3:1, etc)

The following is a sample of MSFT's price data before preprocessing, notice the disparity between open/high/low/close and adjusted_close.

timestamp	open	high	low	close	adjusted_close
2000-01-03	117.37	118.62	112.000	116.56	38.4516
2000-01-04	113.56	117.12	112.250	112.62	37.1519
2000-01-05	111.12	116.37	109.370	113.81	37.5445
2000-01-06	112.19	113.87	108.370	110.00	36.2876
2000-01-07	108.62	112.25	107.310	111.44	36.7626
...					
2018-07-11	101.15	102.34	101.100	101.98	101.9800
2018-07-12	102.77	104.41	102.730	104.19	104.1900
2018-07-13	104.37	105.60	104.090	105.43	105.4300
2018-07-16	105.40	105.82	104.515	104.91	104.9100
2018-07-17	104.61	106.50	104.320	105.95	105.9500

Large price jumps as a result of stock splits could cause our model's performance to suffer. Such an example would be on February 18, 2003, where MSFT split shares 2:1, causing a price jump from \$48.30 to \$24.96.

[7] https://en.wikipedia.org/wiki/Mean_squared_error
[8] https://en.wikipedia.org/wiki/Mean_absolute_percentage_error

Distribution summary of each stock's closing prices:

	MSFT	GOOGL	AAPL	FB	AMZN
count	4664.000000	3502.000000	4664.000000	1550.000000	4664.000000
mean	29.970813	424.389694	45.082381	94.199641	249.048499
std	18.101677	274.516655	49.853977	51.664708	340.739318
min	12.007000	50.159800	0.833400	17.729000	5.970000
25%	19.210125	232.715850	3.657800	51.847500	39.097500
50%	22.486200	303.464250	21.962550	84.115000	84.505000
75%	33.451625	574.242500	75.989275	129.467500	306.127500
max	105.950000	1213.630000	193.980000	209.990000	1843.930000

Each stock contains at least 1500 days of price data, and prices vary wildly from stock to stock with means ranging from \$29.97 to \$424.39 and standard deviations ranging from \$18.10 to \$340.74.

Exploratory Visualization

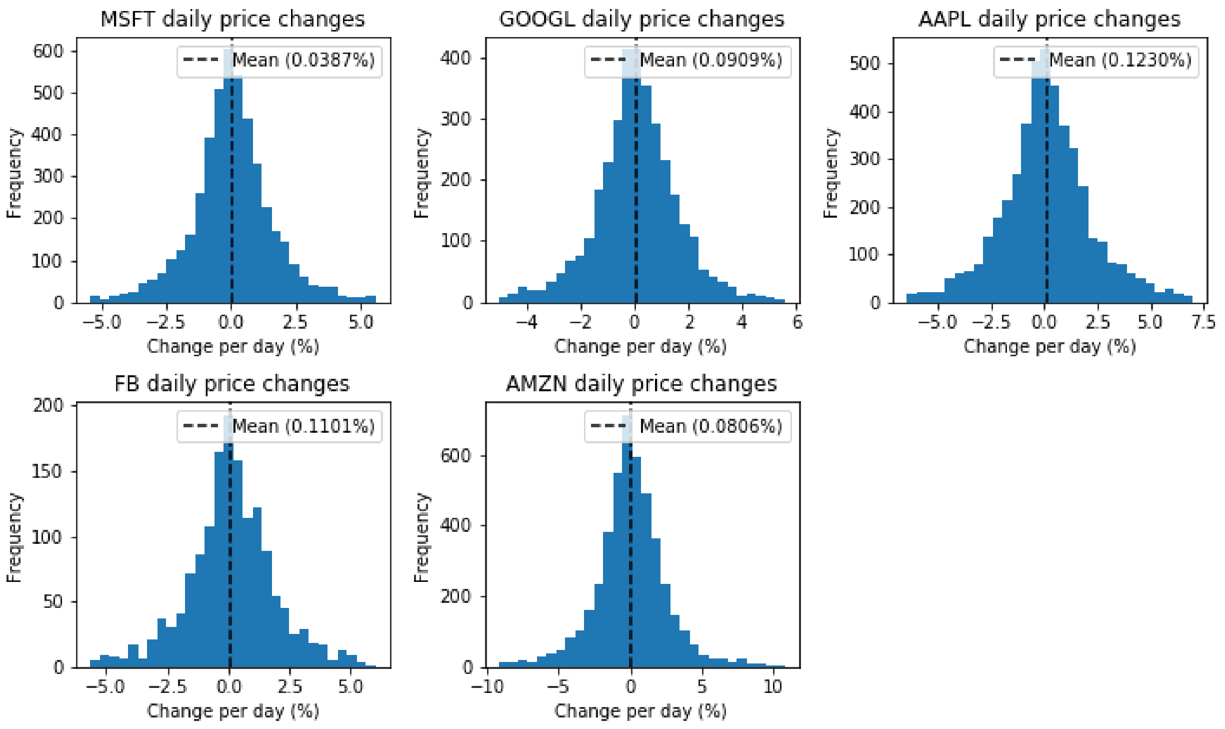

Figure 1: Histograms showing each stock's distribution of day-to-day relative changes, as a percentage. Note how in each stock, the each mean relative change is positive, indicating overall growth.

The relative change above is calculated from the following formula:

$$\text{Relative change}(x, x_{reference}) = \frac{x - x_{reference}}{x_{reference}}$$

where x is the current day's price, and $x_{reference}$ is the previous day's price. The result can be multiplied by 100 to get a percentage value.

Algorithms and Techniques

A neural network will be used as the model; which will receive a window of multiple samples as its input, as a single day's data is likely not enough to form accurate predictions. The network's task would then be to output the relative changes between the current day's OHLC and the next day's OHLC. The next day's OHLC prices can then be reconstructed using the following formula:

$$ohlc\_next = ohlc\_current * (1 + relative\_change)$$

Input windows will be calculated in a similar manner, except when calculating the `relative_change`, the current day will be used as $x_{reference}$, and each previous day will be used as x. This will hopefully allow the network to generalize to stocks that are not in the same price range as our training data.

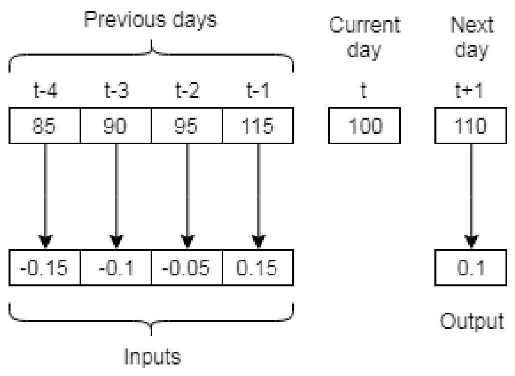

Figure 2: Relative change between each previous day and the current day is calculated

In regards to window spacing, instead of using a consecutive window, I've decided to "exponentially space" samples in the input window:

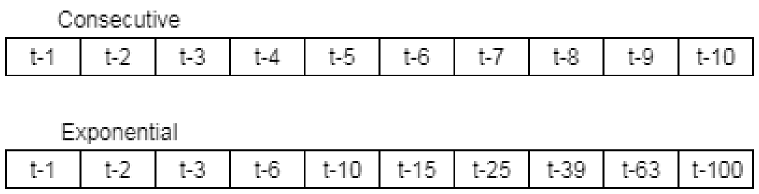

Figure 3: A comparison between a consecutive window and an exponentially spaced window with 10 samples spanning 100 days

The reasoning for such an exponential window is that recent prices are likely more relevant than prices from months prior.

Benchmark

A popular theory regarding stock prices, is that they essentially follow a random walk[9]; so as a benchmark model, I will use a Gaussian random walk[10]. Price changes will be sampled from a normal distribution using the mean and standard deviation of all previous price changes in that stock's history. Samples can then be consecutively added to the current price in order to form the random walk.

[9]https://www.investopedia.com/terms/r/randomwalktheory.asp
[10]https://en.wikipedia.org/wiki/Random_walk#Gaussian_random_walk

$$P_{t+1} = P_t + \mathcal{N}\left(\mu_{differences}, \sigma_{differences}\right)$$

where P_t is the current price, P_{t+1} is the next day's price, $\mu_{differences}$ is the mean of all previous closing price changes, $\sigma_{differences}$ is the standard deviation of all previous closing price changes, and $\mathcal{N}$ represents taking a sample from the normal distribution with the given mean and standard deviation.

III. Methodology

Data Preprocessing

Each stock's price history was first checked for 0 and NaN values. NaNs would indicate missing data, and price values of 0 could interfere with our `relative_change` function and metric calculation. However, all stock price histories were free of 0/NaN values, so no preprocessing was done in this regard.

The data downloaded contained both closing prices (`close`) and "adjusted" closing (`adjusted_close`) prices. However, `open`, `high`, and `low` were not adjusted for stock splits and dividends, so gaps in price values were present wherever a stock split occurred. To remedy this, `open`, `high`, and `low` were each multiplied by the ratio `adjusted_close / close` to bring their units more in-line with `adjusted-close`. In addition, each stock's `close` column was replaced with `adjusted_close`, and the original `adjusted_close` column was removed. After this transformation was performed, each of the `open`, `high`, `low`, and `close` columns represented stock split/dividend-adjusted prices.

The following two tables are data samples from around the February 2003 MSFT split, before and after the transformation, respectively:

timestamp	open	high	low	close	adjusted_close
2003-02-19	24.82	24.88	24.17	24.53	16.237
2003-02-18	24.62	24.99	24.4	24.96	16.468
2003-02-14	47.25	48.5	46.77	48.3	15.9335
2003-02-13	46.41	47.12	46.13	46.99	15.5014

timestamp	open	high	low	close
2003-02-19	16.42895801	16.46867346	15.9987073	16.237
2003-02-18	16.24367628	16.48779327	16.09852564	16.468
2003-02-14	15.58711957	15.99947723	15.42877422	15.9335
2003-02-13	15.31006542	15.54428534	15.217697	15.5014

Implementation

Input window creation

As explained in the *Algorithms and techniques* section, our input data should consist of an input window composed of multiple days' OHLC "relative changes". The window creation process is defined primarily by two variables:

- `window_size`; the number of samples in the window, and
- `window_span`; the total number of days that the window frames will be distributed over.

The resulting window spans `window_span` days, but only contains `window_size * 4` (one for each of open, high, low, and close) values. The distance between each window entry grows as we move back in time; the following code snippet demonstrates this process.

```
window_size = 30
window_span = 300

window_base = np.power(window_span, 1.0 / window_size)
days = [-np.floor(window_base ** x) for x in range(1, window_size + 1)]
# => [-1.0, -2.0, -3.0, -6.0, -10.0, -15.0, -25.0, -39.0, -63.0, -100.0]
```

Here, `days` represents the relative indexes of OHLC frames compared to the current day.

Each window entry contains the *relative change* in prices between the chosen index and the current day:

```
def relative_change(from_val, to_val):
    return (to_val - from_val) / from_val

window[i] = relative_change(data[i - days_before], data[i])
```

where `data` represents a list of OHLC samples and `days_before` is calculated in a similar manner to the entries in `days` above.

The resulting window is then flattened to create a list of `window_size * 4` features.

I've chosen `window_size` to be 30, and `window_span` to be 300. `window_span` was influenced by the fact that there are 365 days in a year, and around 252 days in a stock trading year[11]. During experimentation, I tested `window_sizes` between 10 and 100; a value of 30 seemed appropriate after comparing training time and accuracy.

Target variable

Our target variable is the *relative change* from the current day's OHLC, to the next day's OHLC; calculated using the same `relative_change` function. In this manner, the next day's OHLC can be reconstructed by multiplying the current day's price by `1 + relative_change`.

Model creation

Our model creation function has multiple parameters including number of hidden layers, hidden layer size, dropout rate, learning rate, and activation between layers. The output activation is also tunable, but was usually left at `tanh` to avoid outputs exploding. Using an activation bounded between -1 and 1 is fine for this regression task, as our price differences are not expected to exceed 100% per day.

When calling this function, a model will be returned with equally sized layers with the specified activation and dropout rate. Adam will be used as the optimizer for each model created.

```
def create_model(num_hidden = 2, hidden_size = 100, dropout = 0.0,
                 activation = "relu", final_acti="tanh", lr=0.001):
    model = Sequential()

    for i in range(num_hidden):
        if i == 0:
            model.add(Dense(input_shape = (window_size * len(fields),),
                            units=hidden_size, activation = activation))
        else:
            model.add(Dense(units=hidden_size, activation = activation))

        if dropout > 0.0:
            model.add(Dropout(dropout))
```

[11] https://en.wikipedia.org/wiki/Trading_day

```
    model.add(Dense(len(fields), activation = final_acti))

    opt = optimizers.Adam(lr)
    model.compile(optimizer = opt, loss = "mean_squared_error")

    return model
```

Evaluation

During evaluation, our model will be tasked with predicting prices at least 7 days into the future. In order to facilitate predicting an arbitrary number of days into the future, predicted OHLC values will be reused for future predictions. In this way, predictions form a sort of "sliding window", illustrated below:

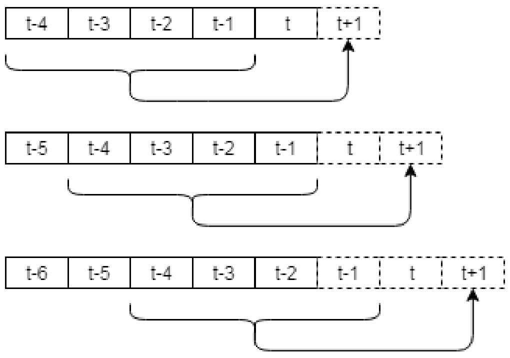

Figure 4: A window of size 4 is slid across a combination of actual and predicted data, allowing the prediction of future values. Here, a consecutive window is shown for simplicity's sake, rather than an exponentially spaced window. The price at t is used as the reference from which `relative_changes` are calculated.

Refinement

The model was trained on 4 out of the 5 stocks, originally without cross validation. MSFT, GOOGL, AAPL, and FB were used as training data, while AMZN was held out for testing.

Trial	No-CV Model	Benchmark
MSFT 7-day	2.47%	5.14%
GOOGL 7-day	2.48%	3.97%
AAPL 7-day	3.58%	6.69%
FB 7-day	2.77%	4.88%
AMZN 7-day	8.09%	11.79%

As expected, model performance on the training data was satisfactory, but fell short on the test data. This was expected, and the next step was to utilize `RandomizedSearchCV` from scikit-learn for hyperparemeter tuning. `RandomizedSearchCV` was chosen over `GridSearchCV` for quicker train-evaluate cycles, as I was initially unsure of what values to allow for the hyperparameters. A 10-iteration, 3-fold `RandomizedSearchCV` was executed many times, with hyperparameter options added and removed between runs.

Below are the final hyperparameter options selected for `RandomizedSearchCV`, each variable here references a parameter in `create_model`:

- `num_hidden = [1, 2, 4]`: Number of hidden layers in the model

- `hidden_size = [50, 100, 200]`: Size of all hidden layers
- `activation = ["relu", "softsign", "tanh", "elu"]`: Activation function applied between layers
- `dropout = [0.0, 0.1, 0.2, 0.4]`: Dropout rate between layers

There were also several hyperparameters left untuned when searching for our final model:

- `batch_size = 1024`: Chosen to speed up training.
- `epochs = 1000`: During early experimentation, the non-CV model continued to slowly improve even after 1000 epochs, so this value was carried over to the cross-validated models.

Out-of-bounds issue

Early on in development, indexes that would cause out-of-bounds history windows (`t < window_span`) were allowed, and these out-of-bounds price changes would be set to 0. Models trained on this data frequently had losses greater than 0.003. After removing this functionality (and consequently placing the restriction that the input dataset must be at least `window_span` days long), our models began performing slightly better, resulting in losses around 0.00025.

IV. Results

Model Evaluation and Validation

The best model found had an average validation loss of 0.000249 with the following hyperparameters:

- `num_hidden`: 4
- `hidden_size`: 100
- `dropout`: 0.2
- `activation`: `'elu'`

Both the model and the benchmark were evaluated using each stock's data across 7-, 30-, and 100-day intervals. Below is the comparison of the model's and benchmark's performance; the MAPE metric was calculated over 100 trials of each (stock, interval) combination:

Trial	Model	Random Walk
MSFT 7-day	2.87%	4.88%
MSFT 30-day	6.62%	10.83%
MSFT 100-day	14.89%	22.94%
GOOGL 7-day	3.52%	4.67%
GOOGL 30-day	8.29%	10.39%
GOOGL 100-day	17.50%	20.17%
AAPL 7-day	4.36%	8.06%
AAPL 30-day	9.09%	12.27%
AAPL 100-day	20.78%	30.18%
FB 7-day	2.86%	4.15%
FB 30-day	5.86%	7.41%
FB 100-day	8.23%	12.99%
AMZN 7-day	5.62%	9.57%
AMZN 30-day	10.83%	20.02%
AMZN 100-day	18.00%	32.27%

The final model outperformed the random walk on all trials, and meets my expectation of less than 5% MAPE on all 7-day trials with the exception of our holdout set, AMZN. The random walk also had a relatively high MAPE on AMZN trials compared to the other stocks, so AMZN may just have a high variance.

Justification

The model did outperform our random walk benchmark. However, the model's predictions are still off by at least 2.8% within 7 days, and upwards of 20% within 100 days. The model also seems to have picked up on the general upward trend of all stocks, which will probably not hold during recessions. For these reasons, I do not believe that this model adequately solves the problem of time series forecasting stock prices.

V. Conclusion

Free-Form Visualization

Below are some sample trials, showing both the model's and the benchmark's predictions against the ground truth. The vertical lines mark the boundary where predictions start. The number of days since trial start is used here, since if actual dates were desired, weekends and stock market holidays would have to be taken into account.

Note how the model seems to generally predict upward, with small variations based on the pre-prediction data.

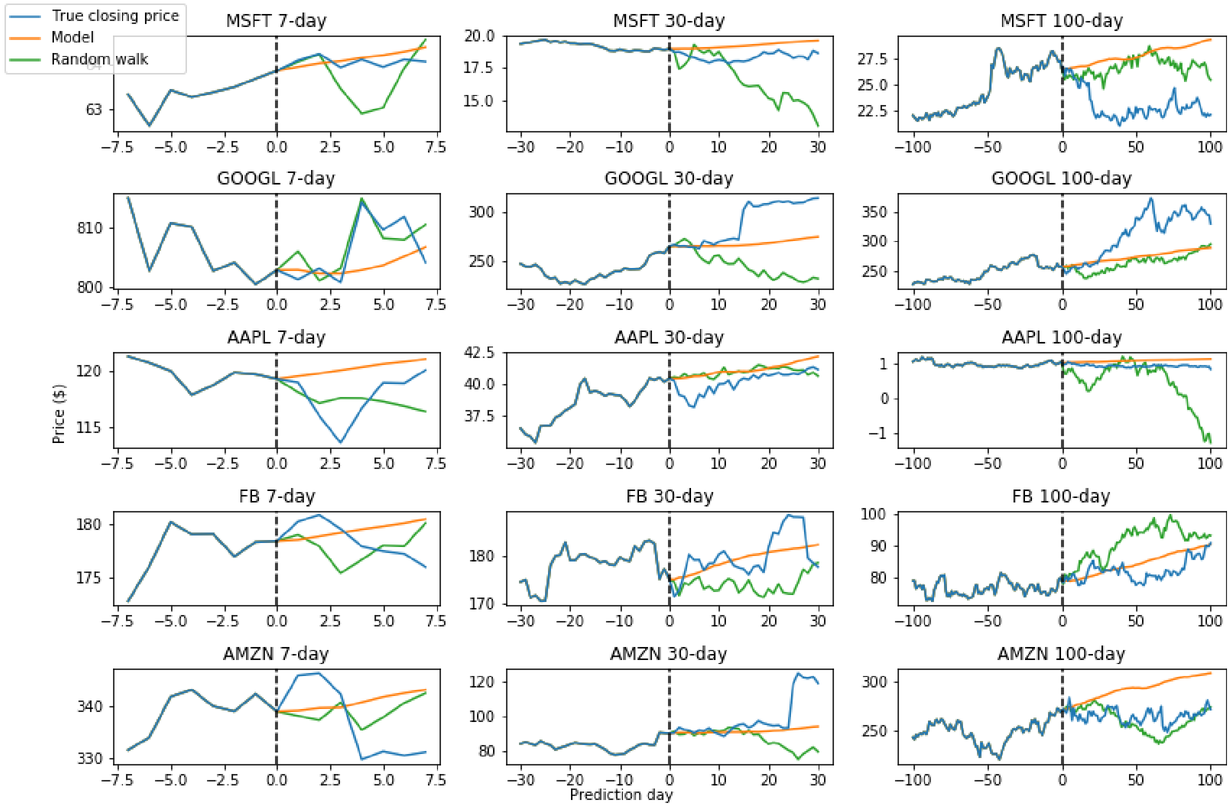

Figure 5: Sample predictions for each stock-interval combination

Reflection

The following steps were performed during this project:

- Each of our datasets was checked for NaNs/zero values.

- All OHLC data was adjusted for stock splits and dividends.
- `relative_change` and `get_window` functions were created to facilitate processing of input and output data.
- A `create_network` function was designed for use with `RandomizedSearchCV`.
- `RandomizedSearchCV` was used to tune hyperparameters and train the network.
- The final network was evaluated and compared to our benchmark across each of our original datasets and multiple time intervals.

The most difficult part of this project was cycling between making adjustments to the algorithm, and then having to wait for `RandomizedSearchCV` to run with the new changes. A lot of research also had to be done on the sklearn/Keras end, particularly figuring out how to integrate a Keras model into `RandomizedSearchCV`.

In conclusion, despite beating the benchmark I set, I would unfortunately not trust this model enough to base any investment decisions on. However, it would be interesting to apply this algorithm to a time series with more obvious patterns, such as seasonal weather data.

Improvement

More accurate predictions might have been achievable using a larger `window_size` or more training epochs, both of which had to be limited in order to have an acceptable training time.

OHLC predictions are also left “unchecked”, allowing for odd combinations of OHLC to be output such as lows above highs, and open/close prices outside of the range established by high/low.

When conducting research on time series forecasting, I found articles that offered alternative methods of solving similar problems; most notably RNNs (recurrent neural networks). These networks are a good fit for time series, as they are capable of storing representations of previously seen data.