Machine Learning Engineer Nanodegree

Capstone

Chris Rectenwald

2/1/2019

Creating Stock Price Predictors for Applications in Investing and Trading

Domain Background

Investing, as Warren Buffet would say is **"…** the process of laying out money now to receive more money in the future." However, investing can be applied to many contexts. For instance, investing in one's athletics career would be to train and study a game hard for greater knowledge and strength in the future that would allow one to excel against his or her competitors.

In this case, investing will apply to only the finance domain, and will focus on the asset class, a group of instruments that are grouped together based on having a similar financial structure, of stocks in the NASDAQ and NYSE. Also, for future reference, here are definitions that investing and trading will have going forward:

1) Investing: to commit (money or capital) in order to gain a financial return.
2) Trading: an economic concept that involves the exchange of goods and services between parties. The most common medium of exchange is money for these transactions, but other mediums of exchange can exist that are referred to a barters. In financial markets such as the NASDAQ and NYSE, trading refers to the buying and selling of securities such as stocks.

Despite focusing my undergraduate studies in Electrical Engineering, two areas that have always interested me are finance and computer science. Both fields are complicated beyond comprehension, but I do see an opportunity to intersect these fields when applying machine learning principles to possibly create opportunities for the average citizen grow their personal wealth whether the opportunity is created through machine learning models to help investors receive high stock market returns in a cost-effective manner or helping an average citizen identify areas where he or she can cut spending to save more.

Problem Statement

Although there are many ways to invest and trade in the stock market with complex derivatives, such as buying and selling call or put options, or to also buy a company's stock and hold for long periods of time, each strategy boils down to the goal of buying a security low and selling said security high. With that, the problem aiming to be solved is to create a reliable predictor of a future price of s tock, given the historical stock price data and financial ratios of said stock, using machine learning and financial knowledge.

Datasets and Input

A dataset that I will be using will be from Yahoo finance's historical data, and will be able to get historical data from over 8,000 stocks trading for all current S&P 500 companies . To obtain this data is free.

I plan to use the historical data and the current stock price available to train my model, so I can predict future stock prices, given that a certain stock will behave similarly over time with its current and historical characteristics. The historical stock data will be used for exploratory data analysis (EDA), feature selection, and model training to predict future stock prices.

Here are the characteristics of the data that I plan to use:

Name	**Value**	**Description**
Name	AAL	Stock symbol for company. A unique combination of letters designed to identify specific company/security
Date	2013-02-08	Lists the date that corresponds to any stock's volume, low price, price at close, high price, etc
Open	15.07	The opening price of a certain security on a certain day in dollars
High	15.12	The high price of a certain security on a certain day in dollars.
Low	14.63	The lowest price of a certain security on a certain day in dollars.
Close	14.75	The closing price of a certain security on a particular day in dollars
Volume	8407500	The number of times a security is traded during a given trading day.

Here's what it looked like in the jupyter notebook for the stock GE:

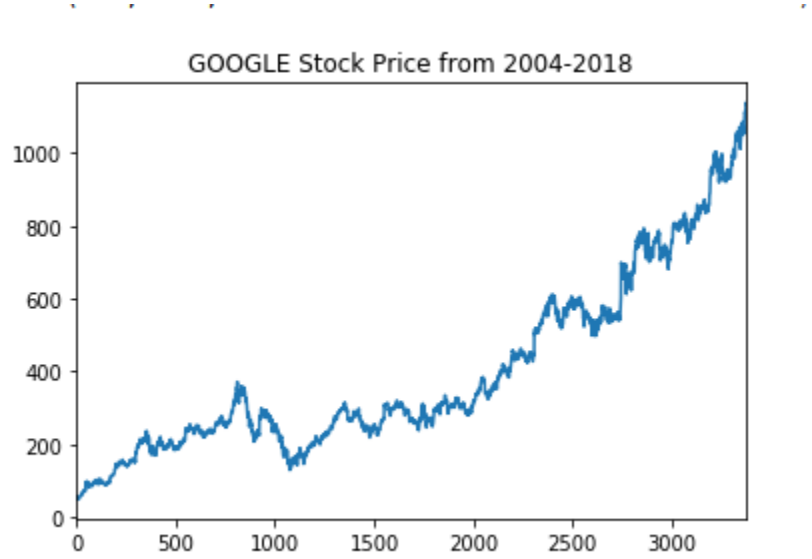
	Date	Open	Volume	High	Low	Close	Adj Close
0	2004-08-19	32.630001	13998500	32.740002	32.330002	32.709999	19.719440
1	2004-08-20	32.660000	16269200	32.799999	32.490002	32.650002	19.683273
2	2004-08-23	32.680000	13402200	32.740002	32.459999	32.509998	19.598867
3	2004-08-24	32.700001	15862300	32.750000	32.410000	32.630001	19.671211
4	2004-08-25	32.520000	15740300	32.990002	32.419998	32.790001	19.767666

Solution Statement

For this project, the task is to build a stock price predictor that will leverage historical stock data over the time period to cast predications of a security's price movement. A training model would use the data range mentioned previously and a list of ticker symbols given a user's input, and builds a model of their stock behavior. The query interface should accept a list of ticker symbols and the amount of shares of each symbol and output the predicted closing stock prices of each stock over, by comparing the predicted price to the actual price.

A potential solution could be to make a supervised learning model trained on historic data to predict the future closing price of a stock for the month after the dataset ends. Hopefully this knowledge would help put the user in an advantageous position to increase their portfolio value.

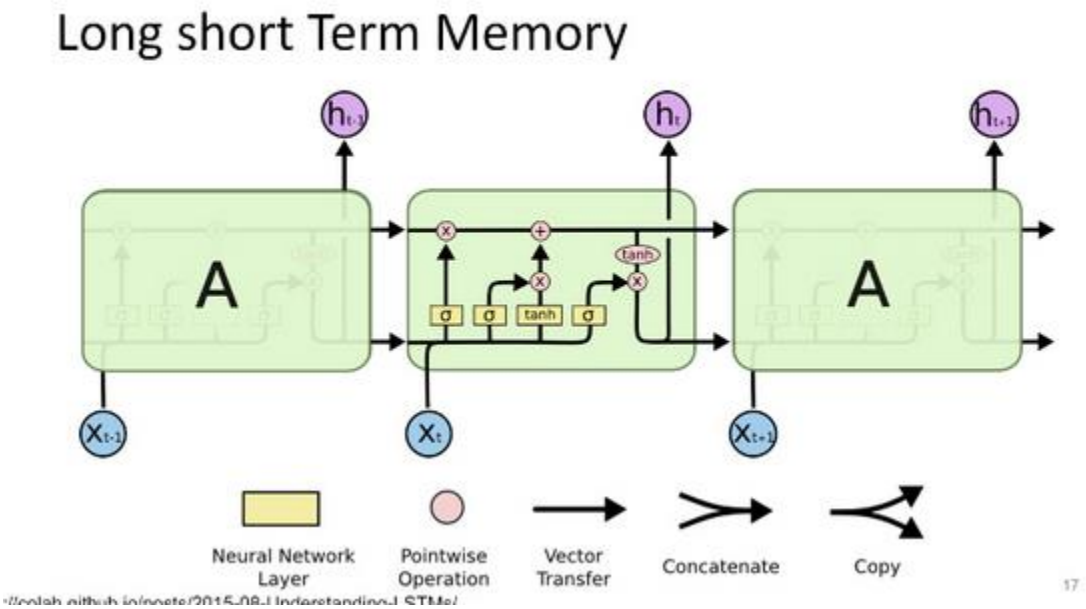
Throughout the project, I wanted to predict the values of the adjusted closing prices of a given stock and trend the data. Below is an example:



An objective of this project was to study time series or in other words, sequenced data that depends on the previous point. I decided to use a Long Short Term Memory (LTSM), a modification of a Recurrent Neural Network (RNN). LTSM networked contain memory cells that are controlled with the help of gates. This is derived from a drawback of Recurrent Neural

Networks even though they share the same architectures as RNNs. In LTSMs, the memory are called cells whose inputs are the previous state and current input. The cells decided what input to keep in their memory, and then combine the input, previous state, and the current memory that aim to capture long-term dependencies. I think that LTSMs were suitable for the problem that I am trying to solve.

Furthermore, the input gate is responsible for the addition to the cell state. There's essentially three steps in receiving the input information: regulating input values through the sigmoid function, creating a vector containing all possible values, and multiple the regulatory factor value to another vector that should be used as information to the cell state. This can illustrated in the figure below.

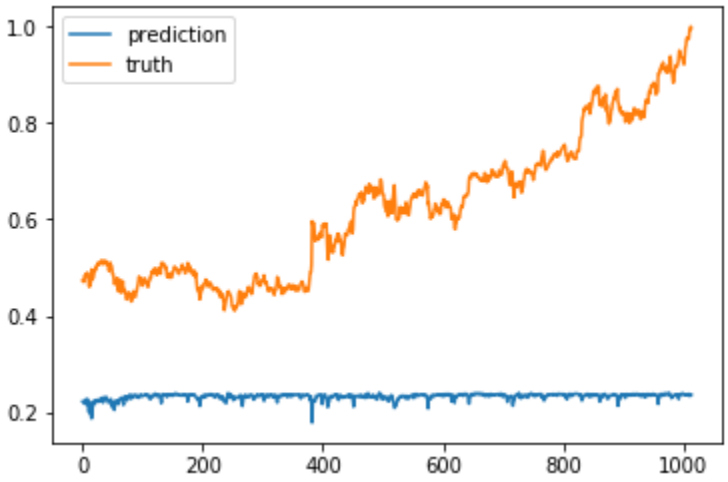


(Reference 4)

Benchmark Model

A benchmark value will be the returns and sharpe ratio of the S&P 500 Index, which basically measures the overall movement of the market effectively. I can do this by calculating the portfolio returns in terms of percentage, subtract that number by the risk-free rate, then divide the result by the standard deviation of the closing price of that stock over that period.

The benchmark model for this project was using a linear regression, because my main goal was to see how well a machine learning model could accurately predict stock prices. Here's the linear regression model with the calculated errors below:




```
from sklearn.metrics import r2_score, mean_squared_error
r2_score(y_test,predicted_price),np.sqrt(mean_squared_error(y_test,predicted_price))
```

```
(-6.883813509711871, 0.4162894089239953)
```

Methodology:

1. Data Filtering or Pre-Processing
    a. Attain data from yahoo finance
    b. Because there's different stocks with different prices and different scales, per se, as well as different volume scales, we need to scale or normalized the inputs to make the neural network converge quicker than the alternative. Then we normalize the data that is a requirement for LSTMs with the MinMaxScaler of the sklearn library
    c. Split the dataset into test and training data (30% and 70%, respectively)
    d. Utilize the window concept to help attain sequence generation to help accuracy of model. In this case, the window was set to 5 days.
2. Implementation (Benchmark Model/Linear Regression Model)
    a. Import the necessary python libraries. In this case I used the sklearn library that
    b. Create an object for the model
    c. Train the data: Use model.fit(), the fit function with the feature vector (X_train) and label vector (y_train) a the inputs
    d. Then predict the values for the model that the training data does not see using the model.predic() function with the feature vector of unseen data as an input
    e. For linear regression we get the slope coefficient and y-intercept
3. Implementation (LSTM network)

a. Using the Keras Module, we need to import the type of layers we need to add in the output model. In this case, they were the following: Dense, activation, Dropout, and LSTM.
b. Next we create an object for the model by calling the Sequential function then add layers to the model with the function add.
c. For the first LSTM layer there are 3D arrays as inputs with a dimension or number of neurons as 300. We aspire to have a full output sequence, so I set the return_sequence value to True.
d. Add a dropout layer of 0.2, so a LSTM network with the added dropout layer.
e. This process continues until the final layer has been reached 1 neuron with a sigmoid function.
f. Compile the sigmoid function using loss function parameter, optimizer and metrics in order to check for the model's efficiency.
g. Give input of the number of epochs, batch size and validation split to help train the model. An issue I had was figuring out the setting the input dimnesions for the two models. This required an understanding of the difference in data dimensions.

```
#initialize model


from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.layers import LSTM,Dropout

model = Sequential()


model.add(LSTM(units=300, return_sequences=True, input_shape = (X_train.shape[1], 4)))
model.add(Dropout(.2))


model.add(LSTM(units=150, return_sequences=False))
model.add(Dropout(.2))




model.add(Dense(units=1, activation='sigmoid'))

# Compiling the RN
model.compile(loss='mae', optimizer='nadam', metrics=['mean_squared_error'])

# Train :)
history = model.fit(X_train,y_train, epochs=100,validation_split=0.1, batch_size=32)

############ Save & load Trained Model ############
# Save Trained Model
model.save('TICKER-RNN.h5')
```

```
Using TensorFlow backend.
```

4. Refinement
    a. New model was created with using windows. This function takes the window length as input and then iterates through the whole data for the end date of the window.

```
(2364, 1)
```

```
#Window Creation


def windowing(data, window):

    sequence = []
    for index in range(len(data) - window):
        sequence.append(data[index: index + window])
    return np.asarray(sequence)
```

b) The LSTM now looks like the figure below:

```
#initialize model


from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.layers import LSTM,Dropout

model = Sequential()


model.add(LSTM(units=300, return_sequences=True, input_shape = (X_train.shape[1], 4)))
model.add(Dropout(.2))


model.add(LSTM(units=150, return_sequences=False))
model.add(Dropout(.2))




model.add(Dense(units=1, activation='sigmoid'))

# Compiling the RN
model.compile(loss='mae', optimizer='nadam', metrics=['mean_squared_error'])

# Train :)
history = model.fit(X_train,y_train, epochs=100,validation_split=0.1, batch_size=32)

############ Save & load Trained Model ############
# Save Trained Model
model.save('TICKER-RNN.h5')
```

```
Using TensorFlow backend.
```

Now, there are a sequence length, number of layers, the loss function and the optimizer hyper parameters.

Evaluation Metrics

(approx. 1-2 paragraphs)

Two evaluation metrics that I think will be appropriate for measuring the benchmark and the solution model will be the R^2 score metrics as well as the Root Mean Squared Error (RMSE).

1) **R^2 Squared-** this is an indicator of how good the predicted data fits to the actual values. Ideally a good model will have a high R^2 squared model of above a .8 (max is 1)

2) **RMSE –** This is used to measure of the differences between values predicted by the solution model vs the values observed. RMSE can never be negative, and the closer the RMSE is to 0, the closer the model fits the observed data.

Results Context of Project

Final parameters of each model are:

- 1st LSTM Model:
  - Batch size: 32
  - Epoch number = 100
  - Number of neurons in layers: between 300 and 150
  - Calculating the errors:

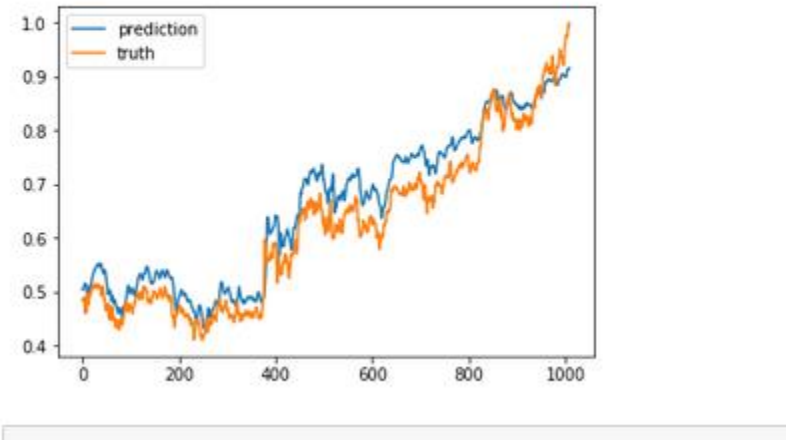
```
In [19]: trainPredict = model.predict(X_train)
         testPredict = model.predict(X_test)

In [20]:
from sklearn.metrics import r2_score, mean_squared_error
r2_score(y_test,testPredict),np.sqrt(mean_squared_error(y_test,testPredi

Out[20]: (0.9110467131714055, 0.04421892604371353)
```

-

```
plt.show()
```

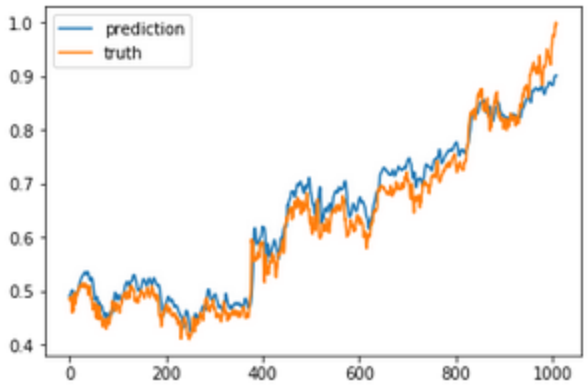


- 2nd (Improved) LSTM Model
  - Batch size=32
  - Epoch number=20
  - Number of neurons in layers = Between 200 and 100

```
In [41]: from sklearn.metrics import r2_score, mean_squared_error
         r2_score(y_test2,testPredict),np.sqrt(mean_squared_error(y_test2,testPre

Out[41]: (0.9560398615848988, 0.031087049091358874)
```

```
:  plt.plot(testPredict, label='prediction')
   plt.plot(y_test2, label='truth')
   plt.legend()
   plt.show()
```



Model follow up: in order to see if my model was good for other stocks, I tried the new and improved LSTM for GE's stock price. I ended up getting a R2 score of about 96% that while high, could be too high.

```
from sklearn.metrics import r2_score, mean_squared_error
r2_score(train_Y,trainPredict),np.sqrt(mean_squared_error(train_Y,trainP

(0.9639816635549392, 0.04097459145117883)
```

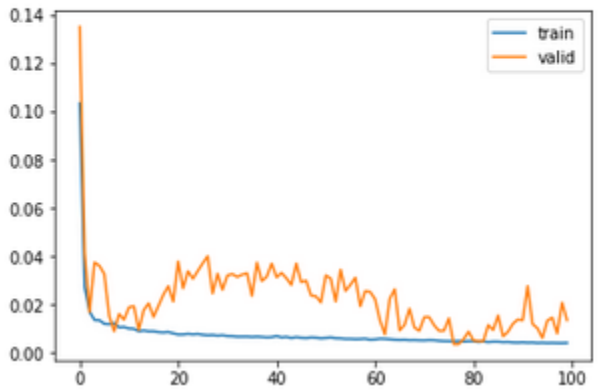
```
plt.plot(trainPredict, label='prediction')
plt.plot(train_Y, label='truth')
plt.legend()
plt.show()
```

Conclusion

While the r2 score for the benchmark model was around a -6.8 that indicates that the model does very well fit the data, and was improved with the LSTM model initially to 91% and then 96% after additional revisions to the LSTM model. Thus this LSTM could be trusted to make predictions for GOOGLE stock in the past. In addition, the LSTM model proved to work well with past GE stock data, with a very high r2 score.

Something that I have taken away from this project is the importance of the convergence of the loss function in neural networks. The convergence can indicate whether the model is trained in addition to telling one if the model is either underfitted or overfitted. The figure below shows the learning curve of the process of the model. As one can see, the loss for both the training and the validation set decreased as the number of epochs increased, while converging to zero, which indicates a successful training.

```
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='valid')
plt.legend()
plt.show()
```



For improvement purposes, while I was able to create a LSTM model with relatively high R2 scores for multiple stocks, I believe that my model is far from real life interpretation as the market conditions can be too dynamic, the speed of the model could not be that to suit desirable trading styles that can reach high returns, or look into GRUs to improve my method with LSTMs. All in all, this project was enjoyable while challenging.

References

1) https://www.quantstart.com/articles/Forecasting-Financial-Time-Series-Part-1
2) http://sebastianraschka.com/Articles/2014_intro_supervised_learning.html
3) https://www.investopedia.com/university/beginner/beginner1.asp

4) https://www.altoros.com/blog/text-prediction-with-tensorflow-and-long-short-term-memory-in-six-steps/