Stock Market Prediction with LSTMs

Project Final Report
ITCS 5156 - Applied Machine Learning

A project report
Submitted By:-
Group 9
Project Repo: https://github.com/aceetheridge/MLProject.git

Name	**Niner Id**
Aaron Etheridge	800936351
Suhas Raju	801026625
Sathwik Doguparti	801217248
Sapna Pareek	801165364
Neel Shah	801203511

At
University of North Carolina at Charlotte

Introduction

After some initial research, it would appear that Long short-term memory (LSTM) systems are standard practice throughout the industry for tackling stock market prediction. These systems can recognize old patterns while also adjusting fast enough to keep up with rapid rises and drops in the stock market. Using the Huge stock market dataset from Kaggle.com, as well as some other market indicators like the CBOE volatility index we will examine if providing these indicators to an LSTM system can provide an accurate prediction of the future value of the stock market.

The Problem:-

The stock market has always been a challenge for the average person to accurately predict. A commonly stated statistic suggests that only the top 3% of all investors are able to consistently outperform the standard market growth of 5~7% a year. There are a myriad of social, political, and economic factors that can have an effect on the stock market; these factors are the reason why so few investors are able to outperform the market. The problem then becomes building a tool that can see through the fog of complex analysis and be able to make accurate predictions for the future.

Motivation:-

Possessing the ability to comfortably put your money into an asset with some level of assurance that it will perform well is a very tempting concept. At some point, every member of our team has tried their luck with the stock market. None of us has had any significant success and we would all like to have a tool that could provide some assistance to our future investments. At the same time building this tool would also provide more insight and understanding into the stock market and the analysis behind it.

Known Challenges:-

Data collection relating to the stock market can be difficult. It is relatively easy to get the simple metrics like price and volume but other more sophisticated metrics can be difficult for an individual to acquire as they are often property of firms that provide professional analysis as a business model. As there are many factors that play a part in the stock market, it can be difficult to associate a numerical value to something like a world event like Covid-19 pandemic. Stock market data also has a relativity problem, in that data collected from the 1970's may not provide an accurate example of what would happen in more modern trading scenarios.

There are also problems when building a general predictive model for any type of stock. As the metrics between multiple stocks can be very different. Every stock can have a vastly different price range, volume, and other input factors that may not be applicable to all stocks so building a 1 size fits all model is very difficult.

Concise summary of your solution (approach) :-

Knowing that the task of building a highly accurate prediction method has been a pursuit of large multi-billion dollar companies for decades. It is impractical to think that our solution could outperform those proprietary systems. We attempted to build a simpler system with the end goal of being able to make simple predictions of the next day's value.This system is only to provide some level of certainty of what the stock price may be. Using the Tensorflow-Keras python library, we intended to build a LSTM system that could accurately predict the future price of a stock. For this approach we focused on one specific stock, Pfizer (PFE). This method is not exclusive to Pfizer but within the Huge Stock Market Dataset, Pfizer had the most number of records dating back to the early 1970's. This gave us the most number of datapoints to work with. Firstly, we looked at the performance of simple input metrics like closing price, high, low, and volume. We then tried combining some of these features to see if the performance improves over single features. We then looked for more sophisticated input metrics that would work well with our original dataset. We then tried to create a new loss function to more accurately represent a rise and fall of the stock price. After the model was built we then tinkered with the data as well as tested different hyperparameters to find the best settings for our model.

Literature Survey

In order to identify different directions for future stock market predictions using machine learning, Strader et al [1] in 2020, provided us with a systematic literature review that categorized relevant peer reviewed journal articles from the past 20 years. Based on the study by Strader et al. machine learning stock market research taxonomy fits into one of the 4 categories.

-Artificial neural networks

-Support vector machines

-Genetic algorithms coupled with other techniques

-Studies using hybrids of artificial intelligence approaches.

Among the various categories mentioned we are particularly interested in exploring simple linear regression methods and LSTM, which is a type of Recurrent Neural network.

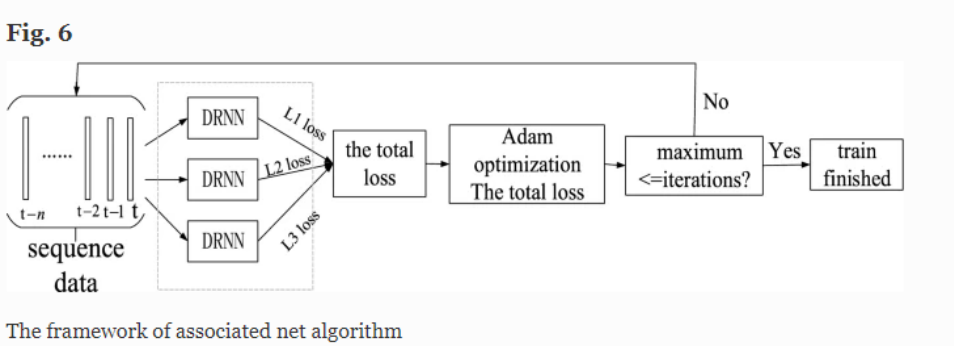
Even though bivariate and multivariate linear regression are very simple and fail to produce accurate results most of the time on cross sectional data [2-6], they will serve as an effective learning tool in understanding basic working structure of regression models.

Moreover, when it comes to advanced models like LSTM, Jia [7] proved the effectiveness of LSTM for predicting stock price at a RMSE score of .0265. Further

studies by Li [8], Ding[9] and Sidra[10] which used LSTM with a combination of other techniques, further cemented LSTM as an effective method to predict stock trends.

Jia [7] explored the effectiveness of an LSTM system when trained by backpropagation. The author research tested a wide range of LSTM networks with a combination of different hidden layer sizes and demonstrated an impressive prediction RSME of 0.0265 in predicting stock market prices. The research taught us that even simple LSTM systems can be very effective at predicting time series data, which is similar to stock market data. Jia's system seemed very easy to implement and analyze but lacked depth and failed to account for multiple input features which we would be testing in our approach.

Ding Et al. [6] proposed a new model that could handle multiple inputs as well as produce multiple outputs with an prediction accuracy of over 95%. Their model used a "LSTM-based deep RNN" also called an associated net model. The model was able to produce low, high, and average prices of a stock for the next day. The model used differed from a traditional LSTM model because it utilized a RNN which allows for the "temporary discarding of the neural network unit" and thus prevented the overfitting of the models. The authors claim that there is room for optimization in thier mean algorithm as it "fails to take account for the relationship between each sub-loss" and other issues. The authors demonstrated a well designed system but this system seems too complex for our proje ct's needs. We will need multiple inputs but not multiple outputs.



Ding [6]

Li Et al.[8] proposed a framework of a LSTM with real-time wavelet denoising. The authors demonstrated a flaw, with then current literature, that future data was included in the training phase. The authors proposed denoising approach utilized a sliding window mechanism instead of a Fourier transform or moving average. Their improvised model attained significant improvements over a traditional moving average method of denoising and also performed well during back-testing of results. However, the authors also pointed out their approach can sometimes be impractical as the future data value is required as part of the input for the prediction task. This research is important to our model because it recommends the use of a moving window when preprocessing the data. A moving window helps to denoise the system and stop outliers

from having too much input in our system.

Li Et al. [8]

How everything relates to our approach:-

The pieces of literature that we examined in our research, helped to guide our team in some of the major design decisions when building our LSTM model. Other recent literature includes a multitude of different methods and data types that could accurately predict the stock market. These methods ranged from social media coverage, to news analysis, and other financial measurements. However, with the currently acquired data and supporting literature, exploring LTSM based models is the method we chose to go with for this project. The literature also affirmed that there is a large amount of investment into LSTM systems and stock market prediction.

Method

Preparing Data:-

The first step in building an LSTM was to import the training and target data. We found a web package in the pandas_datareader library that could import all of the stock data that we needed from this dataset. This function read in all the data between a start and end date and utilized the web to import the data.

```python
def load_data(company, start, end):
    data = web.DataReader(company, 'yahoo', start, end)
    return data
```

Fig 1. LoadData Function

The function in figure 1 returned the, high, low, open, close, volume, and adjusted close. We chose the close price as the target for the experiment because close occured at a consistent time in the day while high and low could be at any time.

Date	High	Low	Open	Close	Volume	Adj Close
1972-06-01	0.815346	0.802993	0.000000	0.815346	2458771.0	0.187400
1972-06-02	0.817817	0.802993	0.815346	0.805463	1613885.0	0.185129
1972-06-05	0.807934	0.798051	0.805463	0.802993	2585251.0	0.184561
1972-06-06	0.825229	0.800522	0.802993	0.820288	2347469.0	0.188536
1972-06-07	0.820288	0.807934	0.820288	0.820288	1032077.0	0.188536
...	...	...	...	...	...	...
2019-12-24	37.419353	37.191650	37.314991	37.277039	5187683.0	35.428619
2019-12-26	37.542694	37.191650	37.286530	37.362431	9384078.0	35.509777
2019-12-27	37.590134	37.239090	37.409866	37.305504	10117662.0	35.455673
2019-12-30	37.400379	36.897533	37.286530	36.916508	11554264.0	35.085968
2019-12-31	37.191650	36.726753	36.802658	37.172676	15175703.0	35.329433

12001 rows × 6 columns

Fig. 2 Pfizer Dataframe from 1972-2020

As stated earlier we chose Pfizer(PFE) as our stock of choice for these experiments because it had the most number of samples in our data set. Stock market data is very well maintained so there wasn't a lot of pruning or dropping that needed to be done to get the data into a workable format.

Next we knew that we would need another data source beyond just the normal stock metrics for our system. After doing some searching around the web, we found a pretty popular indicator called the CBOE volatility index. This indicator factors in elements like total volume change, rate at which the price is changing, and overall market trends and combines all of these factors into a single numeric value that represents the volatility of a stock on a particular day. After downloading a CSV file from the CBOE website we had to read in the data and merge it with our original data at the corresponding dates.

```
df_cboe=  pd.read_csv('./cboe.csv')
df_cboe['Date']=df_cboe['Date'].apply(lambda x: datetime.datetime.strptime(x, "%m/%d/%Y").strftime("%Y-%m-%d"))
df_cboe=df_cboe.dropna()
display(df_cboe)
```

Fig 3. Load CBOE function

The date time format had to be changed from the original mm/dd/yyyy format to yyyy-mm-dd format in order to blend the two data sources together. The resulting data is represented in figure 4.

```
data = pd.merge(data, df_cboe, how='inner')
display(data)
```

	Date	Open	High	Low	Close	Volume	OpenInt	VIX Open	VIX High	VIX Low	VIX Close
0	2004-01-02	27.731	28.136	27.667	27.825	21170596	0	17.96	18.68	17.54	18.22
1	2004-01-05	28.152	28.567	28.136	28.567	39160284	0	18.45	18.49	17.44	17.49
2	2004-01-06	28.802	28.802	28.371	28.550	29626644	0	17.66	17.67	16.19	16.73
3	2004-01-07	28.348	28.802	28.293	28.794	21424852	0	16.72	16.75	15.50	15.50
4	2004-01-08	28.802	28.820	28.231	28.530	23630636	0	15.42	15.68	15.32	15.61
...	...	...	...	...	...	...	...	...	...	...	...
3486	2017-11-06	35.268	35.288	34.802	35.000	10722965	0	9.63	9.74	9.38	9.40
3487	2017-11-07	34.980	35.099	34.921	35.040	10992031	0	9.31	10.31	9.29	9.89
3488	2017-11-08	35.050	35.070	34.713	35.020	13693200	0	9.79	10.27	9.50	9.78
3489	2017-11-09	35.080	35.250	34.840	35.200	13364668	0	9.94	12.19	9.79	10.50
3490	2017-11-10	35.100	35.190	34.795	35.180	15045115	0	10.78	11.58	10.50	11.29

3491 rows × 11 columns

Fig 4. CBOE Dataframe and Merge Function

After the dates were adjusted we were able to combine the two dataframes and remove any rows that didn't have all the necessary columns. As the CBOE index only started in 2004, this significantly reduced the total number of entries of data down from 12000 to just 3500. Because of this we kept two separate dataframes, one with both the VIX and Stock info, as well as one with just the stock info.

Scaling:-

Research on the web [11-12] seemed to suggest that the standard scaling method for dealing with stock market data is the MinMax scaler. Applying this scaler ranged all of our data from 0 to 1.

```
scaler = MinMaxScaler(feature_range=(0,1))
```

```
df_sc = scaler.fit_transform(new_data.values)
display(df_sc)
df_sc.shape
```

```
array([[0.0082445 , 0.00805147],
       [0.00803444, 0.00528481],
       [0.00798195, 0.00846564],
       ...,
       [0.78379877, 0.03313121],
       [0.77553113, 0.03783549],
       [0.78097568, 0.04969422]])

(12001, 2)
```

Fig 5. MinMax Scaler function

In our first iteration of this model we scaled both the train and target because we didn't realize that the scaled target data would affect our loss scores. During the first round of testing we were getting MSE loss scores of less than .001. However, because MSE measures the difference between two numbers, if the numbers themselves are small the MSE score becomes smaller, skewing our results. Eventually, we came back and to this stage and subsequent steps in the preprocessing and undo the scaling done to the target data. This gave us more accurate MSE results for the rest of our tests.

The next step was to break the data down into segments for training, validation, and testing. We chose to do a 60% train, 20% validation, and 20% testing split. With just our stock metrics dataframe, this gave us 7200 trainable samples but with the VIX data that amount was cut to just 2100 samples. Concerned with the limited data of the VIX dataset the ratios of the split were adjusted to 80%-10%-10%.

Building the Sliding Window:-

Our research indicated that sliding windows were important to implement in our system to denoise data. Once the data was properly split, the sliding window function broke the data into segments of a variable window size as shown in figure 6.

```
window = 30
```

```
def prepData(scaled_data, unscaled, window, numFeatures):
    x_train = []
    y_train = []

    for x in range(window, len(scaled_data)):
        x_train.append(scaled_data[x - window:x])
        y_train.append(unscaled[x,0])
    x_train, y_train = np.array(x_train), np.array(y_train)

    x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], numFeatures))
    print(x_train.shape)
    print(y_train.shape)
    return x_train, y_train
```

Fig 6. Sliding Window Function

This function builds a train and target set of data windows. As mentioned earlier in the scaling section, this function originally returned the scaled version of train data and the target (line 7). Later the scaling on the Y was removed as it was affecting our MSE loss values.

```
x_test, y_test = prepData(test_df, un_test_df, window)
display(x_test)
display(y_test)
```

```
(2371, 30, 2)
(2371,)

array([[[0.29762327, 0.24099464],
        [0.29540514, 0.15778438],
        [0.2927837 , 0.15132367],
        ...,
        [0.29338863, 0.13927891],
        [0.29520348, 0.2019836 ],
        [0.29338863, 0.15198324]],
```

Fig 7. Shape of Data After Sliding Window

As the output in figure 7 shows the data was now broken into segments with a window size of 30 and 2 input vectors.

Building the LSTM:-

Following the Keras documentation, and some examples online we started with a simple LSTM model structure. With two layers and one input feature dimension.

```
def LSTM_model():

    model = Sequential()

    model.add(LSTM(units = 50, return_sequences = True, input_shape = (x_train.shape[1],1)))

    model.add(LSTM(units = 50))

    model.add(Dense(units=1))

    return model
```

Fig 8. Original Model Structure

The first iteration of the model needed to be simple to make sure our process was correct before tweaking the structure. Units is the dimensionality of the output space. Which means that for our purposes it needs to be greater than 2 but not too large as our data was fairly limited. From examples we found online it seemed as though anywhere between 50 to 100 performed similarly. The return sequences property makes the layer return the entire output of the layer and not just the last iteration of the output. This is important to have on because we are passing a sliding window to this layer and we want that data preserved for the next layer. The input_shape simply had to match the input shape of the data we were passing to the layer. As this was the first iteration with only one input feature the second parameter was set to one.

```
def LSTM_model():

    model = Sequential()

    model.add(LSTM(units = 100, return_sequences = True, input_shape = (x_train.shape[1],numFeatures)))
    model.add(Dropout(dropOut))

    model.add(LSTM(units = 50, return_sequences = True))
    model.add(Dropout(dropOut))

    model.add(LSTM(units = 50))
    model.add(Dropout(dropOut))

    model.add(Dense(units=1))


    return model
```

Fig 9. Improved Model

Firstly we added dropout layers but this initially hurt our performance with just one input feature. Once multiple input features were added the dropout layers improved the losses. We tried between 2 and 5 LSTM layers but it seemed that 3 was the sweet spot in terms of performance. More than 3 we saw over fitting to the train data.

Compiling the Model:-

The adam optimizer was the one that appeared most often in our research so it is the one we chose to go with throughout the experiments. Having the correct loss function for the task at hand is very important to the quality of the model. Mean Squared Error is a common, easy to implement loss metric that can be used with stock market prediction because it calculates the difference between the projected price and the actual target price.

```
model = LSTM_model()
model.summary()
model.compile(optimizer='adam',
              loss='mean_squared_error')
```

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_27 (LSTM)               (None, 30, 100)           41200
_________________________________________________________________
dropout_23 (Dropout)         (None, 30, 100)           0
_________________________________________________________________
lstm_28 (LSTM)               (None, 30, 50)            30200
_________________________________________________________________
dropout_24 (Dropout)         (None, 30, 50)            0
_________________________________________________________________
lstm_29 (LSTM)               (None, 50)                20200
_________________________________________________________________
dropout_25 (Dropout)         (None, 50)                0
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 51
=================================================================
Total params: 91,651
Trainable params: 91,651
Non-trainable params: 0
```

Fig 10. Optimizers

The Mean Squared Error is a good loss function but fails to actually do what a “stock predictor” should do which is determine whether the price is going up or down. MSE only indicates how wrong a prediction is from the target. When someone is buying a stock, that person wants to know if the stock is gonna go up or down, not necessarily what the exact future price will be. Because MSE was not fulfilling our needs we decided to design our own loss function to be implemented with the keras system. Upon further research we stumbled upon a guide by Zedric Cheung [11] that seemed to provide a loss function similar to what we wanted. With some minor modifications we were able to produce the Loss function in fig 11.

```python
def customLoss(y_true,y_pred):
    #next days price
    y_true_next = y_true[1:]
    y_pred_next = y_pred[1:]

    #today's price
    y_true_tdy = y_true[:-1]
    y_pred_tdy = y_pred[:-1]

    #Subtract to get whether the price moved up or down
    y_true_diff = tf.subtract(y_true_next, y_true_tdy)
    y_pred_diff = tf.subtract(y_pred_next, y_pred_tdy)

    #create a standard tensor with zero value for comparison
    standard = tf.zeros_like(y_pred_diff)

    #compare with the standard; if true, UP; else DOWN
    y_true_move = tf.greater_equal(y_true_diff, standard)
    y_pred_move = tf.greater_equal(y_pred_diff, standard)
    y_true_move = tf.reshape(y_true_move, [-1])
    y_pred_move = tf.reshape(y_pred_move, [-1])


    #find indices where the directions are not the same
    condition = tf.not_equal(y_true_move, y_pred_move)
    indices = tf.where(condition)

    #move one position later
    ones = tf.ones_like(indices)
    indices = tf.add(indices, ones)
    indices = K.cast(indices, dtype='int32')


    #create a tensor to store directional loss and put it into custom loss output
    direction_loss = tf.Variable(tf.ones_like(y_pred), dtype='float32')
    updates = K.cast(tf.ones_like(indices), dtype='float32')
    alpha = 1000
    direction_loss = tf.tensor_scatter_nd_update(direction_loss, indices, alpha*updates)

    custom_loss = K.mean(tf.multiply(K.square(y_true - y_pred), direction_loss), axis=-1)

    return custom_loss
```

Fig 11. Attempted Custom Loss Function.

After spending a few days trying to troubleshoot the keras library issues and dimensionality issues. We did successfully get this function to work but its loss values were in the billions and trillions and the predictions were wildly off. There were some issues understanding exactly what some of the steps did in the guide we were following making it difficult for us to troubleshoot. Eventually for time's sake we had to move on without this function and move back to MSE. This model in theory calculates a directional loss value. So if the LSTM system incorrectly guesses the direction of the stock price's movement, the more off it is from the direction the higher the loss.

Training the Model:-

Following the Keras documentation we used the model checkpoint function to train our model. For the initial testing we set the epochs to 10 so that it could run

quickly. This was eventually changed because our model needed more iterations before settling on a consistent loss value.

```
checkpointer = ModelCheckpoint(filepath = 'weights3.hdf5',
                               verbose = 2,
                               save_best_only = True)

model.fit(x_train,
          y_train,
          epochs=10,
          batch_size = 32,
          validation_data=(x_val,y_val),
          callbacks = [checkpointer])
```

Fig 12. Fighting the Model

Testing the Model:-

For testing we used three standard tests to determine how well the model was performing. The first was a simple model evaluate function. We looked at not only the test evaluation but the training evaluation as well. This would let us know if our model was overfitting the training data. This also shows a quick representation of how well are model was performing for any particular configuration.

```
model.evaluate(x_train, y_train)
```

```
225/225 [==============================] - 3s 11ms/step - loss: 0.3028

0.30278950929641724
```

```
model.evaluate(x_test, y_test)
```

```
75/75 [==============================] - 1s 12ms/step - loss: 0.6507

0.6507191061973572
```

Fig 13. Evaluating Methods

Plotting the predicted price versus the offered a visual representation of how the model was doing. The closer the two lines were together the better our model performed.

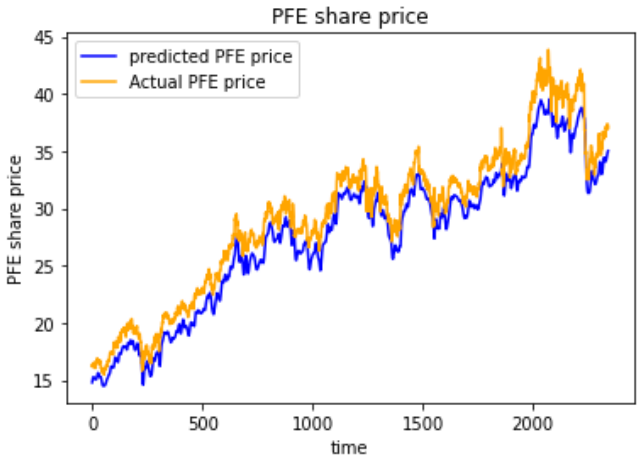


Fig 14. Predicted vs Actual

A prediction test was the last test we used to identify how well our model was performing. This prediction was created by passing the last window of the data to model and predicting the next days value.

```
prediction = model.predict(real_data)
actual_value = load_data(company = 'PFE',
                         start = dt.datetime(2019,12,30),
                         end = dt.datetime(2020,1,3))
print("Closing price prediction for 2020-01-02 ", prediction)
display(actual_value)
```

```
Closing price prediction for 2020-01-02  [[35.490303]]
```

Date	High	Low	Open	Close	Volume	Adj Close
2019-12-30	37.400379	36.897533	37.286530	36.916508	11554264	35.085968
2019-12-31	37.191650	36.726753	36.802658	37.172676	15175703	35.329433
2020-01-02	37.333965	36.888046	37.286530	37.134724	16514072	35.293362
2020-01-03	37.229603	36.688805	36.736244	36.935486	14922848	35.104000

Fig 15. Predicting the Value

By predicting the next day's value for the same day in every test, we were able to see if

our model was accurately predicting the next day's value in a real world application. This also demonstrated the nominal value of the prediction model.

Experiments

Initial Model Evaluations:-

With just the closing cost as the only input feature. 2 LSTM layers, and 10 epochs, the initial model produced a MSE of ~1 meaning that our training predicted value was about 1$ off the actual training value. However our test value was more than 3 off.

```
1 model.evaluate(x_train, y_train)
```

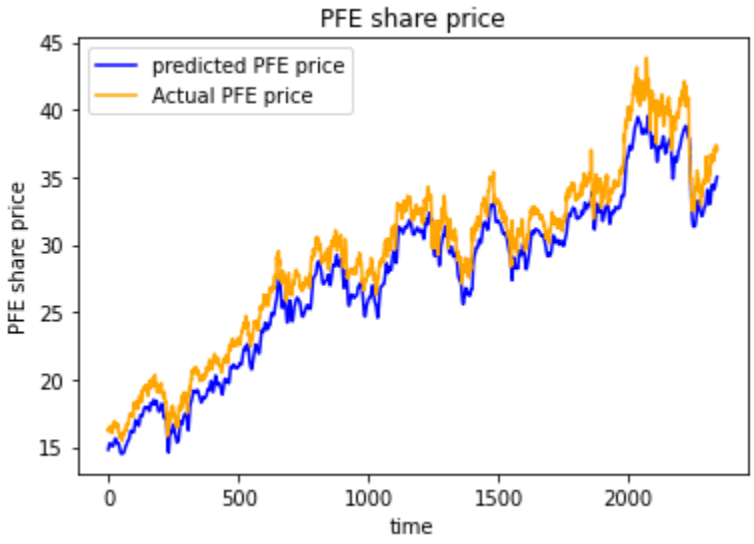
```
224/224 [==============================] - 3s 11ms/step - loss: 0.9160

0.9159723520278931
```

```
1 model.evaluate(x_test, y_test)
```

```
74/74 [==============================] - 1s 12ms/step - loss: 3.6467

3.646731376647949
```



```
Closing price prediction for 2020-01-02  [[34.390303]]
```

Date	High	Low	Open	Close	Volume	Adj Close
2019-12-30	37.400379	36.897533	37.286530	36.916508	11554264	35.085968
2019-12-31	37.191650	36.726753	36.802658	37.172676	15175703	35.329433
2020-01-02	37.333965	36.888046	37.286530	37.134724	16514072	35.293362

Fig 16. Initial Model Evaluations

Initial Modifications to the Model:-

The first thing we attempted to do was increase the number of training iterations. We attempted to add dropout layers at this point, however, this caused the model to perform worse. This dip in performance only accrued when testing with just 1 input feature.

```
1 model.evaluate(x_train, y_train)
```

```
224/224 [==============================] - 4s 18ms/step - loss: 0.6085
0.6085312366485596
```

```
1 model.evaluate(x_test, y_test)
```

```
74/74 [==============================] - 1s 18ms/step - loss: 1.6480
1.6480053663253784
```

Fig 17. Initial Model Improvements

Adding Input Features:-

In order to add input features we had to go back and rework some of our preparation steps to handle more input features. Firstly we started with the combinations of the general stock market information. We found that adding the volume data as an input had performed the best but the result was not significantly better than one input feature.

```
1 model.evaluate(x_train, y_train)
```

```
225/225 [==============================] - 3s 12ms/step - loss: 0.6137

0.613733172416687
```

```
1 model.evaluate(x_test, y_test)
```

```
75/75 [==============================] - 1s 12ms/step - loss: 1.5800

1.5799670219421387
```

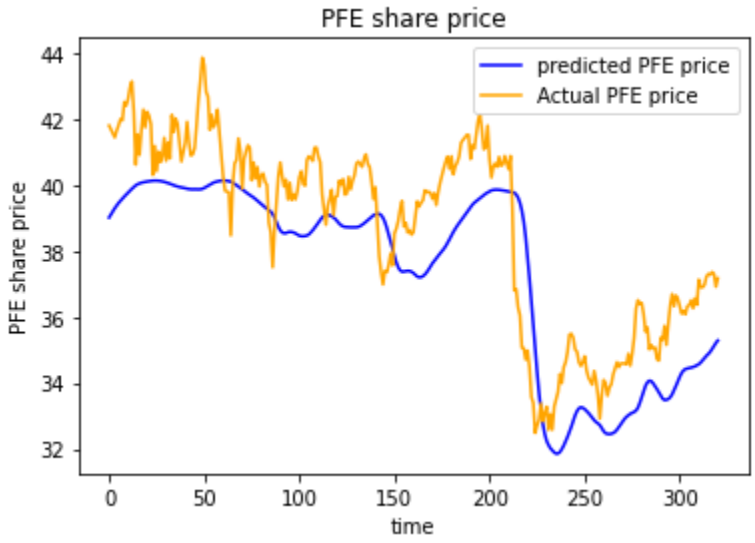


Fig 18. Volume and Closing Input Features

Adding Dropout Layers and LSTM Layers:-

After we added more features we tried adding the dropout layers back in. With more input features it seemed to improve the overall performance of the system. Testing dropout rates of 0.1, 0.2, and 0.3 we found that 0.2 seemed to do the best. On top of the dropout layers we also experimented with adding different numbers of LSTM layers to find the best performance. From our results, 3 layers seemed to be the sweet spot.

```
1 model.evaluate(x_train, y_train)
```

```
225/225 [==============================] - 3s 12ms/step - loss: 0.5791
37

0.579137
```

```
1 model.evaluate(x_test, y_test)
```

```
75/75 [==============================] - 1s 12ms/step - loss: 1.4200

1.4200
```

Fig 19. 3x LSTM Layers and Dropout(.2) Layers

Adding CBOE Volatility Index:-

Adding the volatility index also required some more tweaking for our model so that it could handle 3 input features. However, merging the volatility index with the other initial stock data significantly cut our data rows down from 12000 to 3500 rows. This is most likely the reason we saw a significant decrease in performance. The MSE jumped all the way to 16.5 on the training data. To fix this we tried modifying the split of the train and target samples. We also tried tuning some of the hyperparameters of the train and model but the best result we could get was a MSE training score of about 0.2. 0.2 was the best training MSE score we had achieved thus far however it also came with a test score above 7 which was way higher than our 1.4 from earlier. The loss in data rows seems to outweigh the benefit of adding this extra input feature.

```
1 model.evaluate(x_train, y_train)
```

```
64/64 [==============================] - 1s 22ms/step - loss: 16.4975

16.497535705566406
```

```
1 model.evaluate(x_test, y_test)
```

```
20/20 [==============================] - 0s 22ms/step - loss: 209.7003

209.70028686523438
```

Fig 20. Included Volatility Index

```
model.evaluate(x_train, y_train)
```

```
66/66 [==============================] - 0s 5ms/step - loss: 0.2262

0.22624292969703674
```

```
model.evaluate(x_test, y_test)
```

```
22/22 [==============================] - 0s 4ms/step - loss: 7.9971

7.997062683105469
```

Fig 21. Results with After Tuning with Volatility Index

Building a Custom Loss Function:-

As shown in figure 11, we did try to build a custom loss function. The goal of this custom loss function was to numerically evaluate the direction of the price movement. Unfortunately we were unable to bring this concept to fruition. We ran some tests with our custom loss function but the values were in billion and trillions, with predicted values that were in the negative value range.

Final Version and Performance:-

After finding that including the VIX technical indicator hurt our performance we went back to just the two input features, closing price and volume. We then retested the number of layers as well as some of the hyperparameters to make sure our model was as good as we could make it. Our best performing model is represented in figure 22

```
def LSTM_model():

    model = Sequential()

    model.add(LSTM(units = 100, return_sequences = True, input_shape = (x_train.shape[1],2)))
    model.add(Dropout(0.2))

    model.add(LSTM(units = 50, return_sequences = True))
    model.add(Dropout(0.2))

    model.add(LSTM(units = 50))
    model.add(Dropout(0.2))

    model.add(Dense(units=1))


    return model
```

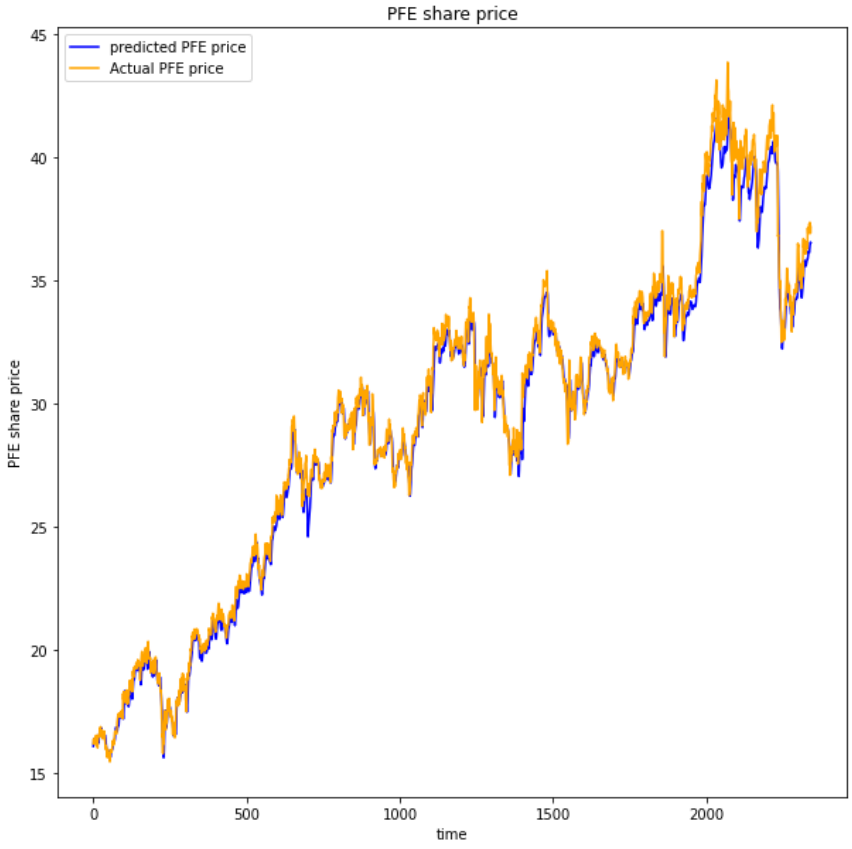
Fig 22. Best Performing model

```
1 model.evaluate(x_train, y_train)
```

```
224/224 [==============================] - 4s 18ms/step - loss: 0.1833
0.18329666554927826
```

```
1 model.evaluate(x_test, y_test)
```

```
74/74 [==============================] - 1s 18ms/step - loss: 0.3715
0.3714931011199951
```




```
Closing price prediction for 2020-01-02  [[37.693867]]
```

Date	High	Low	Open	Close	Volume	Adj Close
2019-12-30	37.400379	36.897533	37.286530	36.916508	11554264	35.085968
2019-12-31	37.191650	36.726753	36.802658	37.172676	15175703	35.329433
2020-01-02	37.333965	36.888046	37.286530	37.134724	16514072	35.293362
2020-01-03	37.229603	36.688805	36.736244	36.935486	14922848	35.104000

Fig 23. Best Performing Model

As the Graph above shows, our final model was able to produce a test loss value of only 0.37. Our next day prediction was also only .5 off of the actual price for that day.

Performance Analysis:-

The most performant model's 0.4 MSE testing loss seems to indicate that our model was performing well. However, when looking at this performance from a practical perspective, our model predicted that from 12-31 to 1-02 the price would rise by 50 cents. When in actuality the price continues to drop for the next few days. This is a trend that is carried out throughout all of the testing predictions. Taking a closer look at the graph in figure 23 shows that our model consistently underpredicted the price. In fact all of our iterations of our model either consistently overestimated or underestimated the price. This is where our attempted loss function becomes more relevant. If the custom loss function would have successfully been implemented. The model would have been more focused on the direction of the price movement instead of the predicted price. This in a practical sense is a more optimal solution. As an investor it is better to know the direction a stock will move, rather the magnitude of price movement. An investor doesn't want to buy at the peak or sell at the bottom. The most important aspect of investing is the direction of the price movement which our model fails to do. Further analysis of the graphs we produced throughout the lab showed that the predicted values we produced always followed the actual values instead of leading them. This all means that this predictive model fails to predict.

Concluding Thoughts

This project was a fun experiment to try and pull off. No one on our team had really tried doing anything like this before but we all had a curiosity to know how it worked. As we were developing this project, we realized the stock prediction research area is far more expansive than we initially imagined. We thought that just plugging in a few simple metrics would generate something that was moderately useful in our personal trading activities. From the data collecting and processing aspect, to the complex technical indicators. Everything was a lot more complex than we had imagined. Using just the stock market data was only the very tip of the iceberg as there are an exponential number of possible variables that play a part in predicting a price. After completing this project, it is still disappointing not to have gotten the custom loss function working properly. It would be nice to know if our loss function would have actually been able to predict the direction of the stock. This was our biggest challenge and also our biggest defeat. We were able to get our loss function to compile and test, however, there were unseen errors with our logic and execution that ultimately let us down. With more development time we have no doubt we would have been able to get this function working and even integrate other metrics to test against

References

[1]. Strader, Troy J.; Rozycki, John J.; ROOT, THOMAS H.; and Huang, Yu-Hsiang (John) (2020) "Machine Learning Stock Market Prediction Studies: Review and Research Directions," Journal of International Technology and Information Management: Vol. 28 : Iss. 4 , Article 3.

[2]. Jaffe, J., Keim, D. B. Keim, and Westerfield, R.: Earnings Yields, Market Values and Stock Returns. Journal of Finance, 44, 135 - 148 (1989)

[3]. Fama, E. F. and French, K. R.: Size and Book-to-Market Factors in Earnings and Returns. Journal of Finance, 50(1), 131 - 155 (1995)

[4]. Chui, A. and Wei, K.: Book-to-Market, Firm Size, and the Turn of the Year Effect: Evidence from Pacific Basin Emerging Markets. Pacific-Basin Finance Journal, 6(3-4), 275 – 293 (1998)

[5]. P. Meesad and R. I. Rasel, "Predicting stock market price using support vector regression," 2013 International Conference on Informatics, Electronics and Vision (ICIEV), Dhaka, Bangladesh, 2013, pp. 1-6, doi: 10.1109/ICIEV.2013.6572570.

[6]. Rosenberg, B., Reid, K., Lanstein, R.: Persuasive Evidence of Market Inefficiency. Journal of Portfolio Management, 11, 9 – 17 (1985)

[7]. Jia H (2016) Investigation into the effectiveness of long short term memory networks for stock price prediction. arXiv [cs.NE], pp 1–6.

[8]. Li Z, Tam V (2017) Combining the real-time wavelet denoising and long- short-term-memory neural network for predicting stock indexes. In: IEEE: 2017 IEEE symposium series on computational intelligence (SSCI), pp 1–8

[9]. Ding, G., Qin, L. Study on the prediction of stock price based on the associated network model of LSTM. Int. J. Mach. Learn. & Cyber. 11, 1307–1317 (2020). https://doi.org/10.1007/s13042-019-01041-1

[10]. Sidra Mehtab , Jaydip Sen and Abhishek Dutta. Stock Price Prediction Using Machine Learning and LSTM-Based Deep Learning Models. arXiv:2009.10819.

[11]. https://towardsdatascience.com/customize-loss-function-to-make-lstm-model-more-applicable-in-stock-price-prediction-b1c50e50b16c

[12]. https://www.analyticsvidhya.com/blog/2018/10/predicting-stock-price-machine-learningnd-deep-learning-techniques-python/

[13]. https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs

Project Repo

https://github.com/aceetheridge/MLProject.git