Project Report

Developing Stock Trading Recommender Using Deep Learning Recurrent Neural Networks

1 Definition

1.1 Project Overview

Previously trading stock market and putting money into investment have been the jobs of investment banks or hedge fund managers. But with assessable APPs (e.g *Robinhood*[1]) available, individuals with a small amount spare money can easily invest the stock or bitcoin market, even do short-term trades.

Machine Learning (ML) especially Deep Learning[2] (DL) has been a powerful tool in dealing with statistical data with patterns and trends, in particular if the data are substantial. It would be foolish to claim that any tool or person can predict the future of the stock market accurately, as it is intrinsically probabilistic. But one can at least give good recommendations based on historical data, if there are inherent dominant deciding factors that can be learned from, even these factors are correlated and complex that cannot be expressed in a single mathematical formalism. The premise of this project is machine learning techniques can be used to inform better investment decisions, thus one's trades/investments are based on data not random guesses.

1.2 Problem Statement

There are so many things one can do with stock market data, since the data themselves can be huge and diverse. We want to delimit our data to daily prices (e.g. close price of that day) and try to predict short- (e.g days) and long-term (e.g. months) price changes

[1] www.robinhood.com

[2] Li Deng and Dong Yu. 2014. Deep Learning: Methods and Applications. Found. Trends Signal Process. 7, 3–4 (June 2014), 197–387. DOI:https://doi.org/10.1561/2000000039

for specific stocks. As the main goal of the model, we will try to focus ourselves in tech sectors (e.g. Microsoft, Google, etc), since more similarities can be found for the stocks from the same sector at given periods of time.

My goal in this project is to develop a recommender that predicts the future stock prices for chosen companies based on their historical data, using DL Neural Networks (NN), specifically Recurrent NN (which has been heavily used in natural language processing[3]) and provide advice on investing and trading.

1.3 Metrics

The basic metrics will be how accurate the model's prediction against the true test historical values. Two accuracy measurements will be chosen:

1. Naïve percentage evaluation: y_predicted – y_test / y_test
2. Root mean square error (RMSE) of the overall prediction against the tests.

The reasons why these two measurements are chosen are as following:

The first measurement is simple, straightforward and intuitive, hence "naïve". It doesn't require much math, giving us quick idea as to if the model works as intended. For example, if this metric goes too big (i.e. ~ 100%), then I may know something is not right (possibly a bug in the code for example). As a first check, this quantity also testifies the second more technical measurement. Plus, if our result (1 day of predicted stock value) has only one data point, the second most sophisticated metric actually does not give much more information, therefore this naïve measurement could be by itself enough.

The second measurement of RMSE[4] is quite standard metric for regression problems, and the stock price prediction problem is eligibly a regression problem. It is measuring the standard deviation of the difference between the predicted values and the actual target values. RMSE says how spread out the predictions are or how concentrated the data is around the line of best fit, which is suitable for our topic in the project.

2 Analysis

2.1 Data Exploration

There are many different resources of historical data of the stock market. Some of them may be proprietary and thus require access fees or memberships. Here for my purpose

[3] Mikolov, Tomáš / Karafiát, Martin / Burget, Lukáš / Černocký, Jan / Khudanpur, Sanjeev (2010): "Recurrent neural network based language model", In INTERSPEECH-2010, 1045-1048.

[4] https://www.kdnuggets.com/2018/04/right-metric-evaluating-machine-learning-models-1.html

of general exploration and analysis, not a full-fledged trading system, *yahoo finance*[5] is chosen, which is free and available via lots APIs (e.g. *Python* module `pandas_datareader`. See code snippet example below).

```
import pandas_datareader as web
df = web.DataReader('GOOGL', 'yahoo', start='2018-01-01',
end='2018-01-31')
```

Using *Google Colab*[6] we could load up the data-frame directly or save it to local drives (e.g. in `csv` format) for easy access. For illustration purposes, I will take Google stock from year 2004 till the present day and the head and tail of the dataset are shown as follows:

```
df.head()
```

Date	High	Low	Open	Close	Volume	Adj Close
2004-08-19	52.082081	48.028027	50.050049	50.220219	44659096.0	50.220219
2004-08-20	54.594597	50.300301	50.555557	54.209209	22834343.0	54.209209
2004-08-23	56.796799	54.579578	55.430431	54.754753	18256126.0	54.754753
2004-08-24	55.855858	51.836838	55.675674	52.487488	15247337.0	52.487488
2004-08-25	54.054054	51.991993	52.532532	53.053055	9188602.0	53.053055

```
df.tail()
```

Date	High	Low	Open	Close	Volume	Adj Close
2021-04-13	2263.469971	2243.050049	2250.989990	2254.429932	1284100.0	2254.429932
2021-04-14	2267.429932	2236.020020	2267.429932	2241.909912	1050100.0	2241.909912
2021-04-15	2296.000000	2251.169922	2261.949951	2285.250000	1458700.0	2285.250000
2021-04-16	2294.239990	2270.919922	2289.239990	2282.750000	1313600.0	2282.750000
2021-04-19	2304.084961	2270.179932	2289.239990	2296.520020	143250.0	2296.520020

[5] www.finance.yahoo.com

[6] https://colab.research.google.com/, online Jupyter notebook style collaboration platformed developed by Google with lots of advantages such as easy-to-use, web-based and free GPU usages.

The summarized dataset information is as follows:

```
df.info()
```

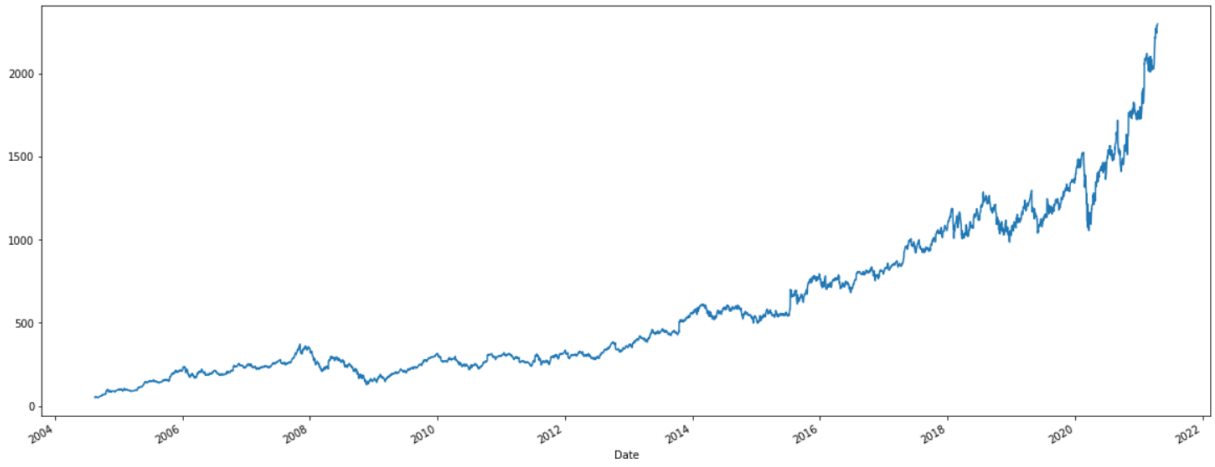
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4195 entries, 2004-08-19 to 2021-04-19
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   High       4195 non-null   float64
 1   Low        4195 non-null   float64
 2   Open       4195 non-null   float64
 3   Close      4195 non-null   float64
 4   Volume     4195 non-null   float64
 5   Adj Close  4195 non-null   float64
dtypes: float64(6)
memory usage: 229.4 KB
```

We can see most of the entries are float values and it contains all the 17 years of trading data of the Google stock. We can also see different columns contain different aspects of stock prices (e.g. Open, High, Close, etc.) at a given date. In this project, I will focus on predicting the "*Close"* prices and simply ignore other prices given they are highly correlated in a single day.

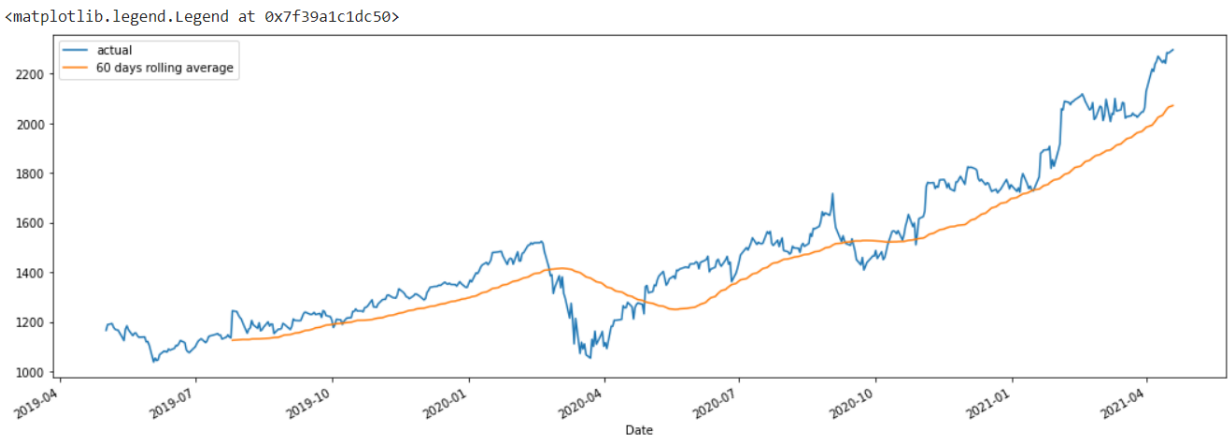
2.2 Exploratory Visualization

Here is a plot of the overall trend of the Google stock over more than 15 years and the prices rises almost 100 folds in particularly for the last 5 years or so. The rise could be driven primarily by a drastic increase of total revenue and good performance of the market overall. We can also see a clear dip of the price in the early 2020 because of COVID-19 pandemic even with such a large-range scale plot.

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f39a2bfd750>
```



We can also look at the rolling average of the price (e.g. orange in the graph for 60 days average moving price versus the actual Closing price) compared to the real stock price of the day from April 2019 till now. The trend of falling during April 2020 to July 2020 and then big rise become even more obvious from this perspective.



2.3 Algorithms and Techniques

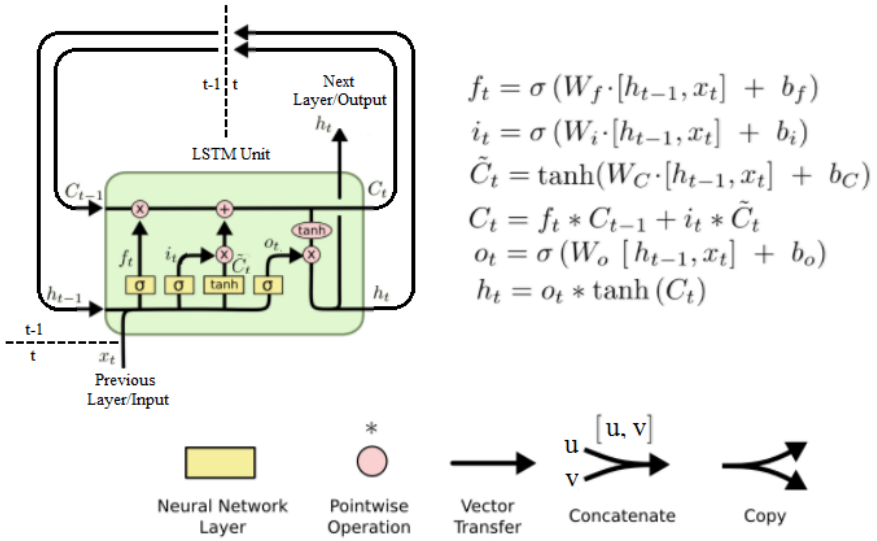
All sorts of different data science and machine learning techniques have been employed to complete this project, including algorithms of traditional non-deep-learning methods and all the data processing and visualization techniques.

Nevertheless, the core algorithm used is long short-term memory (LSTM)[7], which is one type of RNN architecture. With multiple units of these memory cells (graph shown below[8]), RNN can effectively has "memories" that will correlate input remote in time. Therefore, such structures capture data evolution dynamics, performing especially good for time series data.

Specifically for LSTM unit, you have a forget gate (f_t in the illustrated mathematical notation in the graph), that functions as how good the unit "remembers" previous/old states. If it ~ 0 that means the LSTM should forget the state, while close to 1 means it remembers the state very well. Next is the update gate (i_t). It has similar mathematical expression to the forget gate, capturing the essence of how good the unit remembers new states. This effect of the update state, after it is calculated with all the weights and biases will be add up to the forget state (with certain operations) forming final output states (h_t) to the next layer.

[7] Gers. *et al.* "Learning Precise Timing with LSTM Recurrent Networks." Journal of machine learning research : JMLR. 3.Aug (2002)

[8] https://stackoverflow.com/questions/44273249/in-keras-what-exactly-am-i-configuring-when-i-create-a-stateful-lstm-layer-wi



Other traditional regression models (e.g. linear regression) though are widely used in different machine learning fields (e.g. housing price prediction), are not suitable for this stock market prediction problem. The main differentiator between these regression models and LSTM is that these models cannot effectively re-occur or relate to previous states, thus making them depend only on the current timestep, kind like Markovian white noise[9]. Therefore the strength of LSTM is indeed the weakness of these other Markovian regression models (which we will see in the following Benchmark section in detail), which makes LSTM unique.

In order to make LSTM work effectively, one of the main feature engineering techniques used here is using the past n days (e.g. n=30, 60, 90, etc) to predict one day price in the future. This pertains feature engineering to make the “previous” n days prices as labels while the n+1 day the targets[10]. Based on this basic technique, I will further extrapolate future days stock prices, thus forming the core of the stock recommender function.

2.4 Benchmark

It is necessary to benchmark the RNN approach with other traditional ML methods, in particular with those of regression types. Here linear regression(LR), decision tree regression(DTR) and support vector(SVR) machine are selected for benchmark purposes. *Scikit-Learn* has a rich collection of ready-to-go models to be taken advantage of. First, the datetime needs to be converted to numeric values, since the dates are the actual features/labels that the regressors will take in and the targets will

[9] https://stackoverflow.com/questions/45341769/why-should-we-use-rnns-instead-of-markov-models
[10] This technique is inspired by the Youtube post here: https://www.youtube.com/watch?v=PuZY9q-aKLw&list=PLvut403NUF3pGJfxkM-qX7SLV1iAfJJXz&index=3

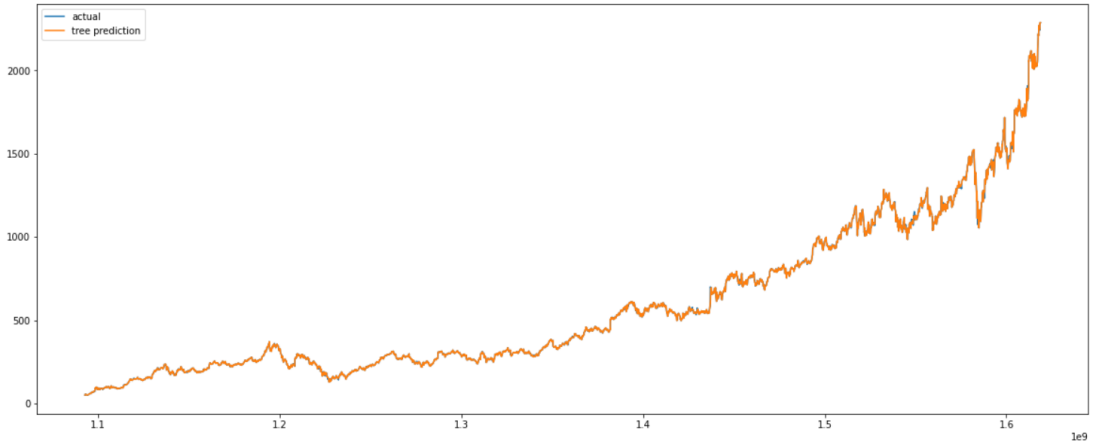
be our *Close* values of stock prices of that day. Then, I cross examine which of these three models perform best with mean squared error measurement.[11]

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error
```

```
for name, model in models:
    kfold = KFold(n_splits=num_folds, random_state=seed,
shuffle=True)
    cv_results = cross_val_score(model, X_train_bench,
y_train_bench, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(),
cv_results.std())
    print(msg)
```

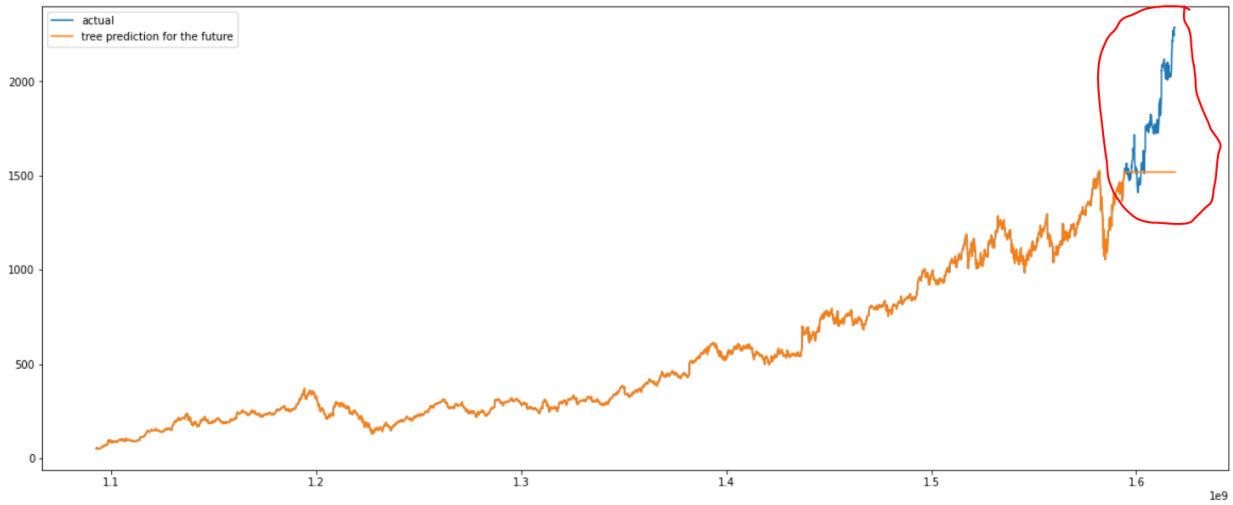
```
LR : 0.821653 (0.018659)
DTR : 0.999111 (0.000305)
SVR : 0.844230 (0.029382)
```

It is found the DTR model perform best (seen from the standard deviation of the cross validation results). Its fitting curve can be plotted as below:

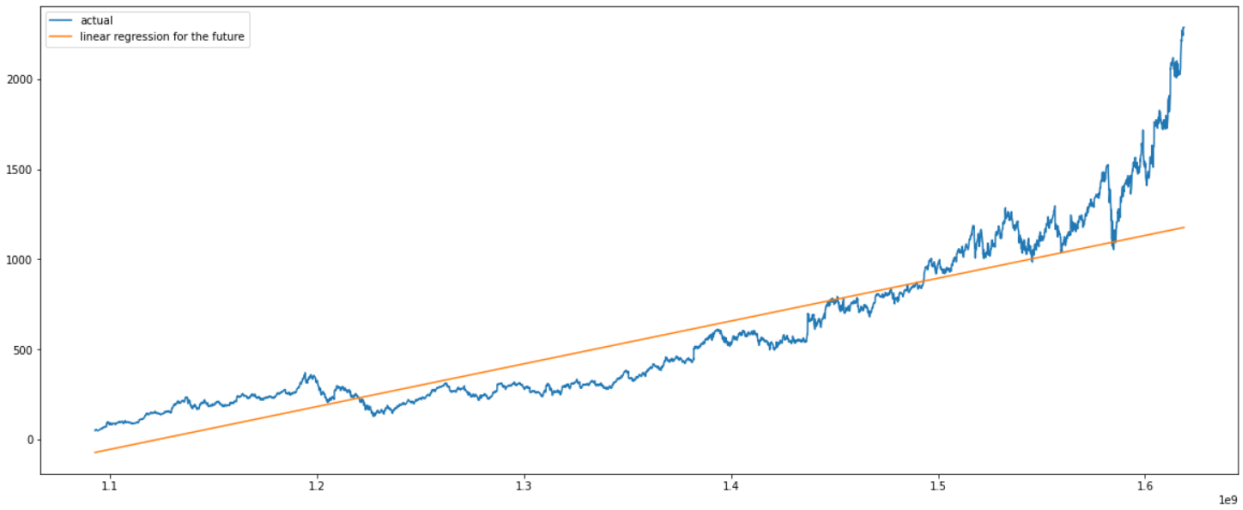


[11] Note here the training and validation datasets are randomized first (see uploaded source code) to get "non-biased" fitting, which is not quite realistic for actual stock price predictions.

From this figure, we can actually see a pretty good fit between the predicted prices and the actual stock prices. But does this mean DTR model is good enough? The answer is unfortunately No. While the plot has a good alignment and the score is high, the test data are still from the past, not from the future! Since during the process of creating training data, the whole dataset is actually randomized, therefore, there is no time difference for the “past” and “future” data, which renders such a prediction useless. After all, what is the point to predict the past stock prices? Therefore, we re-split the train and test data for the past (say 4000 days) days and “future” days, acting as “true” test of the trained model. Here is the adjusted prediction of the later stock price (orange line in the below graph).



We can see from the circled area, the prediction is completely off the mark, showing the inability of the DTR model for predicting the future based solely on the current dates. In comparison LR model (see below) might do better in this sense, as it at least gives us a simple up trend.



Therefore, through these benchmarks, we can conclude that “remember” the history of the prices data is critical for the success of the model, and this is where LSTM-based RNN will shine.

3 Methodology

3.1 Data Preprocessing

After loading data from the web API or from local drive, necessary steps are needed to process the data to give optimal performance of the training processes.

3.1.1 preprocessing

The LSMT model units are sensitive to the (actual) values of the input data, it is a good practice to scale the values of the stock prices, before splitting or train the data. (*Scikit-Learn* `MinMaxScaler` method is used here)

```
scaler = MinMaxScaler(feature_range=(0, 1))
my_data = scaler.fit_transform(df['Close'].values.reshape(-
1,1))
```

3.1.2 Feature creation

As I have alluded earlier, one of feature engineering to be done is taking the past n (integer) days of stock prices as features, and predicting the n+1 day price (target), thus forming properties that mimic memories for LSTM. The following code snippet shows how the function is implemented.

```
def create_feature(data, days):
    X, y = [], []
    for i in range(days, len(data)):
        X.append(data[i-days:i, 0])
        y.append(data[i, 0])
    return np.array(X), np.array(y)
```

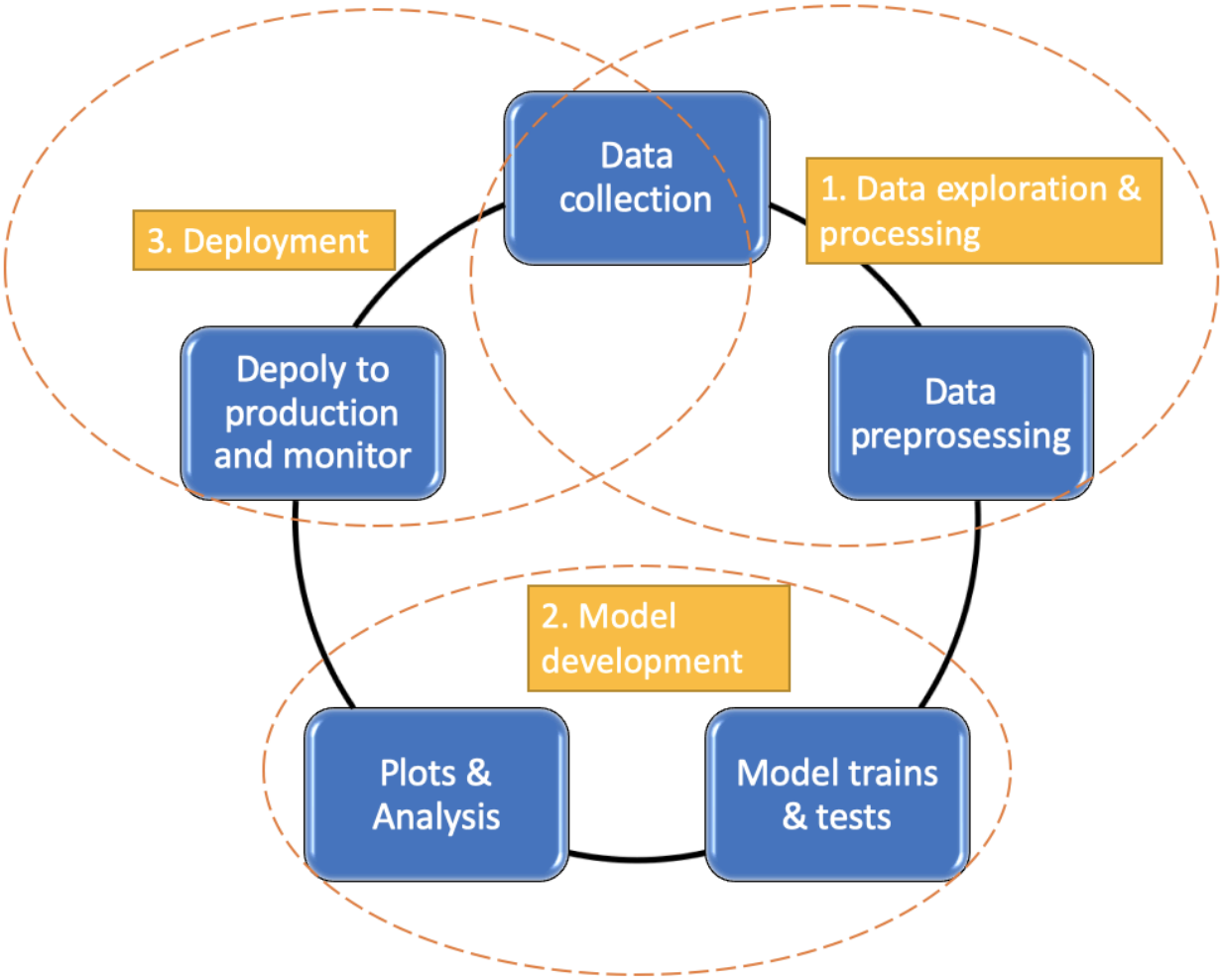
3.2 Implementation

The details of all the implementations can be found in the uploaded source code. Here a diagram is given, showing from a high-level how the project is carried out (as the graph shown below) :

1. I will collect the historical stock prices data via online platforms or *Python* APIs, possibly with specifications just to delimit how much data we want to download. Meanwhile we might also want to explore, visualize a bit of the raw data see if there are any patterns or anything worth noting.
2. I need to preprocess the data (e.g. drop all NULL entries) and scale properly of the data for training purposes. If needed, we will also engineer certain features to give better representations of the data or reduce the dimensions. Lastly, we need to split the data into train and test sets (or maybe validation set as well) for the convenience of the next step.
3. I will select different models (one of them will be LSTM) and train them for validation and benchmarks. We will utilize *Google Colab* notebook since it provides free GPU support, which could be useful and potentially make the train faster. *Tensorflow/Keras* framework will be used for the design of layers of NN. After training we will test our model, and do whatever visualization and analysis needed.
4. After multiple stocks are specified and our trained recommender will predict and compare between prices and gains, and then output the best pick and trade one might have with these tech stocks.
5. In principle, it is possible to use *ColabCode* companied *Colab* to deploy our developed recommender[12], so that one can enter specific tech stock, and return recommendations on future buy or short options. We leave this API launch for future tasks instead of exhausting everything in this one project.

```
def create_RNN(shape, units):
    model = Sequential()
    model.add(LSTM(units=units, return_sequences=True,
input_shape=shape))
    model.add(LSTM(units=units, return_sequences=True))
    model.add(LSTM(units=units))
    model.add(Dense(units=1))
    return model
```

[12] Refer to the Medium post: https://towardsdatascience.com/colabcode-deploying-machine-learning-models-from-google-colab-54e0d37a7b09



3.3 Refinement

We will need some fine tunning of hyper-parameters of the model (or the architecture of NN), such as unit numbers, stacked layer numbers, and dropout layers and so on. There are no definitive answers as to which refinements are the best, as many good choices come from experiences. One example of the alternatives (I tried many others for multiple times) I take is to insert dropout layers (with certain dropout layers, 0.2 in this case) between each LSTM layers, mainly to avoid overfitting and make the model more applicable and general.

```
def create_RNN(shape, units, dropout_rate):
    model = Sequential()
    model.add(LSTM(units=units, return_sequences=True,
input_shape=shape))
    model.add(Dropout(dropout_rate))
    model.add(LSTM(units=units, return_sequences=True))
    model.add(Dropout(dropout_rate))
    model.add(LSTM(units=units, return_sequences=True))
    model.add(Dropout(dropout_rate))
    model.add(LSTM(units=units))
    model.add(Dropout(dropout_rate))
    model.add(Dense(units=1))
    return model
```

In general, adding more layers or units will make the model more sophisticated, but at the meantime will also make the training more expensive. Depending on specific problem, the trade-off might not necessarily favor added complexity. In this case, I found 3 to 4 NN layers are sufficient for the purposes.

4 Results

4.1 Model Evaluation and Validation

After the training (with model information summarized in the graph) is done, I look at the test data (which has 575 input entries).

```
[55] my_model.summary()

     Model: "sequential"
     _________________________________________________________________
     Layer (type)                 Output Shape              Param #
     =================================================================
     lstm (LSTM)                  (None, 120, 50)           10400
     _________________________________________________________________
     lstm_1 (LSTM)                (None, 120, 50)           20200
     _________________________________________________________________
     lstm_2 (LSTM)                (None, 50)                20200
     _________________________________________________________________
     dense (Dense)                (None, 1)                 51
     =================================================================
     Total params: 50,851
     Trainable params: 50,851
     Non-trainable params: 0
```

And inspecting the two measurements we have defined above, i.e. Naïve percentage evaluation: y_predicted – y_test / y_test and Root mean square error (RMSE) of the overall prediction against the tests.

```
[93] test_pred_back = scaler.inverse_transform(test_pred)
     y_test_back =scaler.inverse_transform(y_test)
     relate_diff = abs(np.sum(test_pred_back-y_test_back))/np.sum(y_test_back[:,0])
     print(relate_diff)

     0.018744424754768l9
```

```
[91] smse = np.sqrt(mean_squared_error(y_test,test_pred))
     print(smse)

     0.02268463886067224
```

We can see these two metrics are both small and close to each other, indicating the model perform well against the test data.

4.1.1 Predicting far into the future

Now let us put our model into test in predicting future stock indices. In order to extrapolate beyond 1 day prediction, we need “feed-back” our predicted data into label/training data. The following *Python* function I created does this job perfectly.

```
def predict_future(model,future_day,input_X):
    theshape = input_X.shape
    if future_day == 1:
        pred = model.predict(input_X)
        return scaler.inverse_transform(pred)[:,0]
    output = []
    for day in range(future_day):
        if day == future_day - 1:
            pred = model.predict(input_X)
            pred_inverse = scaler.inverse_transform(pred)[:,0]
            output.append(pred_inverse.tolist())
        else:
            pred = model.predict(input_X)
            pred_inverse = scaler.inverse_transform(pred)[:,0]
            output.append(pred_inverse.tolist())
            input_X =
np.append(input_X[:,1:,:],pred).reshape(theshape)

    return np.array(output)
```

Also we need re-train the model using the whole dataset (till today(04-19-2021)), so that we don't have any reference to the future prices but solely rely on our own predictions (it might be interesting to come back later in the future to see how accurate our predictions are).

For Google, using the future prediction function we defined above, one day apart (that is on 04-20-2021), the stock (Close) price will be $ 2311.4888, going up!

```
days = 180
X_future = X_train_future[-1,:,:].reshape(1,days,1)
d_future = 1
my_pred = predict_future(my_model_future,d_future,X_future)
print(my_pred)
```

```
[2311.4888]
```

Waiting for a week, my model predicts the price goes up to $2434.62 (~5% increase, should I buy it now?!).

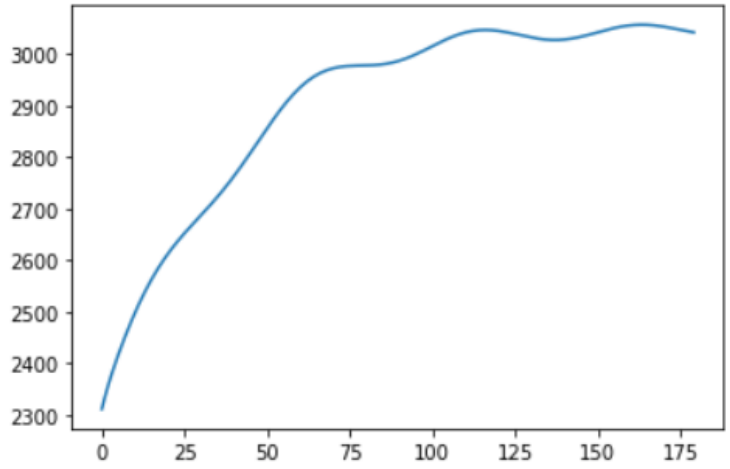
```
days = 180
X_future = X_train_future[-1,:,:].reshape(1,days,1)
d_future = 7
my_pred = predict_future(my_model_future,d_future,X_future)
print(my_pred)
```

```
[[2311.48876953]
 [2336.20483398]
 [2358.8671875 ]
 [2379.64208984]
 [2398.94189453]
 [2417.18164062]
 [2434.62329102]]
```

Optimistic as it may, the model predicts a "plateau" (see the snippet below) about 100 days (3 months) later. So probably I can do a mid-term investment on Google for three months?

```
d_future = 180
my_pred = predict_future(my_model_future,d_future,X_future)
plt.plot(my_pred)
```

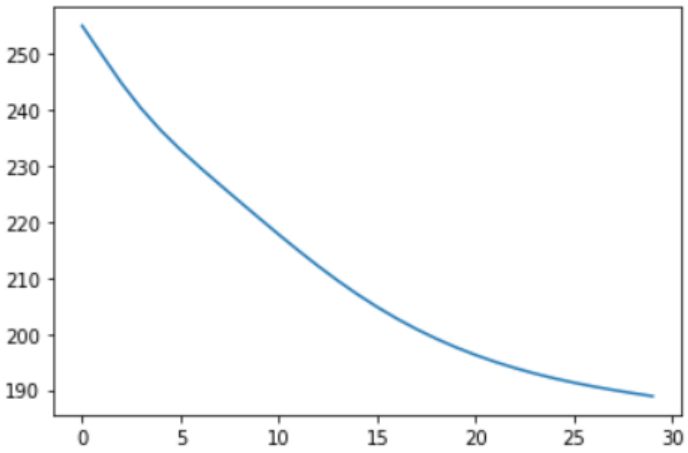
```
[<matplotlib.lines.Line2D at 0x7fb6bf0f3790>]
```



To fulfill the duty of a recommender, I also play around with two other tech stocks, namely, Microsoft and Shopify (snippets below respectively).

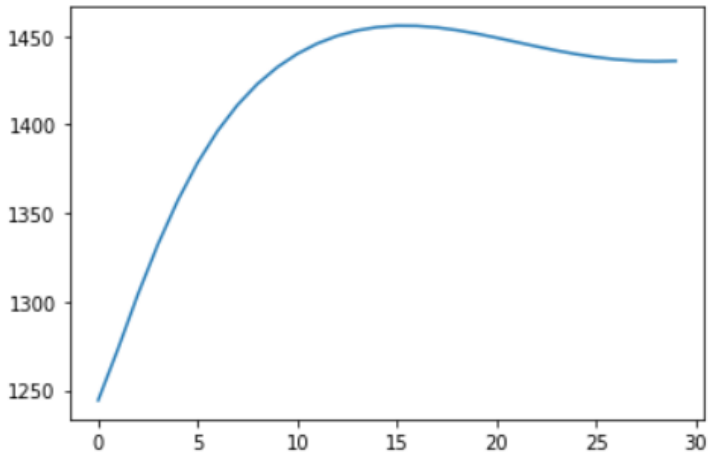
```
d_future = 30
my_pred = predict_future(my_model_future,d_future,X_future)
plt.plot(my_pred)
```

```
[<matplotlib.lines.Line2D at 0x7fb6b8f9bb50>]
```



```
d_future = 30
my_pred = predict_future(my_model_future,d_future,X_future)
plt.plot(my_pred)
```

```
[<matplotlib.lines.Line2D at 0x7fb6b941b410>]
```

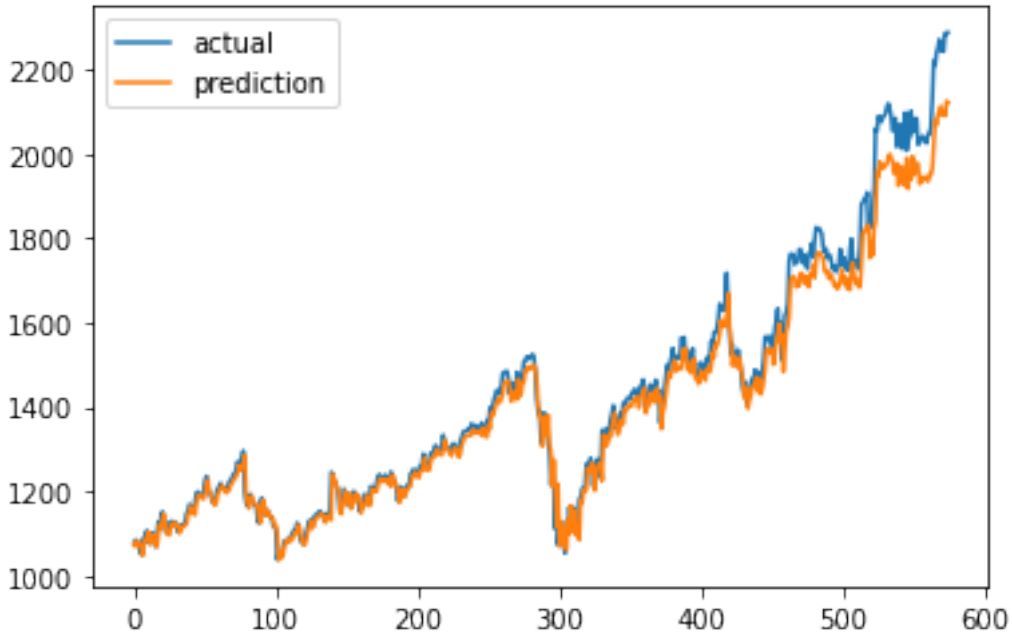


I set my recommenders criteria to be: 1. Pick the stock that goes up (for certain days apart); 2. Pick the stock that gives highest percentage gain (among all the provided stocks, in this case: Google, Microsoft and Shopify); 3. Return the recommended days/period that gives maximum gain, i.e. biggest upward slope.

Based on these criteria, my stock pick among these three examples would be **Shopify** for a 15 days period with almost 26% gain (buy on 04-19-2021 and sell on 05-07-2021, omitting weekends).

4.2 Justification

By further looking at the comparison between the predicted and actual prices, the model gives a remarkable prediction. When we compare this result to the benchmarked models (LR and DTR) above, the power and superiority of LSTM-based RNN approach is clearly shown.



5 Conclusion

A certain degree of success has been achieved in this project of developing stock market price predictor and trading recommender, mainly by utilizing Deep Learning recurrent neural network architecture. I have managed to predict one day stock market closing price based on the preceding multiple days (e.g. 120 days) historical prices with high accuracy, especially compared to low-performance traditional ML methods (e.g. regression flavors).

In addition, I have extended the model capacity by extrapolating stock prices for the future days based on the input stock ticker. By comparing the stock prices and their changes for different companies, the recommender is able to provide best trade options for given stocks. It is still a bit elementary model, since it only considers the past prices. But still, it could give average investors (like myself) some educated directions as to when and what to invest on the stock market (similar model can be constructed for Bitcoin market as well), which is both interesting and helpful.

5.1 Reflection

By installing packages such as `pydantic`[13], `fastAPI`[14], on the notebook, combining `ColabCode` in Google Colab, I should be able to deploy my model to the cloud. But

[13] https://pydantic-docs.helpmanual.io/
[14] https://fastapi.tiangolo.com/

there could be web related issues (e.g. stability of the pipeline, performance, feeding and update of the model) that may not easily addressed in this project. Nonetheless, it could be constructed with certain amount of time, possibly optimizations of the API and interfaces as well.

5.2 Improvement

To develop a fully-fledged stock market recommender, that is an infrastructure that can be used to advice actual trading in institutional settings (could be implemented by a group of machine learning engineers and software developers), one still needs:

1. More features (e.g. seasonal information including timestamps of quarterly earnings, big announcements, tweets, etc.) and proper data selections.
2. Accompanying these new features of data is drastically increased data volume, so big data tools, such as Spark, Hadoop, might be needed, possibly collaborations with data engineers.
3. Fast execution of DL models with automated trading systems.
4. Diversified portfolios design that automated by combined ML algorithms with built-in correction mechanisms.

All these implementations can be costly both in terms of manpower and resources. Nonetheless, in order to make profits these costs are also investments. Maybe they themselves should be taken into account as inputs to the machine learning model as well? If this is the trend, we might see robots dominate Wall Street in the near future.