Time-Series Forecasting of Stock Prices

CS-C3240 Machine Learning

Introduction

Many data sets feature time dependencies that traditional machine learning (ML) models cannot adequately capture. Time-series models allow for capturing that time dependency and predicting future values based on previously observed values. The time-series data, also known as non-stationary input data, contains statistical properties, where the standard deviation and the mean vary over the time. Some examples of time-series include stocks prices, house prices or temperature values over time.

In this report, I will present a Long Short Term Memory (LSTM) model to predict a value of Google (Alphabet) stocks based on their previous values (with a lag of 1 day). LSTMs, as the name suggests, has a capability of remembering information for a long period of time by applying LSTM gates in a recurrent neural network (RNN), which allows for a much more efficient and predictive use case of forecasting stock prices. Other methods like ARIMA or stochastic modelling could also be used for this example, but due to the long-term capabilities of LSTM, I find this model very efficient for the purpose of this report. Using simpler machine learning models like *Regression* or *Binary Trees* would not be able to capture the important time dimensionality of the data as adequately as LSTMs. Therefore, I am only going to used one model, but I am going to apply two slightly different versions of it for comparison and discussion purposes.

The Problem Formulation section will define the potential sources of the data points for the model. *The Method* section will present two slightly differently LSTM models, and everything is implemented using popular Python libraries, such as numpy, pandas, matplotlib and keras. *The Results* section will compare and discuss the obtained results. Finally, *the Conclusion* section will summarize the main findings as well outline suggestions for future improvements.

Problem Formulation

For the purpose of this report, I am going to use Alphabet Inc. (GOOG) data stocks on the previously introduced models. Datasets for stock values are easily found on any finance-related website, like Yahoo Finance (https://finance.yahoo.com/), Google Finance (https://www.google.com/finance). Websites like these allow you to choose any particular date interval and download its dataset as a .csv file. The dataset that I used is from Yahoo Finance, and its date interval is from 24th February of 2015 to 23rd February of 2021. The dataset from Yahoo Finance contains comma separated columns, which looks like this:

In [40]:
```
# Reading in the file
data = pd.read_csv('google_stocks.csv')
# data.shape = (1511, 7)
# Displaying the first entries
data.head()
```

Out[40]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2015-02-24	528.548889	535.320251	526.803650	534.622192	534.622192	1005052
1	2015-02-25	534.432739	544.724487	533.978943	542.380920	542.380920	1826000
2	2015-02-26	541.722717	554.617310	540.017395	553.959106	553.959106	2311529
3	2015-02-27	552.722473	563.163818	551.386169	556.871094	556.871094	2410199
4	2015-03-02	558.995300	570.583435	557.220154	569.775696	569.775696	2129631

The datasets like this consist of many columns, from which the most interesting are the Date and the Close columns. Close column refers to the stock closing price at a given date (in USD $). In this example, I am going to use the column Date as my features (easily measured property) and the column Close as my labels (quantity of interest).

There are 1511 features and labels in total. The data will be divided into training and test datasets. The training set will consist of 80% of the total dataset, and the remaining 20% will be the testing set.

The loss functions used in this problem is the Mean Squared Error. I decided to use it due its strength when dealing with outliers, which thus allows for a more accurate forecasting of the daily stock prices. The optimizer function used is 'Adam optimizer'. I chose it due its great stochastic gradient descent method, which particularly works well in the field of deep learning.

Method

We start by preparing and assigning the necessary data into separate variables.

In [25]:
```
# Dividing the data
train_len = math.ceil(len(data) * 0.8) # 1209

# Dividing our dataset into training
training_data = data.iloc[:train_len, 1:2].values
```

Later, we are going to normalize all the training_data variables, which is done to increase the model efficiency. Our model is going to use 60 time-stamps, meaning that we are going to create x_train and y_train variables, where x_train is going to hold its 60 last values, and y_train will hold the resulting value. In practice, it means that in order to predict our output, we need 60 last inputs.

In [26]:
```
# Normalizing the data
sc = MinMaxScaler(feature_range = (0, 1))
sc_training_data = sc.fit_transform(training_data)

# Creating a data structure with 60 time-stemps with only 1 output
x_train = []
y_train = []
for i in range(60, train_len):
    x_train.append(sc_training_data[i-60:i, 0])
    y_train.append(sc_training_data[i, 0])

# Transforming into a numpy array
x_train, y_train = np.array(x_train), np.array(y_train)

# Reshaping the data
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
# x_train.shape = (1149, 60, 1)
```

Then, we are going to create a few datasets. Some of them will be used for visualization, and the variable dataset will be used to create x_test later in the program.

In [27]:
```
# Creating datasets for later visualizations and predictions of the model
dataset_train = data.iloc[:train_len, 1:2] # From 0 to train_len
dataset_test = data.iloc[train_len:, 1:2] # From train_len to the end
dataset_total = pd.concat((dataset_train, dataset_test), axis = 0) # Putting them together

dataset = dataset_total[train_len - 60:].values # For prediction
dataset = dataset.reshape(-1,1)
dataset = sc.transform(dataset) # Normalizing it
```

Creating, the x_test variable will happen as with x_train. We will append its last 60 values.

In [30]:
```
# Creating a test array for the prediction using the previously assigned dataset
x_test = []

for i in range(60, len(dataset)):
    x_test.append(dataset[i-60:i, 0])

# Transforming into a numpy array and reshaping it
x_test = np.array(x_test)
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))
x_test.shape # (302, 60, 1)
```

Each model will consist of 4 hidden layers with 50 neuron each and one output layer. The second model will have added dropout regularization, increased epochs and batch sizes. We train the model

using x_train and y_train variables. Dropout regularization is a technique where some random neurons are ignored, or 'dropped-out', during the modelling phase. The batch size is a number of samples processed before the model is updated. The number of epochs is the number of complete passes through the training dataset. Changing these hyperparameters has an effect on the final result of the model.

```
## Model 1
# The LSTM model with 50 neurons and 4 hidden layers
model = Sequential()
# Four hidden layers
model.add(LSTM(units=50, return_sequences = True, input_shape = (x_train.shape[1], 1)))
model.add(LSTM(units=50, return_sequences = True))
model.add(LSTM(units=50, return_sequences = True))
model.add(LSTM(units=50))

model.add(Dense(units = 1)) # Output layer
model.compile(optimizer = 'adam', loss = 'mean_squared_error') # Compiling the model
model.fit(x_train, y_train, epochs=1, batch_size=1) # Fitting the data set
```

```
## Model 2
# The LSTM model with 50 neurons and 4 hidden layers
model2 = Sequential()
# Four hidden layers with dropout regularization
model2.add(LSTM(units=50, return_sequences = True, input_shape = (x_train.shape[1], 1)))
model2.add(Dropout(0.2))
model2.add(LSTM(units=50, return_sequences = True))
model2.add(Dropout(0.2))
model2.add(LSTM(units=50, return_sequences = True))
model2.add(Dropout(0.2))
model2.add(LSTM(units=50))
model2.add(Dropout(0.2))

model2.add(Dense(units = 1)) # Output layern
model2.compile(optimizer = 'adam', loss = 'mean_squared_error') # Compiling the model
model2.fit(x_train, y_train, epochs=100, batch_size=32) # # Fitting the data set
```

Once our models finish compilling and fitting our training data, we can use training sets for checking purposes and testing sets for predicting purposes to predict our new predicted prices. We also need to inverse_transform the result, as the result out of the model is normalized.

```
# Model 1
predicted_price1 = model.predict(x_test) # Prediction based on testing data
predicted_price1 = sc.inverse_transform(predicted_price1)
predicted_price11 = model.predict(x_train) # Prediction based on training data
predicted_price11 = sc.inverse_transform(predicted_price11)

# Model 2
predicted_price2 = model2.predict(x_test) # Prediction based on testing data
predicted_price2 = sc.inverse_transform(predicted_price2)
predicted_price22 = model2.predict(x_train) # Prediction based on training data
predicted_price22 = sc.inverse_transform(predicted_price22)
```

Each of the predicted_price returns an array of stock prices, and thus the LSTM method is finished.

Results

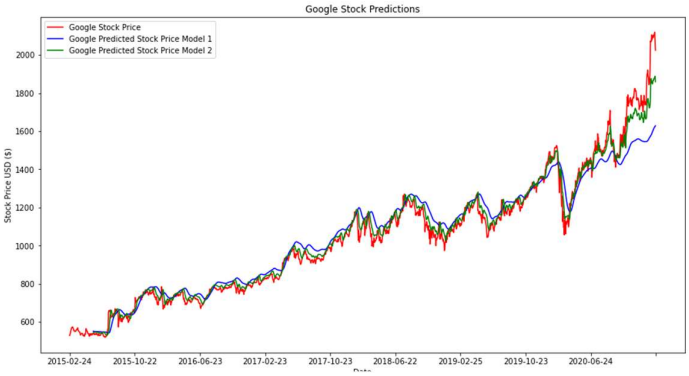
Before we plot all of our results, we are going to calculate the loss value using the mean squared error function.

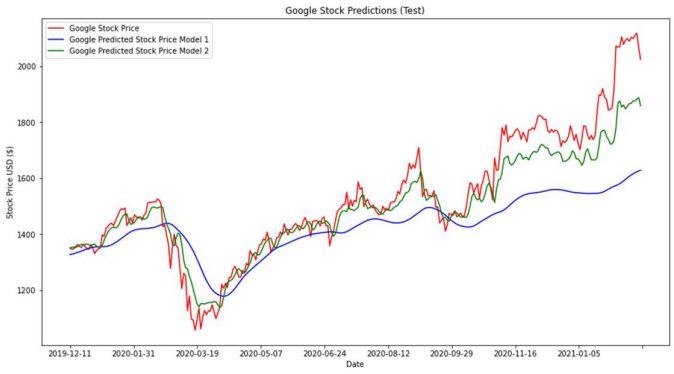
```
## Error loss
# Model 1
mse1 = np.mean((predicted_price1 - dataset_test)**2) # From the testing set, # 63494.042577
mse11 = np.mean((predicted_price11 - dataset_train[60: ])**2) # From the training set, # 3771.916702

# Model 2
mse2 = np.mean((predicted_price2 - dataset_test)**2) # From the testing set, # 7995.087714
mse22 = np.mean((predicted_price22 - dataset_train[60: ])**2) # From the training set, # 458.328947
```

As expected, the error from the training dataset (on which the model is trained), is much smaller than the error from the testing dataset. However, it can already be observed that the second model has an almost tenfold improvement in the error.

We are going to plot two graphs, first of the entire datasets and the second of the testing set.





Conclusion

Overall, both models were able to approximately predict the stock prices from the training data (which is not a surprise). The more important part is how those models were able to handle the testing data. As mentioned above, the error from the second model is much smaller than from the first model. That is a direct result of applying better hyperparameters (dropout, batch size and epochs). Those hyperparameters, in particular, allow for a better optimization of the model as they change the rate at which the neurons are updated in the network (epochs and batch size), as well as create some kind of randomness (dropout) to create more bias in the model and decrease overfitting of the model. Just by adding the dropout hyperparameter set at 0.2 in the first model, we can decrease the error on the testing data by more than a half.

The model is still not perfect. The possibility of changing and adjusting neural networks is infinite, and finding the right parameters for the right problem is a key in creating the most optimized system. One possible reason for which the model has a potentially much bigger error from the testing data is the global economic crash which is due the COVID-19 pandemic. Many of the stock prices of big companies and corporations suffered due the pandemic, and that is clearly visible on any stock price graph. That still does not erase the fact that LSTM is one of the most suitable models for this kind of problem.

One of the improvements that can be implemented is considering all the values from a stock market, like ‘Open’, ‘High’, ‘Low’, ‘Close’ and etc. This would allow the model to work with multiple features, which could potentially increase the accuracy of the predictions. Another improvement could be by playing around with the number of time-stamps. Certainly trying every possible number would take a lot of computational power (as the models themselves compile for around 2-5min using Aalto JupyterLab).

This brings the reason for which Deep Learning is becoming a popular field. The computational power of all of our devices is constantly increasing, which thus allows for much faster computations of any neural network, which in turn allows for more possibilities when optimizing the model.

Bibliography

- Chollet, F. (2018). Deep learning with Python. Shelter Island, NY: Manning Publications.
- Budhiraja, A. (2018, March 06). Learning less to learn better - dropout in (deep) machine learning. Retrieved February 25, 2021, from https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5
- Brownlee, J. (2019, October 25). Difference between a batch and an epoch in a neural network. Retrieved February 25, 2021, from https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/
- Brownlee, J. (2020, August 27). Time series prediction with LSTM recurrent neural networks in Python with keras. Retrieved February 25, 2021, from https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/
- Dataset: https://finance.yahoo.com/quote/GOOG/
- My code (prepared for this project): https://github.com/Dmkk01/LSTM-Stock-Prediction