1 Stock Price Prediction

1.1 Dataset

Firstly, use *read_csv* function from *pandas* to load the original dataset, then use the *.iloc* method to select features and labels for the dataset, where the data from the columns **Open**, **High**, **Low**, and **Volume** are used as features and the data from the column "Open" is labels. As the data from each column varies a lot, especially the **Volume**, it needs to be normalized to help the model converge quicker to find the local or global minimum. *MinMaxScaler* imported from *sklearn* is used to transform each feature individually into the same range. After normalization, two empty lists, features and labels, are created to store values, and a for-loop is conducted to generate samples using a 3 window size. Each sample contains three rows (12 values in total) as features and uses the previous row's **Open** value as its label. Finally, convert the list features and list labels into NumPy array to feed into the LSTM network.

In the testing set, almost the same preprocessing steps are done on the *test_data_RNN.csv* file, except the MinMaxScaler is applied to the training set first to fit the training data, then the scaler is applied on the testing data.

1.2 Model

In the final model, the first layer is an LSTM layer with 100 units. The second layer is a Dense layer with 20 units using *Relu* as activation function to improve performance, and the output layer is a normal Dense layer with one unit using *Linear* activation as this is a regression task, not a classification task, just needs to output the prediction values.

Mean_absolute_error is used to calculate the loss in each epoch, and the optimizer used in this model is *Adam*. Totally 50 epochs are run to obtain the final model.
In the design procedure, only 1 LSTM layer with 32 units was used at first, following the idea that starting with a simpler model. As the model cannot achieve low loss after running many epochs, a Dense layer using *Relu* was added to improve the model's performance and the number of units was increased to 100 in the LSTM layer. At the same time, to avoid overfitting, a Dropout layer with a dropout ratio of 0.1 was added between the two layers mentioned above. But by observing the final loss in both training and testing, this model actually didn't occur overfitting with the training data. So, the Dropout later was removed in the end, and the final loss dropped to 3 from 6.

1.3 Results

The training loss after 50 epochs is around 3.2, which is 63 at the beginning (first epoch). Actually the loss already reaches around 3.5 at the 20 epoch and dropping extremely slowly after that. To achieve higher accuracy, the architecture of the model need to be tuned instead of running more epochs.
The loss of testing data is 3.9 using the stored model. This value is expected, meaning that this model hasn't overfitted the training set. And fig.1 shows the plot of the true and predicted value, in which the predicted values follow the general trend but differences do exist in each sample.

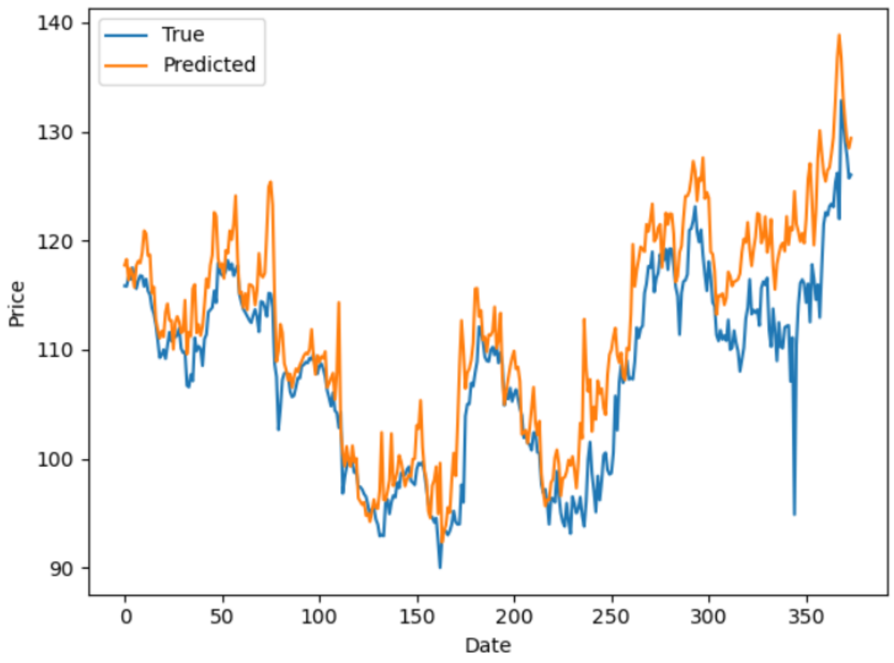

Figure 1: True and predicted values

1.4 Using More Features

Using the same model, only change the window size to 10 when generating data samples, that is to say, using ten days of data as one sample's features. The running time would increase as more features are used, but at the same time, the loss might drop under the same epoch.

2 Problem 3

2.1 Preprocessing

In the *train_NLP.py* file, firstly load the original dataset from *./data/aclImdb* folder and create two empty lists to store positive samples and negative samples as they come from different files. Then use the *gen_set* function to generate each sample from */train/pos* and */train/neg* respectively and put them into the corresponding positive list or negative list (using *.append()*) method. Combine the positive list and negative list to form the original training set. After having the original training set, the training labels are created by *np.ones* and *np.zeros*, in which positive samples obtained labels 1 and negative samples obtained labels 0. 20000 was chosen as the vocabulary size parameter, meaning that only the first 20000 words with the highest frequency would be retained. And 100 is the max length size parameter, that is to say, every sample would have the same length, 100 words. Then, conducting text vectorization, using *texts_to_sequences* method from *Tokenizer* to transfer the text into numbers. And as some sentences are shorter than 100 words themselves or some sentences contain less than 100 words after filter the low-frequency words, the padding step is conducted to the obtained sequences. In the *test_NLP.py* file, the same preprocessing steps are implemented to the testing set. In the text vectorization procedure, as testing samples need to be fit in the same tokenizer as the training samples, the tokenizer is generated again in *test_NLP.py* file.

2.2 Model

The first layer is an *Embedding* layer using 32 as the word vector dimension parameter. *Embedding* layer provides the same effect as a fully connected layer who has conducted one-hot encoding to the input data, thus improving training efficiency. As the output of the *Embedding* layer is a 2D vector, in which each word in the input sequence of words has one embedding. A *Flatten* layer is added after the *Embedding* layer to flatten the 2D matrix into a 1D vector to feed into the next *Dense* layer. After the *Embedding* and *Flatten* layers, a *Dense* layer with 256 units using *Relu* activation function is added. And

finally, the output layer with 1 units using *Sigmoid* function is added to the NLP model to complete the binary classification task.
Totally 4 epochs are run on training set, with a 512 batch size using *adam* optimizor and *binary_crossentropy* loss function. And the model is saved as 20862738_NLP_model.h5.

2.3 Accuracy

The final training accuracy reaches 98.7% after 4 epochs which seems kind of overfitting, but the testing accuracy is around 84% which is not very bad.